Polygraph: Automatically Generating Signatures for Polymorphic Worms

James Newsome
Carnegie Mellon University
jnewsome@ece.cmu.edu

Brad Karp
Intel Research Pittsburgh
brad.n.karp@intel.com
Carnegie Mellon University
bkarp+@cs.cmu.edu

Dawn Song
Carnegie Mellon University
dawnsong@cmu.edu

Abstract

It is widely believed that content-signature-based intrusion detection systems (IDSes) are easily evaded by polymorphic worms, *which* vary *their payload on every infection attempt. In this paper, we present Polygraph, a signature generation system that successfully produces signatures that match polymorphic worms. Polygraph generates signatures that consist of* multiple disjoint *content substrings. In doing so, Polygraph leverages our insight that for a real-world exploit to function properly, multiple* invariant *substrings must often be present in all variants of a payload; these substrings typically correspond to protocol framing, return addresses, and in some cases, poorly obfuscated code. We contribute a definition of the polymorphic signature generation problem; propose classes of signature suited for matching polymorphic worm payloads; and present algorithms for automatic generation of signatures in these classes. Our evaluation of these algorithms on a range of polymorphic worms demonstrates that Polygraph produces signatures for polymorphic worms that exhibit low false negatives and false positives.*

1. Introduction and Motivation

Enabled by ever-more pervasive Internet connectivity, an increasing variety of exploitable vulnerabilities in software, and a lack of diversity in the software running on Internet-attached hosts, Internet worms increasingly threaten the availability and integrity of Internet-based services.

Toward defending against Internet worms (and other attacks), the research community has proposed and built intrusion detection systems (IDSes) [20, 21]. A network administrator deploys an IDS at the gateway between his edge network and the Internet, or on an individual end host. The IDS searches inbound traffic for known patterns, or *signatures*, that correspond to malicious traffic. When such malicious traffic is found, the IDS may raise an alarm; block future traffic from the offending source address; or even block the remainder of the offending flow's traffic. To date, to detect and/or block Internet worm flows, IDSes use signatures that match bytes from a worm's *payload*, using matching techniques including string matching at arbitrary payload offsets [20, 21]; string matching at fixed payload offsets [21]; and even matching of regular expressions within a flow's payload [20].

It is natural to ask where the signature databases for IDSes come from. To date, signatures have been generated manually by security experts who study network traces after a new worm has been released, typically hours or days after the fact. Motivated by the slow pace of manual signature generation, researchers have recently given attention to *automating* the generation of signatures used by IDSes to match worm traffic. Systems such as Honeycomb [14], Autograph [13], and EarlyBird [22] monitor network traffic to identify novel Internet worms, and produce signatures for them using *pattern-based* analysis,[1] *i.e.*, by extracting common byte patterns across different suspicious flows.

These systems all generate signatures consisting of a *single, contiguous substring* of a worm's payload, of *sufficient length* to match only the worm, and not innocuous traffic. The shorter the byte string, the greater the probability it will appear in some flow's payload, regardless of whether the flow is a worm or innocuous. Thus, these signature generation systems all make the same underlying assumptions: that there exists a single payload substring that will remain *invariant* across worm connections, and will be sufficiently unique to the worm that it can be used as a signature without causing false positives.

Regrettably, the above payload invariance assumptions are naïve, and give rise to a critical weakness in these previ-

[1] TaintCheck recently proposed a new approach, *semantic-based* automatic signature generation [18]. We discuss this further in Section 8.

ously proposed signature generation systems. A worm author may craft a worm that substantially changes its payload on every successive connection, and thus evades matching by any single substring signature that does not also occur in innocuous traffic. *Polymorphism* techniques[2], through which a program may encode and re-encode itself into successive, different byte strings, enable production of changing worm payloads. It is pure serendipity that worm authors thus far have not chosen to render worms polymorphic; virus authors do so routinely [17, 24]. The effort required to do so is trivial, given that libraries to render code polymorphic are readily available [3, 10].

It would seem that given the imminent threat of polymorphic worms, automated signature generation, and indeed, even filtering of worms using human-generated signatures, are doomed to fail as worm quarantine strategies. In this paper, we argue the contrary: that it is possible to generate signatures automatically that match the many variants of polymorphic worms, and that offer low false positives and low false negatives. This argument is based on a key insight regarding the fundamental nature of polymorphic worms as compared with that of polymorphic viruses. Polymorphic viruses are executables stored locally on a host, invoked by a user or application. As such, their content may be entirely arbitrary, so long as when executed, they perform the operations desired by the author of the virus. That is, a polymorphic generator has free reign to obfuscate all bytes of a virus. In sharp contrast, to execute on a vulnerable host, a worm must exploit one or more specific server software vulnerabilities.

In practice, we find that exploits contain *invariant* bytes that are crucial to successfully exploiting the vulnerable server. 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 server's execution. Individually, each of these invariant byte strings may cause false positives. Thus, in our work, we explore automatic generation of signature types that incorporate *multiple disjoint byte strings*, that used together, yield low false positive rates during traffic filtering. These signature types include conjunctions of byte strings, token subsequences (substrings that must appear in a specified order, a special case of regular expression signatures, matched by Bro and Snort), and Bayes-scored substrings.

Our contributions in this work are as follows:

Problem definition: We define the signature generation problem for polymorphic worms.

Signature generation algorithms: We present Polygraph, a suite of novel algorithms for automatic generation of signatures that match polymorphic worms.

Evaluation on real polymorphic worms: We use several real vulnerabilities to create polymorphic worms; run our signature generation algorithms on workloads consisting of samples of these worms; evaluate the quality (as measured in false positives and false negatives) of the signatures produced by these algorithms; and evaluate the computational cost of these signature generation algorithms.

We proceed in the remainder of the paper as follows. In Section 2, we first provide evidence of the existence of invariant payload bytes that cannot be rendered polymorphic using examples from real exploits, to motivate several classes of signature tailored to match disjoint invariant byte strings. We continue in Section 3 by setting the context in which Polygraph will be used, and stating our design goals for Polygraph. Next, in Section 4, we describe Polygraph's signature generation algorithms, before evaluating them in Section 5. We discuss possible attacks against Polygraph in Section 6; discuss our results in Section 7; review related work in Section 8; and conclude in Section 9.

2. Polymorphic Worms: Characteristics and Signature Classes

To motivate Polygraph, we now consider the anatomy of polymorphic worms. We refer to a network flow containing a particular infection attempt as an *instance* or *sample* of a polymorphic worm. After briefly characterizing the types of content found in a polymorphic worm, we observe that samples of the same worm often share some *invariant* content due to the fact that they exploit the same vulnerability. We provide examples of real-world software vulnerabilities that support this observation. Next, we demonstrate that a single, contiguous byte string signature[3] cannot always match a polymorphic worm robustly. Motivated by the insufficiency of single substring signatures and the inherent structure in many exploits, we identify a family of signature types more expressive than single substrings that better match an exploit's structure. While these signature types are more complex than single substring signatures, and thus computationally costlier to generate and match, they hold promise for robust matching of polymorphic worms.

2.1. Exploits and Polymorphism

Within a worm sample, we identify three classes of bytes. *Invariant bytes* are those fixed in value, which if changed, cause an exploit no longer to function. Such bytes

[2] Throughout this paper, we refer to both polymorphism and metamorphism as polymorphism, in the interest of brevity.

[3] For brevity, we hereafter refer to such signatures as *single substring* signatures.

are useful as portions of signatures. *Wildcard bytes* are those which may take on any value without affecting the correct functioning of a worm—neither its exploit nor its code. Finally, *code bytes* are the polymorphic code executed by a worm, that are the output of a polymorphic code engine. Typically, the main worm code will be encrypted under a different key in each worm sample. Execution starts at a small decryption routine, which is obfuscated differently in each worm sample. The degree of variation in code bytes from worm sample to worm sample depends on the quality of the polymorphic obfuscator used—a poor polymorphic obfuscater may leave long regions of bytes unchanged between the code instances it outputs, whereas a more aggressive one may leave nearly no multi-byte regions in common across its outputs. In this work, we do not depend on weaknesses of current code obfuscators to be able to generate quality signatures. Instead, we render worms to be *perfectly polymorphic*, by filling in code bytes with values chosen uniformly at random. We will also show that the current generation of polymorphic obfuscators actually do produce invariant byte sequences in their output, which means that we should be able to generate even higher quality signatures for worms that use these real-world code obfuscators.

