Catch Me, If You Can: Evading Network Signatures with Web-based Polymorphic Worms

Matthew Van Gundy
University of California, Davis
mdvangundy@ucdavis.edu

Davide Balzarotti, Giovanni Vigna
University of California, Santa Barbara
{balzarot,vigna}@cs.ucsb.edu

Abstract

Polymorphic worms are self-replicating malware that change their representation as they spread throughout networks in order to evade worm detection systems. A number of approaches to detect polymorphic worms have been proposed. These approaches use samples of a polymorphic worm (and of benign traffic as well) to derive a signature that can detect all instances of the worm without producing excessive false positives. Even though these systems claim to be able to generate signatures for *any* type of worm, all the examples that are used to show the ability to detect polymorphic worms are based on exploits that target memory corruption vulnerabilities. In this paper, we show how a different class of worms, namely those based on web vulnerabilities and scripting languages, can be much harder to detect than "traditional" polymorphic worms. We developed a polymorphic engine for PHP code and we tested the ability of state-of-the-art tools to detect this type of worm. The results of our experiments show that a PHP-based polymorphic worm would be able to successfully evade existing signature generation systems.

Keywords: Polymorphic Worms, PHP, Attack Mutations, Network Intrusion Detection, Signature Evasion.

1 Introduction

Polymorphic worms represent a serious threat [15]. As worms, they are able to spread throughout a network in a very limited amount of time [13, 16]; as polymorphic malware, they are able to evade simple detection systems and their signature generation components.

Even though large-scale, highly-polymorphic worms have not yet appeared in the wild, there has been a substantial amount of research whose goal is to develop techniques that can identify polymorphic worms in a reliable way [9, 8, 7].

These techniques have been implemented in tools that appear to be able to generate signatures for polymorphic worms by analyzing samples of both malicious and benign traffic. Even though most systems claim to be able to generate signatures for *any* type of worm, the examples used to evaluate the proposed techniques are always based on worms that exploit memory corruption vulnerabilities (e.g., a buffer/heap overflow or a format string vulnerability) in order to execute arbitrary binary code.

For example the authors of Polygraph [9] claim that they *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.* Nonetheless, the system is then evaluated on attacks such as a hypothetical worm (based on the Apache-Knacker vulnerability [4]) and the Lion worm (based on the BIND TSIG vulnerability [12]), both of which exploit memory corruption vulnerabilities.

Similarly, in [8] the authors state that *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.* Even though, this claim is rather general, the proposed technique is evaluated on Code Red II, Apache-Knacker, ATPhttpd, and, in addition, on the CLET [6] and TAPiON [1] shellcode generation engines.

This lack of coverage of other possible types of worms prompted us to try to understand if the assumptions made by these approaches are valid for all types of worms. Therefore, we developed a PHP-based worm that exploits a web-based vulnerability. Web-based vulnerabilities are very common. A report published by Symantec in March 2007 states that, out of the 2,526 vulnerabilities

that were documented in the second half of 2006, 66% affected web applications [14], and many web-based vulnerabilities allow for arbitrary code execution.

Therefore, we developed a polymorphic engine for PHP, we created a polymorphic worm that exploits an arbitrary code execution vulnerability, and we evaluated it with respect to two state-of-the-art tools for polymorphic worm detection, namely Polygraph and Hamsa. The results show that our PHP-based worm is able to evade detection by existing pattern extraction tools.

The contributions of this paper are the following:

- We developed PHolyP, a PHP polymorphic engine that is able to encrypt a PHP payload and obfuscate the corresponding decryption routine.
- We developed a polymorphic worm that exploits a PHP arbitrary code execution vulnerability, and, for the first time, we analyzed the ability of existing polymorphic worm detection systems to identify this class of worms, showing that this type of worm is able to evade detection by Hamsa and Polygraph.

The rest of this paper is structured as follows. In Section 2, we present our PHP polymorphic engine. In Section 3, we present how the detection systems being evaluated operate. Then, in Section 4, we describe our PHP-based worm, and how we experimentally evaluated the detection rate of the systems being analyzed. Section 5 presents related work on evasion of polymorphic worm detectors. Finally, Section 6 briefly concludes.