2.2. Invariant Content in Polymorphic Exploits

If a vulnerability requires that a successful exploit contain invariant content, that content holds promise for use in signatures that can match all variants of a polymorphic worm. But to what extent do real vulnerabilities have this property? We surveyed over fifteen known software vulnerabilities, spanning a diverse set of operating systems and applications, and found that *nearly all* require invariant content in any exploit that can succeed. We stress that we do not claim all vulnerabilities share this property—only that a significant fraction do. We now describe the two chief sources of invariant content we unearthed: exploit framing and exploit payload.

Invariant Exploit Framing A software vulnerability exists at some particular code site, along a code path executed upon receiving a request from the network. In many cases, the code path to a vulnerability contains branches whose outcome depends on the content of the received request; these branches typically correspond to parsing of the request, in accordance with a specific protocol. Thus, an exploit typically includes *invariant framing* (*e.g.*, reserved keywords or well known binary constants that are part of a wire protocol) essential to exploiting a vulnerability successfully.

Invariant Overwrite Values Exploits typically alter the control flow of the victim program by overwriting a jump target in memory with a value provided in the exploit, either to force a jump to injected code in the payload, or to force a jump to some specific point in library code. Such exploits typically must include an address from some small set of narrow ranges in the request. In attacks that redirect execution to injected code, the overwritten address must point at or near the beginning of the injected code, meaning that the high-order bytes of the overwritten address are typically invariant. A previous study of exploits contains a similar observation [19]. Attacks that redirect execution to a library also typically select from a small set of candidate jump targets. For example, CodeRed causes the server to jump to an address in a common Windows DLL that contains the instruction `call ebx`. For this technique to be stable, the address used for this purpose must work for a range of Windows versions. According to the Metasploit op-code database, there are only six addresses that would work across Windows 2000 service packs zero and one [4].

2.3. Examples: Invariant Content in Polymorphic Worms

We manually identified the invariant content for exploits of a range of vulnerabilities by analyzing server source code (when available), and by studying how current exploits for the vulnerabilities work. We now present six of the vulnerabilities and exploits that we studied to illustrate the existence of invariant content in polymorphic worms, even with an ideal polymorphic engine. We also present our analysis of the output of one of the polymorphic generators, to show how close the current generators are to the ideal.

Apache multiple-host-header vulnerability First, we consider the hypothetical payload of a polymorphic worm structured like the payload of the Apache-Knacker exploit [9], shown in Figure 1. This exploit consists of a `GET` request containing multiple `Host` headers. The server concatenates the two `Host` fields into one buffer, leading to an overflow. This exploit contains several invariant protocol framing strings: “GET”, “HTTP/1.1”, and “Host:” twice. The second `Host` field also contains an invariant value used to overwrite the return address.

BIND TSIG vulnerability Next, we consider the Lion worm [5]. We constructed a polymorphic version of the Lion worm, shown in Figure 2. The Lion worm payload is a DNS request, and begins with the usual DNS protocol header and record counts, all of which may be varied considerably across payloads, and are thus wildcard bytes; only a single bit in the header must be held invariant for the exploit to function—the bit indicating that the packet is a request, rather than a response. Next come two question entries. The second contains an invariant value used to overwrite a return address (also encoded in a QNAME). Finally, to take the vulnerable code path in the server, the exploit payload must include an Additional record of type TSIG; this requirement results in three contiguous invariant

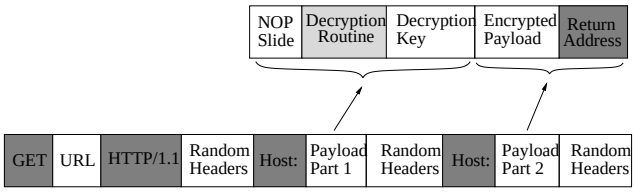


Figure 1. Polymorphed Apache-Knacker exploit. Unshaded content represents wildcard bytes; lightly shaded content represents code bytes; heavily shaded content represents invariant bytes.

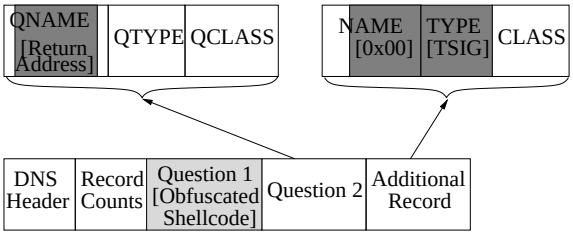


Figure 2. BIND TSIG vulnerability, as exploited by the Lion worm. Shading as for Apache vulnerability.

bytes near the end of the payload.

Slapper The Slapper worm [1] exploits a heap buffer overrun vulnerability in Apache's mod_ssh module. Note that the attack takes place during the initial handshake, meaning that it is not encrypted. It is a two-part attack; It first uses the overrun to overwrite a variable containing the session-id length, causing the server to leak pointer values. This part must contain the normal protocol framing of a `client-hello` message, as well as the value used to overwrite the variable (0x70).

In the second part of the attack, another session is opened, and the same buffer is overrun. This time, the leaked data is patched in, allowing the exploit to perform a longer buffer overrun while still not causing the server to crash. The heap metadata is overwritten in such a way as to later cause the GOT entry of `free` to be overwritten with a pointer to the attacker's code, placed previously on the heap. Thus, there is an invariant overwrite value that points to the attacker's code, and another that points to the GOT entry for `free`. An aggressively polymorphic worm may try to target other GOT entries or function pointers as well. However, there will still only be a relatively small number of values that will work.

SQLSlammer The SQLSlammer [2] exploit must begin with the invariant framing byte 0x04 in order to trigger the vulnerable code path. It uses a buffer overrun to overwrite a return address with a pointer to a `call esp` instruction contained in a common Windows DLL. There are only a small number of such values that work across multiple windows versions.

CodeRed The CodeRed [6] exploit takes advantage of a buffer overflow when converting ASCII to Unicode. The exploit must be a `GET` request for a `.ida` file. The value used to overwrite the return address must appear later in the URL. CodeRed overwrites the return address to point to `call esp`. There are only a small number of such pointers that will work across multiple Windows versions. Hence, the exploit must contain the invariant protocol framing string "GET", followed by ".ida?", followed by a pointer to `call esp`.

AdmWorm The AdmWorm [7] exploits BIND via a buffer overrun. Unlike the other exploits described here, there are *no* invariant protocol framing bytes in this exploit. However, there is still an invariant value used to overwrite a return address.

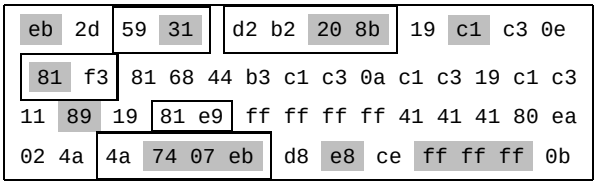


Figure 3. Output by Clet polymorphic engine includes invariant substrings. Boxed bytes are found in at least 20% of Clet's outputs; shaded bytes are found in all of Clet's outputs.

Clet polymorphic engine Figure 3 shows a sample output by the Clet polymorphic code engine [10].[4] The output consists of encrypted code, which is completely different each time, and a decryption routine that is obfuscated differently each time. In order to determine how effective the Clet obfuscation is, we generated 100 Clet outputs for the same input code, and counted substrings of all lengths in common among the decryption routines in these 100 outputs. Strings that were present in *all* 100 outputs appear with shaded backgrounds; those that were present in at least 20 outputs, but fewer than all 100, appear boxed. Clearly, Clet produces substrings that are entirely invariant across payloads, and other substrings that occur in a substantial fraction of payloads. However, upon examining the Clet source

[4] We also evaluated the ADMmutate [3] polymorphic engine. We present Clet as the more pessimal case, as it produced less invariant content than the ADMmutate engine.

code, it seems likely that the obfuscation engine could be improved significantly, reducing the number of substrings in common between Clet outputs.

2.4. Substring Signatures Insufficient

As described previously, the pattern-based signature generation systems proposed to date [14, 13, 22] generate single substring signatures, found either in reassembled flow payloads, or individual packet payloads. These systems thus make two assumptions about worm traffic:

- A single invariant substring exists across payload instances for the same worm; that is, the substring is *sensitive*, in that it will match all worm instances.[5]
- The invariant substring is sufficiently long to be *specific*; that is, the substring does not occur in any non-worm payloads destined for the same IP protocol and port.

Can a sensitive and specific single substring signature be found in the example payloads in the Apache and DNS exploits described in Section 2.3? Consider the Apache exploit. The unshaded bytes are wildcards, and cannot be relied upon to provide invariant content; note that even the NOP slide can contain significantly varying bytes across payloads, as many instruction sequences effectively may serve as NOPs. If we assume a strong code obfuscator, we cannot rely on there being an invariant substring longer than two bytes long in the obfuscated decryption routine, shown with light shading. The only invariant bytes are the heavily shaded ones, which are pieces of HTTP protocol framing, and a return address (or perhaps a two-byte prefix of the return address, if the worm is free to position its code anywhere within a 64K memory region). Clearly, the HTTP protocol framing substrings individually will not be specific, as they can occur in both innocuous and worm HTTP flows. By itself, even the two-to-four-byte return address present in the payload is not sufficiently specific to avoid false positives; consider that a single binary substring of that length may trivially occur in an HTTP upload request. As we show in our evaluation in Section 5, we have experimentally verified exactly this phenomenon; we have found return address bytes from real worm payloads in innocuous flows in HTTP request traces taken from the DMZ of Intel Research Pittsburgh.

The Lion worm presents a similar story: the heavily shaded invariant bytes, the high-order bytes of the return address, and TSIG identifier, two and three bytes long, respectively, are too short to be specific to the Lion worm. As we show in our evaluation in Section 5, we found false positives when searching for those substrings in DNS traffic traces from a busy DNS server that is a nameserver for top-level country code domains.

We conclude that single substring signatures cannot match polymorphic worms with low false positives and low false negatives.

[5] It is possible that a worm's content varies only very slightly across instances, and that at least one of a small, constant-cardinality set of substring signatures matches all worm instances. We view this case as qualitatively the same as that where worm content is invariant, and focus our attention herein on worms whose content varies to a much greater extent, such that a small set of substring signatures does not suffice to match all variants.

2.5. Signature Classes for Polymorphic Worms

Motivated by the insufficiency of single substring signatures for matching polymorphic worms robustly, we now propose other signature classes that hold promise for matching the particular invariant exploit framing and payload structures described in this section. All these signatures are built from substrings, or *tokens*. The signature classes we investigate in detail in Section 4 include:

Conjunction signatures A signature that consists of a set of tokens, and matches a payload if *all* tokens in the set are found in it, in any order. This signature type can match the multiple invariant tokens present in a polymorphic worm's payload, and matching multiple tokens is more specific than matching one of those tokens alone.

Token-subsequence signatures A signature that consists of an *ordered* set of tokens. A flow matches a token-subsequence signature if and only if the flow contains the sequence of tokens in the signature with the same ordering. Signatures of this type can easily be expressed as regular expressions, allowing them to be used in current IDSes [20, 21]. For the same set of tokens, a token subsequence signature will be more specific than a conjunction signature, as the former makes an ordering constraint, while the latter makes none. Framing often exhibits ordering; *e.g.* the TSIG record in the Lion worm, which must come *last* in the payload for the exploit to succeed, and the return address, which must therefore come before it.

Bayes signatures A signature that consists of a set of tokens, each of which is associated with a *score*, and an overall *threshold*. In contrast with the exact matching offered by conjunction signatures and token-subsequence signatures, Bayes signatures provide probabilistic matching—given a flow, we compute the probability that the flow is a worm using the scores of the tokens present in the flow. If the resulting probability is over the threshold, we classify the flow to be a worm. Construction and matching of Bayes signatures is less *rigid* than for conjunction or token-subsequence signatures. This provides several advantages. It allows Bayes signatures to be learned from suspicious flow pools that contain samples from unrelated worms, and even innocuous network requests. (We show that the other signature

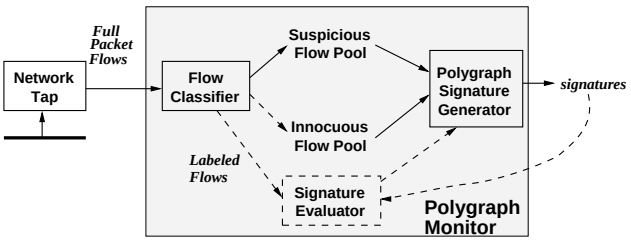


Figure 4. Architecture of a Polygraph monitor.

generation algorithms can be adapted to deal with these situations effectively, but at a higher computational cost). It also helps to prevent false negatives in cases where a token is observed in all samples in the suspicious flow pool, but does not actually appear in every sample of the worm. This issue is further discussed in Section 6.

3. Problem Definition and Design Goals

We now consider the context in which we envision Polygraph will be used, both to scope the problem we consider in this paper, and to reveal challenges inherent in the problem. Having defined the problem, we then offer design goals for Polygraph.

3.1. Context and Architecture

Figure 4 depicts a typical deployment of a Polygraph monitor, shown as the shaded region, which incorporates the Polygraph signature generator. In this paper, we concern ourselves with the detailed design of algorithms for the Polygraph signature generator. We now give a brief overview of the remaining pieces of a Polygraph monitor, to provide context for understanding how the Polygraph signature generator fits into an end-to-end system.

We envision that a Polygraph monitor observes all network traffic, either at a monitoring point such as between an edge network and the Internet, or at an end host. In this work, we consider only a single monitor instance at a single site.[6] Monitored network traffic passes through a flow classifier, which reassembles flows into contiguous byte flows, and classifies reassembled flows destined for the same IP protocol number and port into a *suspicious flow pool* and an *innocuous flow pool*. Flow reassembly (including traffic normalization) at a monitor has been well studied in the IDS research community [20]; we defer a discussion of the liabilities of conducting flow reassembly to Section 6.

[6]While we believe that extending Polygraph to work distributedly holds promise for reasons explored in previous signature generation systems [13], we leave such extensions to future work.

There is a rich literature on methods for identifying anomalous or suspicious traffic. Previous signature generation systems have used inbound honeypot traffic [14] or port scan activity [13] to identify suspicious flows. Far more accurate techniques are also available, such as monitoring the execution of a server to detect exploits at run time, and mapping exploit occurrences to the network payloads that caused them, as is done in run-time-detection-based methods [18]. These current techniques are not suitable for blocking individual infection attempts, either because they are too inaccurate or too slow, but they are suitable for use in a flow classifier for Polygraph. The design of flow classifiers is outside the scope of this paper, but we assume as we design and evaluate algorithms for the Polygraph signature generator that a flow classifier will be imperfect—that it may misclassify innocuous flows as suspicious, and vice-versa. Such misclassified flows increase the difficulty of avoiding the generation of signatures that cause false positives; we refer to such innocuous flows in the suspicious flow pool as *noise*.

Another challenge in generating high-quality signatures is that we presume the flow classifier does not distinguish between different worms (though some classifiers may be able to help do so [18]); it simply recognizes all worms as worms, and partitions traffic by destination port. Thus, the suspicious flow pool for a particular destination port may contain a mixture of different worms—that is, worms that are *not* polymorphic variants based on the same exploit. For example, a single edge network may include hosts running different HTTP server implementations, each with different vulnerabilities. In such a case, different worm payloads in the suspicious flow pool for port 80 observed at the DMZ may contain different exploits. As we describe in our design goals, the signature generation algorithm should produce high-quality signatures, even when the suspicious flow pool contains a mixture of different worms.

In the simplest possible setting, signature generation works as a single pass: the Polygraph signature generator takes a suspicious flow pool and an innocuous flow pool as input, and produces a set of signatures as output, chosen to match the worms in the input suspicious flow pool, and to minimize false positives, based on the innocuous flow pool. However, we believe that incorporating feedback, whereby the Polygraph signature generator is provided information concerning the false positives and false negatives caused by signatures it has previously generated, can significantly improve signature quality and allow Polygraph to adapt to attacks that change over time.

3.2. Problem Definition for Polygraph Signature Generator

In the remainder of this paper, we focus on the signature generation algorithms in the Polygraph signature generator.

We now formally define the signature generation problem, and introduce terminology and notation used in subsequent exposition.

The signature generation algorithm is given a training pool containing a *suspicious flow pool* where each flow is *labeled* as a worm flow, and an *innocuous flow pool* where each flow is labeled as a non-worm. Note that these labels are not necessarily accurate. In particular, the suspicious pool may contain some innocuous flows. We refer to innocuous flows in the suspicious flow pool as *noise*.

The signature generation algorithm then produces a set of signatures. We state that the signature set causes a false positive for a flow if it *is not* a worm, but one or more signatures in the signature set matches it. If a network flow *is* a worm, but no signature in the signature set classifies the payload as a worm, we state that the signature set causes a false negative for that network flow.