2 PHolyP: A Polymorphic PHP Engine

In the virus literature, the term "polymorphism" describes an approach used to modify a virus' code in order to hide its presence from anti-virus software [3].

The same technique can be applied to any form of malicious code. In particular, a *polymorphic* worm is a worm that changes its appearance at each infection. In general, this is achieved by encrypting the worm body each time with a different key, and by appending (or prepending) the code required to decrypt and execute the payload. The encrypted body also carries a special module (usually called *polymorphic engine*) that is responsible of generating a different decryption routine at each infection.

A perfect polymorphic worm should not contain any recurring pattern of bytes that can be matched by a signature-based intrusion detection system. However, writing a perfect polymorphic code is a very difficult task.

For example, the results of a recent experiment [8] have shown how even the best mutation engines available for binary code leave distinctive traces that can be identified by a properly-written set of signatures.

Our target is to show how, using a web-based worm written in a scripting language, it is possible to easily achieve a level of polymorphism that can evade the current state-of-the-art systems for polymorphic worm detection.

To support our hypothesis, we designed and implemented a polymorphic engine for the PHP language. The engine was then integrated in an automated tool called *PHolyP*. PHolyP takes a PHP source file as input and generates the polymorphic version of the code as output. Both the user code and the polymorphic engine are encrypted with a random key and the result is placed in a temporary variable. At runtime, a decryption routine decrypts the payload, retrieves the original code, and finally executes it through an `eval` statement.

The only part of the code that can potentially be matched by a signature is the decryption routine. To avoid the presence of any constant sequence of bytes, a number of transformation techniques are applied to the generation of the decryption code:

- *Randomization of variables names*:
  the name of each PHP variable is substituted with a random string of variable length.
- *Randomization of the cryptographic routine*:
  the cryptographic algorithm used to encrypt/decrypt the worm body is randomly chosen at each worm propagation. The current implementation chooses between a simple XOR-based encryption, DES, 3DES, BLOWFISH, and XTEA algorithms.
- *Comment insertion*:
  randomly generated comments are inserted inside each line of code. The comment position and the delimiter character are also randomly chosen. For example, the assignment

  ```
  $X = 2;
  ```

  can be obfuscated introducing a number of innocuous comments as follows:

  ```
  $X\*aB88*\=\*-&*\2;#blah
  ```

- *Space separator substitution*:
  separator characters can either be removed, or re-

placed with an arbitrary number of spaces, tabs, newlines, or any combination thereof.

- *NOP insertion*:
  the code is modified by interleaving a number of NOP-equivalent instructions with the legitimate decryption code. A NOP-equivalent instruction is an instruction that does not affect the execution of the program. For example, fake assignments or useless function calls can be inserted anywhere in the program. A very simple and easily randomizable NOP-like instruction can be generated as a consequence of the fact that the PHP interpreter ignores any line containing just a variable name or bare string. For example:

```
$A = 1;
$B = 2;
```

  becomes:

```
$A = 1;
$aX77_aA9AFF0fa_s;
$B = 2;
URqSbhgJ6ahoDlSY8;
```

- *Instruction shuffling*:
  some of the decryption routine's instructions can be safely reordered without altering the code behavior. For instance, the order in which variables are declared does not affect the execution of the decryption routine.

- *Function name randomization*:
  in PHP, function names are case-insensitive. This allows the polymorphic engine to randomly change the case of the letters that compose any function name in the code.

- *Code nesting*:
  in order to reduce the total number of semicolons, the polymorphic engine collapses some of the decryption instructions, nesting most of the instruction together inside a single line. For example:

```
$x = "...";
$y = decrypt($x);
eval($y);
```

  becomes:

```
eval(decrypt("..."));
```

- *String delimiter substitution*:
  this simple transformation can be used to substitute all the string delimiters with either single quotes or double quotes. This can be very important, because the frequent appearance of a specific type of quotes would easily be detected by the signature generation tools.