3.3. Design Goals

Polygraph[7] must meet several design goals to work effectively:

Signature quality. Our end-to-end goal in Polygraph, as has been the case in prior worm signature generation systems, is to generate signatures that offer low false positives for innocuous traffic and low false negatives for worm instances, including polymorphic worm instances.

Efficient signature generation. The signature types proposed in Section 2.5 are more complex than the single substring signatures generated automatically by today's signature generation systems. To the extent possible, we seek to minimize the computational cost of signature generation in the size of the suspicious flow pool. Thus, we seek efficient algorithms for signature generation.

Efficient signature matching. Each signature type also incurs a different computational cost during matching against network traffic. We characterize these matching costs for each signature type, to argue for the tractability of filtering using these more complex signatures.

Generation of small signature sets. Some constraint must be made on the *number* of signatures Polygraph generates to match a suspicious flow pool. In the extreme case, Polygraph might generate one signature for each polymorphic payload. Clearly, such behavior does not qualify as generating a signature that matches a polymorphic worm. We seek to minimize the number of signatures Polygraph generates for a suspicious flow pool, without sacrificing signature quality (causing false positives). Such sets of signatures cost less bandwidth to disseminate, and cost less to match at traffic filtering time.

[7]For the remainder of the paper, we refer to the Polygraph signature generator as Polygraph, in the interest of brevity.

Robustness against noise and multiple worms. A successful signature generator must generate high-quality signatures on workloads that contain noise or a mixture of different worms on the same destination port. If the system cannot find a fully general signature that matches all worms in the pool and does not cause false positives, it should instead generate multiple signatures, each of which matches some subset of flows in the suspicious flow pool (most likely a subset that employ the same exploit), and such that the set of signatures together exhibits low false positives and low false negatives.

Robustness against evasion and subversion. An adversary who knows the design of Polygraph may attempt to evade or subvert the system. Several well known attacks against IDS systems may be mounted against Polygraph, but there are novel attacks specific to Polygraph as well. An adversary may, for example, evolve a worm's payload over time, in an effort to cause signatures previously generated by Polygraph to cease matching the worm. We consider several evasion and subversion strategies an adversary might adopt in Section 6, and describe defenses against them.

4. Signature Generation Algorithms

In this section, we will describe our algorithms for automatically generating signatures of different classes including conjunction signatures, token subsequence signatures, and Bayes signatures. For ease of explanation, we first consider the problem of generating one signature that matches every sample (or most of the samples) in the suspicious flow pool. However, when the suspicious flow pool has noise or contains a mix of different worms (or a worm with different attack vectors), generating one signature that matches every flow is not always possible or will result in low-quality signatures. In Section 4.3, we will show how these algorithms can be adapted to handle the cases when there is noise and when there are multiple worms in the suspicious pool, by generating a set of signatures where each signature in the set only matches part of the suspicious pool and the set of signatures together match the samples in the suspicious pool.

Many of the algorithms described in this section are based on algorithms found in [11].

4.1. Preprocessing: Token Extraction

We define a *token* to be a contiguous byte sequence. Each signature in the signature classes that we consider is made up of one or more such tokens. Here we discuss algorithms for extracting and analyzing tokens, which will be used in our algorithms for creating signatures.

As a preprocessing step before signature generation, we extract all of the distinct substrings of a minimum length

α that occur in at least *K* out of the total *n* samples in the suspicious pool. By distinct, we mean that we do not want to use a token that is a substring of another token, unless it occurs in at least *K* out of *n* samples *not* as a substring of that token. For example, suppose one of the substrings occurring in at least *K* out of the *n* samples is "HTTP". "TTP" is not a *distinct* substring unless it occurs in at least *K* of the *n* samples, *not* as a substring of "HTTP".

There is a well-known algorithm to find the *longest* substring that occurs in at least *K* of *n* samples [12], in time linear in the total length of the samples. That algorithm can be trivially modified to return a set of substrings that includes all of the distinct substrings that occur in at least *K* out of *n* samples, but also includes some of the non-distinct substrings, in the same time bound. We can then *prune* out the non-distinct substrings and finally output the set of tokens for use in signature generation.

Token extraction can be viewed as a first step toward eliminating the irrelevant parts of suspicious flows. After the token extraction, we can simply represent each suspicious flow as a sequence of tokens, and remove the rest of the payload.

4.2. Generating Single Signatures

We next describe our algorithms that automatically generate a single signature that matches all (or most of) the suspicious flow pool. Note that this approach of forcing all (or most of) the suspicious flow pool to be matched using a single signature is not resilient against noise or when the suspicious flow pool contains a mixture of different worms. We present our full algorithms to address these issues in Section 4.3.

4.2.1. Generating Conjunction Signatures

A conjunction signature consists of an unordered set of tokens, where a sample matches the signature if and only if it contains every token in the signature. To generate one conjunction signature matching every sample in the pool, we can simply use the token extraction algorithm described above to find all the distinct tokens that appear in every sample of the suspicious pool. The signature is then this set of tokens. The running time of the algorithm is linear in the total byte length of the suspicious pool.

4.2.2. Generating Token-Subsequence Signatures

A token-subsequence signature is an ordered list of tokens. A sample matches a token-subsequence signature if and only if the subsequence of tokens is in the sample. To generate a token-subsequence signature, we want to find an ordered sequence of tokens that is present in every sample in the suspicious pool. We begin by showing how to find the signature from two samples, and then show how we can use that algorithm to find a token-subsequence signature for any number of samples.

A subsequence of two strings is a sequence of bytes that occur in the same order in both strings, though not necessarily consecutively. For example, in the strings "xxonexxxtwox" and "oneyyyyytwoyy", the longest common subsequence is "onetwo". The problem of finding the longest common subsequence of two strings can be framed as a *string alignment* problem. That is, given two strings, we wish to align them in such a way as to maximize the number of characters aligned with a matching character. The alignment that gives the longest subsequence in the previous examples is:

```
x x [o n e] x x x - - [t w o] x -
- - [o n e] y y y y y [t w o] y y
```

This alignment can be described by the regular expression ".*one.*two.*".

Note that the *longest* subsequence does not maximize *consecutive* matches, only the total number of matches. For example, consider the strings "oxnxexzxtwox" and "ytwoyoynyeyz". The alignment corresponding to the longest subsequence is:

```
- - - - - [o] x [n] x [e] x [z] x t w o x
y t w o y [o] y [n] y [e] y [z] - - - - -
```

This results in the signature ".*o.*n.*e.*z.*". However, in this case we would prefer to generate the signature ".*two.*", which corresponds to the alignment:

```
o x n x e x z x [t w o] x - - - - - - - -
- - - - - - - - y [t w o] y o y n y e y z
```

Although the second alignment produces a shorter subsequence, the fact that all the bytes are contiguous produces a much better signature. (We can use the technique in Appendix A to show that the first signature has a 54.8% chance of matching a random 1000-byte string, while the second signature has only a .0000595% chance). Thus, we need to use a string alignment algorithm that prefers subsequences with contiguous substrings.

We use an adaptation of the Smith-Waterman [23] algorithm to find such an alignment. An alignment is assigned a score by adding 1 for each character that is aligned with a matching character, and subtracting a gap penalty W_g for each maximal sequence of spaces and/or non-matching characters.[8] That is, there is a gap for every ".*" in the resulting signature. However, we do not count the first and the last ".*", which are always present. In our experiments, we set W_g to 0.8 (We used the technique in Appendix A to help choose this value, based on minimizing the chance of the resulting signature matching unrelated strings). Using these parameters, the score for the alignment producing the signature ".*o.*n.*e.*z.*" has a value of $4 - 3 * .8 = 1.6$,

[8] This differs from the common definition of a gap, which is a maximal sequence of spaces.

while the score for the alignment producing the signature ".*two.*" has a value of $3-0*.8=3$. Hence, the latter signature would be preferred. The Smith-Waterman algorithm finds the highest-scoring alignment between two strings in $O(nm)$ time and space, where n and m are the lengths of the strings.[9]

We generate a signature that matches every sample in the suspicious pool by finding a subsequence of tokens that is present in each sample. We find this by iteratively applying the string-alignment algorithm just described. After each step, we replace any gaps in the output with a special gap character γ, and find the best alignment between it and the next sample. Note that this algorithm is *greedy*, and could reach a local minimum. To help reduce this risk, we first use the token extraction algorithm to find the tokens present in every sample, and then convert each sample to a sequence of tokens separated by γ. This helps prevent an early alignment from aligning byte sequences that are not present in other samples. It also has the added benefit of reducing the lengths of the strings, and hence the running time of the Smith-Waterman pairwise comparisons.

If the suspicious pool consists of s samples, each n bytes long, the running time is $O(n)$ to perform the token extraction, plus $O(sn^2)$ to perform the alignments.

4.2.3. Generating Bayes Signatures

The conjunction and token-subsequence classes of signatures assume that the distinction between worms and innocuous flows involves an exact pattern of a set of tokens. However, the distinction between worms and innocuous flows may instead be a difference in the probability distributions over sets of tokens that may be present. Thus, given two different distributions over sets of tokens (*e.g.*, for worms and innocuous flows), we could classify a flow by the distribution from which its token set is more likely to have been generated. This type of signature allows for probabilistic matching and classification, rather than for exact matches, and may be more resilient to noise and changes in the traffic.

We study the naïve Bayes classifier as a first step toward exploring this class of signatures. This model is characterized by the following independence assumption: the probability of a token being present in a string, when the string is known to be a worm or an innocuous string, is independent of the presence of other tokens in the string. This assumption often holds approximately in many practical scenarios, and is simple enough to allow us to focus on the important question, *i.e.*, how such a probabilistic matching scheme compares to the exact matching schemes. In addition, a naïve Bayes classifier needs far fewer examples to approach its asymptotic error, in comparison to many other models; thus, it will yield very good results when it is used with an extremely large number of dimensions (*i.e.* tokens, in our case) and a moderately sized suspicious pool. In future work, we can easily relax this independence assumption and extend the naïve Bayes model to other more complex Bayesian models to allow more complex dependencies in the presence of sets of tokens.

As in the conjunction and subsequence signature generation, the first step in generating a Bayes signature is to choose the set of tokens to use as features, as described in Section 4.1. Assume that we have a set of n tokens, $\{T_i\}_{1\le i\le n}$, from the preprocessing step. Thus, a flow x could be denoted with a vector $(x_1,\ldots,x_n)$ in $\{0,1\}^n$, where the ith bit x_i is set to 1 if and only if the ith token T_i is present somewhere in the string.

We then calculate the empirical probability of a token occurring in a sample given the classification of the sample (a worm or not a worm), *i.e.*, for each token T_i, we compute the probability that the token T_i is present in a worm flow, denoted as t_i, and the probability that the token T_i is present in an innocuous flow, denoted as s_i. We calculate t_i simply as the fraction of samples in the suspicious flow pool that the token T_i occurs in. We estimate s_i, the probability of a token occurring in innocuous traffic, by measuring the fraction of samples it appears in the innocuous pool, and by calculating it using the technique described in Appendix A. We use whichever value is greater, in an effort to minimize the risk of false positives.

Given a sample x, let $\mathcal{L}(x)$ denote the true label of x, *i.e.*, $\mathcal{L}(x)=\text{worm}$ denotes x is a worm, and $\mathcal{L}(x)=\sim\text{worm}$ denotes x is not a worm. Thus, to classify a sample $x=(x_1,\ldots,x_n)$, we wish to compute $\Pr[\mathcal{L}(x)=\text{worm}|x]$ and $\Pr[\mathcal{L}(x)=\sim\text{worm}|x]$.

To calculate $\Pr[\mathcal{L}(x)=\text{worm}|x]$, we use Bayes law.

$$\Pr[\mathcal{L}(x)=\text{worm}|x]$$

$$=\frac{\Pr[x|\mathcal{L}(x)=\text{worm}]}{\Pr[x]}\Pr[\mathcal{L}(x)=\text{worm}]$$

From the independence assumption of the naïve Bayes model, we can compute this as follows:

$$=\frac{\Pr[\mathcal{L}(x)=\text{worm}]}{\Pr[x]}\prod_{1\le i\le n}\Pr[x_i=1|\mathcal{L}(x)=\text{worm}].$$

We only need to estimate the quantity $\frac{\Pr[\mathcal{L}(x)=\text{worm}|x]}{\Pr[\mathcal{L}(x)=\sim\text{worm}|x]}$ (*i.e.*, if this is greater than 1, then the x is more likely to have been generated by a worm, and vice-versa).

$$\frac{\Pr[\mathcal{L}(x)=\text{worm}|x]}{\Pr[\mathcal{L}(x)=\sim\text{worm}|x]}$$

$$=\frac{\Pr[\mathcal{L}(x)=\text{worm}]\cdot\prod_{1\le i\le n}\Pr[x_i=1|\mathcal{L}(x)=\text{worm}]}{\Pr[\mathcal{L}(x)=\sim\text{worm}]\cdot\prod_{1\le i\le n}\Pr[x_i=1|\mathcal{L}(x)=\sim\text{worm}]}$$

[9] Hirchberg's algorithm can reduce the space bound to $O(m)$, where m is the length of the longer string.

To calculate the result, we need to find a value to use for $\Pr[\mathcal{L}(x) = \text{worm}]$, *i.e.*, the probability that any particular flow is a worm. This value is difficult to determine, and changes over time. We simply set $\Pr[\mathcal{L}(x) = \text{worm}] = \Pr[\mathcal{L}(x) = \sim \text{worm}] = .5$. Since false positives are often considered more harmful than false negatives, we set a threshold so that the classifier reports positive only if it is sufficiently far away from the decision boundary. Given a desired maximum false positive rate, the value of the threshold to use is automatically set by running the classifier on the innocuous traffic pool and the suspicious traffic pool, and selecting a threshold that minimizes the "negative" classifications in the suspicious traffic pool while achieving no more than the maximum false positive rate in the innocuous traffic pool.

In practice, we transform the formula above such that each token is assigned a score based on the log of its term in the formula. To classify a sample, the scores of the tokens it contains are added together. If a token is present in a sample multiple times, it is counted only once. If the total score is greater than the threshold, the sample is classified as a worm. This transformation allows the signatures to be more human-understandable than if we were to use the probability calculations directly.

4.3. Generating multiple signatures

In a practical deployment, the suspicious flow pool could contain more than one type of worm, and could contain innocuous flows (as a result of false positives by the flow classifier). In these cases, we would like for Polygraph to still output a signature, or *set* of signatures, that matches the worms found in the suspicious pool, and does not match innocuous flows.

We show that the Bayes generation algorithm can be used unmodified even in the case of multiple worms or noise in the pool. However, for the token subsequence and conjunction algorithms, we must perform *clustering*. With clustering, the suspicious flow pool is divided into several clusters, where each cluster contains *similar* flows. The system then outputs a signature for each cluster, by using the algorithms previously described to generate a signature that matches every flow in a cluster.

Clearly, the quality of the clustering is important for generating good signatures. First, the clusters should not be too general. If we mix flows from different worms into the same cluster, or mix flows of worms and noise in the same cluster, the resulting signature may be too general and exhibit a high false positive rate. Second, the clusters should not be too specific. If flows of the same worm are separated into different clusters, the signatures for each cluster may be too specific to match other flows of the worm.

We choose to adapt a widely-used clustering method, *hierarchical clustering* [11], to our problem setting. Hierarchical clustering is relatively efficient, does not need to know the number of clusters beforehand, and can be adapted to match our semantics. We next describe the details of how we use hierarchical clustering in Polygraph.

Hierarchical Clustering. Each cluster consists of a set of flows, and a signature generated using that set. Given s flows, we begin with s clusters, each containing a single flow. At this point, the signature for each cluster is very specific. It matches exactly the one flow in that cluster.

The next step is to iteratively merge clusters. Whenever two clusters are merged, the signature generation algorithm being used is run again on the combined set of samples to produce a new, more sensitive signature for the new cluster.

We decide which two of the clusters to merge first by determining what the merged signature would be for each of the $O(s^2)$ pairs of clusters, and using the innocuous pool to estimate the false positive rate of that signature. The lower the false positive rate is, the more specific the signature is. The more specific the signature is, the more similar are the two clusters. Hence, we merge the two clusters that result in the signature with the lowest false positive rate. After each merge, we compute what the merged signature would be between the new cluster and each of the remaining $O(s)$ clusters. We always choose whichever pair of the current clusters results in the signature with the lowest false positive rate to merge next.

Merging stops when the signature resulting from merging any two clusters would result in an unacceptably high false positive rate, or when there is only one cluster remaining. The system then outputs the signature for each of the remaining clusters that has enough samples in it to be likely to be general enough. As we show in Section 5, a cluster should contain at least 3 samples to be general enough to match other samples of the worm. The cost of this algorithm is to compute $O(s^2)$ signatures.

Note that our method for generating signatures with clustering is a greedy approach for finding the best signatures. As is well-known in the literature, a greedy approach may reach local minimum instead of global minimum. For example, two flows from different worms may have some coincidental similarity, causing them to be merged into a single cluster during an early round of the algorithm, possibly preventing the ideal clustering (and set of signatures) from being found. However, due to the complexity of the problem, a greedy approach is worthwhile, since it offers reduced computational cost compared to an exhaustive search of possible clusterings.

5. Evaluation

In our experiments, we evaluate the performance of each Polygraph signature generation algorithm under several scenarios. We first consider the simple case where the suspi-

cious flow pool contains only flows of one worm. We next consider the case where the flow classifier is imperfect, resulting in innocuous requests present in the suspicious flow pool along with the flows from one worm. Last, we consider the most general case, in which the suspicious flow pool contains flows from multiple worms, and from innocuous requests.

5.1. Experimental Setup

We describe our experimental setup below. In all our experiments, we set the token-extraction threshold $k = 3$ (described in Section 4.1), the minimum token length $\alpha = 2$, and the minimum cluster size to be 3. We conduct 5 independent trials for each experiment, and report the 2nd worst value for each data point (*e.g.*, the 80th percentile). All experiments were run on desktop machines with 1.4 GHz Intel® Pentium® III processors, running Linux kernel 2.4.20.

Polymorphic workloads. We generate signatures for polymorphic versions of three real-world exploits. The first two exploits, the Apache-Knacker exploit (described in Section 2) and the ATPhttpd exploit[10] use the text-based HTTP protocol. The third exploit, the BIND-TSIG exploit, uses the binary-based DNS protocol.

In our experiments, we show that Polygraph generates high quality signatures for both HTTP exploits and the DNS exploit, even with an ideal polymorphic engine. In order to simulate an ideal polymorphic engine, we fill wildcard and code bytes for each exploit with values chosen uniformly at random. For the HTTP exploits, we also include randomly generated headers of random length, which do not affect the functioning of the exploit.

Network traces. We used several network traces as input for and to evaluate Polygraph signature generation. For our HTTP experiments, we used two traces containing both incoming and outgoing requests, taken from the perimeter of Intel Research Pittsburgh in October of 2004. We used a 5-day trace (45,111 flows) as our innocuous HTTP pool. We used a 10-day trace (125,301 flows), taken 10 days after the end of the first trace, as an evaluation trace. The evaluation trace was used to measure the false positive rate of generated signatures. In experiments with noisy suspicious pools, noise flows were drawn uniformly at random from the evaluation pool.

We also used a 24-hour DNS trace, taken from a DNS server that serves a major academic institution's domain, and several CCTLDs. We used the first 500,000 flows from this trace as our innocuous DNS pool, and the last 1,000,000 flows as our evaluation trace.

[10] In this ATPhttpd exploit, the attacker provides a long URL in a GET request, which is used to overwrite the return address on the server, transferring control to the attacker's code.

5.2. Experimental Results

We describe our experimental results below.

5.2.1. Single Polymorphic Worm

We first consider the case where the suspicious flow pool contains only flows from one worm. In these experiments, we want to determine what signatures Polygraph would find for each worm, how accurate these signatures are (*e.g.*, how many false positives and false negatives they cause), and how many worm samples are necessary to generate a quality signature. If there are too few worm samples in the suspicious flow pool, the resulting signatures will be too specific, because they will incorporate tokens that those samples have in common only by coincidence, but that do not appear in other samples of the worm. For each exploit, we run our signature generation algorithms using different suspicious pools, of size ranging from 2 to 100 worm samples.

Signature Quality. Tables 1 and 2 show our results for the Apache-Knacker and BIND-TSIG exploits. For sake of comparison, we also evaluate the signatures based on the longest common substring, and the most specific common substring (that is, the one that results in the fewest false positives) for each worm.[11] Token-subsequence signatures are shown in regular expression notation. The Bayes signatures are a list of tokens and their corresponding scores, and the threshold decision boundary, which indicates the score necessary for a flow to match the signature.

The conjunction and token-subsequence signatures generated by Polygraph exhibit significantly fewer false positives than ones consisting of only a single substring. For the Apache-Knacker exploit, the subsequence signature produces a lower false positive rate than the conjunction signature, which is expected since the ordering property makes it more specific. For both exploits, the Bayes signature is effectively equivalent to the best-substring signature. This is reasonable for the Apache-Knacker exploit, since all but one of the tokens occurs very frequently in innocuous traffic. For the BIND-TSIG exploit, the Bayes signature would be equivalent to the conjunction signature if the matching threshold were set slightly higher. We hypothesize that this would have happened if we had specified a lower maximum false positive rate (we used .001%). It also would have happened if the best substring occurred equally often in the input innocuous pool as in the evaluation trace.

Number of Worm Samples Needed. For each algorithm, the correct signature is generated 100% of the time for all experiments where the suspicious pool size is greater than 2, and 0% of the time where the suspicious pool size is only 2.

[11] We do not propose an algorithm to find such a substring automatically—we simply measure the result of using each substring and report the best one.

Class	False +	False −	Signature
Longest Substring	92.5%	0%	HTTP/1.1\r\n
Best Substring	.008%	0%	\xFF\xBF
Conjunction	.0024%	0%	'GET ', ' HTTP/1.1\r\n', ': ', '\r\nHost: ', '\r\n', ': ', '\r\nHost: ', '\xFF\xBF', '\r\n'
Token Subsequence	.0008%	0%	GET .* HTTP/1.1\r\n.*: .* \r\nHost: .* \r\n.*: .*\r\nHost: .*\xFF\xBF.*\r\n
Bayes	.008%	0%	'\r\n': 0.0000, ': ': 0.0000, '\r\nHost: ': 0.0022, 'GET ': 0.0035, ' HTTP/1.1\r\n': 0.1108, '\xFF\xBF': 3.1517. Threshold: 1.9934

Table 1. Apache-Knacker signatures. These signatures were successfully generated for innocuous pools containing at least 3 worm samples.

Class	False +	False −	Signature
Longest Substring	.3279%	0%	\x00\x00\xFA
Best Substring	.0023%	0%	\xFF\xBF
Conjunction	0%	0%	'\xFF\xBF', '\x00\x00\xFA'
Token Subsequence	0%	0%	\xFF\xBF.*\x00\x00\xFA
Bayes	.0023%	0%	'\x00\x00\xFA': 1.7574, '\xFF\xBF': 4.3295 Threshold: 4.2232

Table 2. BIND-TSIG signatures. These signatures were successfully generated for innocuous pools containing at least 3 worm samples.

The signatures generated using 2 samples are too specific, and cause 100% false negatives.

5.2.2. Single Polymorphic Worm Plus Noise

Next we show that Polygraph generates quality signatures even if the flow classifier misclassifies some flows, resulting in innocuous flows in the suspicious flow pool. In these experiments, we use hierarchical clustering with our conjunction and token subsequence algorithms. Ideally, one or more signatures will be generated that match future samples of the polymorphic worm, and no signatures will be generated from the innocuous traffic that will result in false positives. We also demonstrate that Bayes does not require hierarchical clustering, even when there are innocuous flows in the suspicious pool. In each of these experiments, we use 5 flows from a polymorphic worm, while varying the number of additional innocuous flows in the suspicious flow pool.

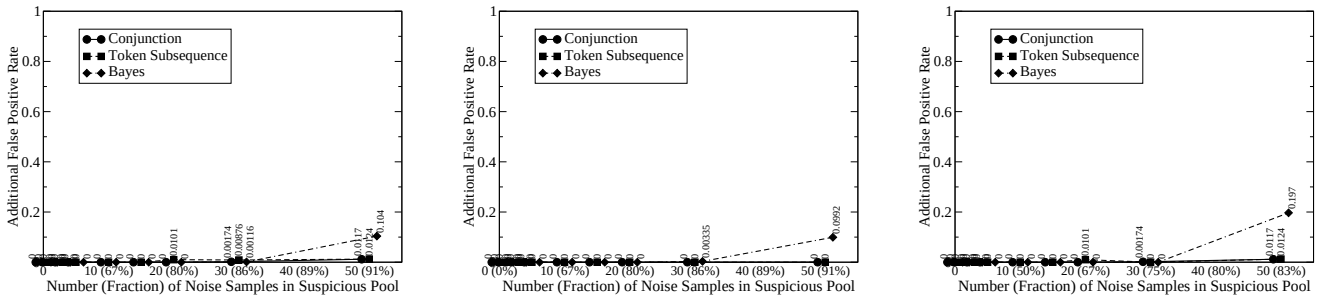
False Negatives For the conjunction and token-subsequence signatures, Polygraph generates a cluster containing the worm flows, and no other flows. The signatures for these clusters are the same signatures generated in the case with no noise, and produce 0% false negatives.

The Bayes signatures are not affected by noise until it grows beyond 80%, at which point the signatures cause 100% false negatives. This is because we are only using a token as a feature in the Bayes signature if it occurs in at least 20% of the suspicious flow pool.[12] This parameter can be adjusted to allow Bayes to generate signatures with higher noise ratios.

False Positives Figures 5(a) and 5(b) show the *additional* false positives (that is, not including those generated by the correct signatures) that result from the addition of noise. In the HTTP case, when there is sufficiently high noise, there are also clusters of innocuous flows that result in signatures. That is, the clustering algorithm interprets similar noise flows that are dissimilar from flows in the innocuous pool as other worms in the pool. We hypothesize that this occurs because our HTTP traces come from a relatively small site. That is, a more diverse innocuous pool would allow the algorithm to determine that the resulting signatures cause too many false positives and should not be output.

Again, once the ratio of noise grows beyond the 80% threshold, Bayes does not use any tokens from the actual worm flows as part of its signature. Instead, the signature consists only of tokens common to the innocuous noise

[12] We do this to minimize the number of tokens that are only coincidentally in common between flows being used as part of the signature.



(a) Apache-Knacker exploit

(b) BIND TSIG exploit

(c) Mixed HTTP exploits

Figure 5. False positives due to noise in suspicious pool.

flows, resulting in false positives.

5.2.3. Multiple Polymorphic Worms Plus Noise

Finally, we examine the fully general case: that in which there are flows from more than one polymorphic worm, and misclassified innocuous flows in the suspicious flow pool. We evaluate Polygraph on a suspicious flow pool containing 5 flows from the Apache-Knacker polymorphic worm; 5 from the ATPhttpd polymorphic worm, and a varying quantity of noise flows. Ideally, Polygraph should generate a signature that covers each polymorphic worm, and not generate any signatures that cover the innocuous flows.

False Negatives. Our results in this case are similar to those with one HTTP worm plus noise. We observe that Polygraph generates conjunction and token-subsequence signatures for each of the two polymorphic worms. Bayes generates a single signature that matches *both* worms. The signatures generated by each algorithm generate 0% false negatives, except for Bayes once the fraction of noise flows increases beyond 80%, at which point it has 100% false negatives.

False Positives. Figure 5(c) shows that the false positive behavior is very similar to when there is only one type of worm in the suspicious pool. Once again, the signatures generated by each algorithm have no false positives until there are a large number of noise samples in the suspicious pool.

5.2.4. Runtime Performance Overhead

Without clustering, all of our signature generation techniques generate a signature very quickly. For example, when training on 100 samples in our Apache-Knacker evaluation, the conjunction signature, the token subsequence signature, and the Bayes signature are each computed in under 10 seconds. The cost of signature generation grows with the square of the number of samples when using hierarchical clustering. However, we still find that the run-times are reasonable, even with our unoptimized implementation. When training on 25 samples, the conjunction and subsequence signatures with hierarchical clustering are generated in under ten minutes.

The performance of our signature generation algorithms can be improved with optimizations. Additionally, some of our algorithms can be parallelized (especially hierarchical clustering), allowing the signature generation time to be reduced significantly by using multiple processors.

6. Attack Analysis

In this section, we analyze potential attacks on Polygraph, and propose countermeasures. Note that some attacks are not unique to Polygraph. For example, resource utilization attacks are common to all stateful IDSes, and previous work addresses these issues. In addition, evasion attacks are common to network-based IDSes, and techniques such as normalization have been proposed to defend against them. We do not discuss these more general attacks here; We focus on Polygraph-specific attacks.

Overtraining Attacks: The conjunction and token-subsequence algorithms are designed to extract the *most specific* signature possible from a worm. An attacker may attempt to exploit this property to prevent the generated signature from being sufficiently general.

We call one such attack the *coincidental-pattern* attack. Rather than filling in wildcard bytes with values chosen uniformly at random, the attacker selects from a smaller distribution. The result is that there tend to be many substrings coincidentally in common in the suspicious pool that do not actually occur in every sample of the worm.

We evaluate Polygraph's resilience to this attack by modifying the Apache-Knacker exploit to set each of the approximately 900 wildcard bytes to one of only two values.

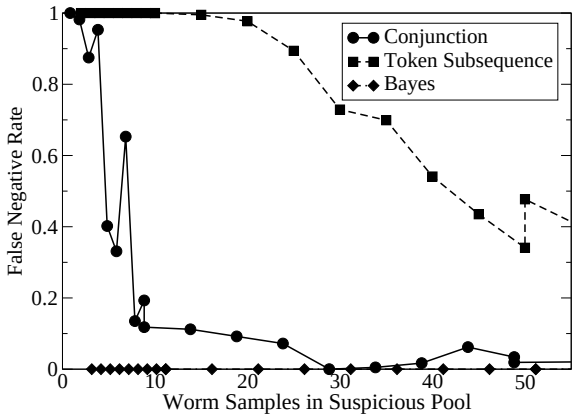


Figure 6. Number of worm samples required when under 'coincidental-pattern' attack.

We used between 2 and 50 samples generated in this way in Polygraph's suspicious pool, and another 1000 generated in this way to measure the false negative rates.

Figure 6 shows the false negative rates of Polygraph's signatures generated for this attack. Bayes is resilient to the attack, because it does not require *every* one of the common tokens to be present to match. However, the conjunction and subsequence algorithms need a greater number of worm samples than before to create a sufficiently general signature.

Another concern is the *red herring* attack, where a worm initially specifies some fixed tokens to appear within the wildcard bytes, causing them to be incorporated into signatures. Over time, the worm can stop including these tokens, thus causing previously produced signatures to no longer match. Again, Bayes should be resilient to this attack, because it does not require every token in its signature to be present to match a worm flow. For the other signature classes, the signature must be regenerated each time the worm stops including a token.

Innocuous Pool Poisoning: After creating a polymorphic worm, an attacker could determine what signatures Polygraph would generate for it. He could then create otherwise innocuous flows that match these signatures, and try to get them into Polygraph's innocuous flow pool. If he is successful, then the worm signature will seem to cause a high false positive rate. As a result, the signature may not be generated at all (when using clustering), or the system may conclude that the signature is insufficiently specific.

This problem can be addressed in several ways. One way is for Polygraph to collect the the innocuous pool using a sliding window, always using a pool that is relatively old (perhaps one month). The attacker must then wait for this time period between creating a worm and releasing it. In the case of a non-zero-day exploit, a longer window gives time for other defenses, such as patching the vulnerability.

Another defense is to deploy distributed Polygraph monitors, each using a locally collected innocuous pool. This would make it significantly more difficult for the attacker to poison all innocuous pools. It also offers other added benefits, such as decreasing the time needed to detect a new worm.

Long-tail Attack: Matching on network flows is tricky, because we cannot examine the entire flow at once—we must let packets through. Sometimes an exploit could have already occurred by the time we see a full signature match. For the token-subsequence signature, it may be desirable to prune off the end of the signature, and keep just enough to ensure few false positives. For a Bayes signature, the flow can be matched before we see every token in the signature. In either case, it may also be desirable to buffer/throttle streams that are in the middle of a partial match.

7. Discussion

We have presented three classes of signatures, and an algorithm to generate signatures for each class. The question remains—which algorithm and signature class should be used?

All three signature classes have advantages and disadvantages. The token-subsequence signature class produces ordered signatures that are more specific than the equivalent unordered conjunction signatures. However, some exploits may contain invariants that can appear in different orders. In that case, the token-subsequence signature will consist of only the tokens that appear in a consistent order, and may actually be less specific than a conjunction signature for the same worm. The Bayes signature class can be generated more quickly than the others, and is more useful when there are tokens that only appear some of the time.

All three algorithms are promising, but no one algorithm is superior to the others for every worm. The most resilient approach for using these algorithms is to consider *all three*, and use the signature that appears to have the fewest false positives and false negatives.

8. Related Work

The Bro [20] and Snort [21] IDSes monitor network traffic, and search the monitored traffic for signatures of known worms and other intrusions; Polygraph solves the complementary problem of how to provide the signature database required by an IDS automatically. We note that both Bro and Snort support matching regular expression signatures in reassembled TCP flows, and thus already support matching the token subsequence signatures generated by Poly-

graph. Shield [26] uses manually generated vulnerability-based signatures to filter out attack flows on a host. These vulnerability-based signatures could be effective against polymorphic worms, however, they need to be manually generated.

Honeycomb [14], Autograph [13], and EarlyBird [22] all generate signatures for novel worms automatically. While these three systems produce signatures by different means, they share a common signature type: a single, contiguous string. As discussed in Section 2.4, these signatures often fail to match polymorphic worm payloads robustly.

TaintCheck [18] is a tool for automatic exploit detection and signature generation. Besides being able to detect very accurately when monitored software is exploited, it proposes a novel method for automatic signature generation, *semantic-based* signature generation—instead of the *pattern-based* signature generation methods in HoneyComb [14], EarlyBird [22], Autograph [13], and Polygraph, which is based on extracting common patterns in sample flows, TaintCheck proposes to automatically generate signatures using information about the vulnerability and how it is exploited. As a first step, TaintCheck demonstrated how to automatically extract values used to overwrite the high-order bytes of the jump target and use that as an invariant part of a signature. Extensions to such semantic-based signature generation could be used to identify other invariant parts for signatures such as protocol framing [18]. Semantic-based signature generation could serve as a complimentary approach for signature generation for polymorphic worms.

Finally, the problem of *motif-finding* in computational biology is reminiscent of that of finding a signature for a polymorphic worm: given long strings, find extremely similar short strings (motifs) contained in them [16, 8, 27, 28]. The underlying assumption in motif-finding, however, is that differences in non-motif regions of the strings are changes caused by random perturbations that occur at a significantly lower rate in the motif regions.

9. Conclusion

The growing consensus in the security community is that polymorphic worms portend the death of content-based worm quarantine—that no sensitive and specific signatures may exist for worms that vary their content significantly on every infection. We have shown, on the contrary, that *content-based filtering holds great promise for quarantine of polymorphic worms*—and moreover, that Polygraph can derive sensitive and specific signatures for polymorphic worms *automatically.* We observe that it is the rigidity of many software vulnerabilities that enables matching of polymorphic worms by fixed signatures; exploits frequently must incorporate invariant byte sequences in order to function correctly. We have defined the signature generation problem for polymorphic worms; provided examples that illustrate the presence of invariant content in polymorphic worms; proposed conjunction, token-subsequence, and Bayes signatures classes that specifically target matching polymorphic worms' invariant content; and offered and evaluated Polygraph, a suite of algorithms that automatically derive signatures in these classes. Our results indicate that even in the presence of misclassified noise flows, and multiple worms, Polygraph generates high-quality signatures. We conclude that the rumors of the demise of content-based filtering for worm quarantine are decidedly exaggerated.

10. Acknowledgments

Thanks to Dan Ferullo, David Brumley, and Hyang-Ah Kim for their help in dissecting exploits. Thanks to Shobha Venkataraman for feedback and pointers into machine learning literature. Thanks to Lisa Sittler and James Gurganus for their assistance in gathering network traces. Finally, thanks to the anonymous reviewers for their insightful feedback.

References

[1] CERT advisory CA-2002-27. `http://www.cert.org/advisories/CA-2002-27.html`.

[2] CERT advisory CA-2003-04. `http://www.cert.org/advisories/CA-2003-04.html`.

[3] K2, admmutate. `http://www.ktwo.ca/c/ADMmutate-0.8.4.tar.gz`.

[4] The Metasploit project. `http://www.metasploit.com`.

[5] SANS Institute: Lion worm. `http://www.sans.org/y2k/lion.htm`.

[6] Symantec Security Response: CodeRed Worm. `http://www.sarc.com/avcenter/venc/data/codered.worm.html`.

[7] Viruslist.com: Net-Worm.Linux.Adm. `http://www.viruslist.com/en/viruses/encyclopedia?virusid=23854`.

[8] T. L. Bailey and C. Elkan. Fitting a mixture model by expectation maximization to discover motifs in biopolymers. In *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, pages 28–36. AAAI Press, Menlo Park, California, 1994.

[9] C. CAN-2003-0245. Apache apr-psprintf memory corruption vulnerability. `http://www.securityfocus.com/bid/7723/discussion/`.

[10] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. V. Underduk. Polymorphic shellcode engine using spectrum analysis. `http://www.phrack.org/show.php?p=61&a=9`.

[11] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[12] L. Hui. Color set size problem with applications to string matching. In *Proceedings of the 3rd Symposium on Combinatorial Patttern Matching*, 1992.

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

[14] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.
[15] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.
[16] C. E. Lawrence, S. S. Altschul, M. S. Boguski, J. S. Liu, A. F. Neuwald, and J. C. Wootton. Detecting subtle sequence signals: A gibbs sampling sequence strategy for multiple alignment. *Science*, 262:208–214, 1993.
[17] C. Nachanberg. Computer virus-antivirus coevolution. *Communications of The ACM*, 1997.
[18] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, Feb. 2005.
[19] A. Pasupulati, J. Coit, K. Levitt, and F. Wu. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *IEEE/IFIP Network Operation and Management Symposium*, May 2004.
[20] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24), December 1999.
[21] T. S. Project. Snort, the open-source network intrusion detection system. http://www.snort.org/.
[22] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2004.
[23] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.
[24] P. Szor. Hunting for metamorphic. In *Proceedings of the Virus Bulletin Conference*, 2001.
[25] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In *Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS)*, Washington, DC, October 2004.
[26] H. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM*, Portland, OR, Aug. 2004.
[27] E. P. Xing and R. M. Karp. Motifprototyper: a profile bayesian model for motif family. In *Proceedings of National Academy of Sciences*, volume 101, 2004.
[28] E. P. Xing, W. Wu, M. Jordan, and R. Karp. Logos: A modular bayesian model for *de novo* motif detection. *Journal of Bioinformatics and Computational Biology*, 2(1):127–154, 2004.

A. Token Distinguishability

We find that it is useful to estimate how likely a token is to appear in innocuous network traffic. This information can be used as a relative measure of how distinguishing one token is compared to another. That is, given two tokens that appear in worm samples equally as often, the one that appears in innocuous traffic *less* often is more distinguishing than the other. Such information can also be used to help estimate the false positive rate of a signature.

One often-used heuristic is that, in general, longer strings are more useful in a signature than short strings. In particular, one automatic signature generation algorithm is to find the longest substring common to a set of suspicious samples [15]. We wish to quantify how *much* better a longer substring is than a shorter substring. This can help to answer questions such as, "Which is a more specific signature- a twenty byte string or the conjunction of two twelve byte strings?" Additionally, considering the length of a string alone can be quite misleading. While the string "HTTP/1.1\r\n" is longer than the string "\xBF\xFF\xFE", it is still much more likely to appear in innocuous requests on port 80.

To calculate the probability that pattern string P occurs in some input text T, we construct a finite state machine that takes T as input, and reaches the accept state if and when P is found. In this machine, there is a state for each character in P, plus the initial state. We denote the probability that the machine is in state i after processing n characters as $Pr_i(n)$. Our goal, then, is to calculate $Pr_{l(P)}(l(T))$, where $l(P)$ is the length of the pattern P, and $l(T)$ is the length of the text T. To calculate $Pr_{l(P)}(l(T))$, we iteratively compute the probability of being in each state after processing each input character. This calculation can be performed in $O(l(P) * l(T))$ time.

To calculate the probability that text matching an input with a uniform-random byte distribution, we set the probability that a transition is taken equal to the number of characters on its label, divided by the size of the alphabet.

We can use the same technique to estimate the probability that a string occurs in a network flow of a particular protocol. Instead of assuming that each character in the alphabet is equally likely, we can use traces of the protocol to measure the relative frequency of each character. We can use these measurements to estimate the probability of each transition in the finite state machine. This model can be further improved by measuring the relative frequency of each character, given the current *context*. For example, if the flow consists of English text, and the last character matched was "q", it is very likely that the next character will be "u". Again, these statistics can easily be measured from traces of a protocol. Compared to simply measuring the frequency of the string in the trace, this technique offers a more generalized view of the protocol. For example, while the trace might not have any requests containing the exact string "www.obscuresite.com", this technique can calculate that it is more plausible than the equally long string "dJesOuruhamjRhoEfvi", since substrings including "www." and ".com" are common, it fits the model for English words, *etc.*