- *Function calls through randomly split string variables*:
  another PHP feature that is very useful when obfuscating the code is the ability to invoke a function using a variable that contains the function name. For example, in order to call the `strlen` function, in PHP it is possible to use the following code:

```
$temp = "strlen";
$len = $temp("hello");
```

  This functionality, combined with the fact that strings can easily be obfuscated and split in many different pieces, allows our polymorphic engine to obfuscate each function call in a very effective way. For example:

```
$x = strlen($y);
```

  becomes:

```
$tmp1 = "e"."N";
$tmp2 = "S"."trL";
$f = $tmp2.$tmp1;
$x = $f($y);
```

While effective in obfuscating PHP code, some of the previous transformations can leave in the program some recognizable footprints (such as an anomalous number of comment-delimiting characters). In order to avoid the presence of these patterns, PHolyP applies only a random subset of all the possible transformations to generate a certain worm instance. PHolyP does not, however, attempt to evade anomaly detection (or other non-pattern extraction) systems.

The current implementation of the PHolyP polymorphic engine consists of only 408 lines of PHP code.

3 Detection of Polymorphic Worms

Signature generation systems, such as Polygraph and Hamsa, attempt to derive signatures for polymorphic worms from pools of network traffic (reassembled application layer traffic from a number of different network

connections). Before being fed to the signature generation system, a flow classifier separates network traffic into two pools: the innocuous pool and the suspicious pool. The innocuous pool contains traffic known to be legitimate, while the suspicious pool contains traffic believed to contain worm instances.

The goal of signature generation is to output one or more signatures which match a large fraction of the network flows in the suspicious pool while matching at most a very small fraction of the innocuous pool (0.001% is typically considered acceptable). Failing to match a worm instance is known as a *false negative* while matching an innocuous flow is known as a *false positive*. In the following, we use the terms *false positive* and *false negative* to refer to both flows in the training pools and flows in the testing pools.

Both Polygraph and Hamsa begin signature generation by extracting tokens (substrings) from the suspicious pool when they appear in a fraction of flows greater than some threshold (3 flows for Polygraph, 15% for Hamsa). The systems differ in the way in which they handle tokens that are substrings of another token. Polygraph only keeps such a token if its occurrence independent of the other tokens is above the token extraction threshold. By contrast, Hamsa keeps all tokens regardless of whether or not they occur independently.

Both systems then attempt to find a combination of tokens that yields a good signature. The strategy employed depends on the type of signature being generated.

- *Polygraph Conjunction*:
  Polygraph's Conjunction signatures consist of a set of tokens. A flow matches a signature if, for each token t_i in the signature, t_i is also contained in the flow. A Conjunction signature for a single flow is the set of tokens present in that flow. To generalize a Conjunction signature to multiple flows, Polygraph takes the intersection of the signatures for all flows in question.

- *Polygraph Token Subsequence*:
  Polygraph's Token Subsequence signatures are an ordered set of tokens. A flow matches a signature if, for each token t_i in the signature, t_i occurs in the flow and for all t_i, t_j, if t_i occurs before t_j in the signature, t_i must occur before t_j in the flow. A Token Subsequence signature for a single flow is the ordered set of tokens which appear in the flow. To generalize a Token Subsequence signature, Polygraph uses a string alignment algorithm which attempts to maximize the number of consecutive matches in the resulting sequence.

- *Hamsa Multiset*:
  Hamsa's Multiset signatures are a set of token-frequency 2-tuples. A flow matches a signature if, for each signature token t_i and the associated frequency n_i, the flow has at least n_i occurrences of t_i.

  To create Multiset signatures Hamsa employs a model $\Gamma(\cdot)$ that allows it to generate signatures in a greedy fashion. The $\Gamma(\cdot)$ model bounds the maximum allowable false positive rate that a signature may have among the training flows as a function of the number of tokens in the signature. Hamsa begins with the empty signature. At step i, Hamsa considers the tokens that, when added to the current signature, have a false positive rate less than $\Gamma(i)$. Among these tokens, Hamsa chooses the token that is contained in the largest fraction of the suspicious pool. The process continues until there is no such token or the maximum number of tokens (15) has been chosen.

  After generating a signature, Hamsa attempts to lower the potential false positive rate by extending the length of all tokens in the signature as long as they do not decrease the signature's coverage in the suspicious pool.

When generating signatures, Polygraph can employ a technique called Hierarchical Clustering. Without Hierarchical Clustering, signature generation regards the entire suspicious pool as a single cluster and outputs a single signature for the whole pool. When employing Hierarchical Clustering, Polygraph places each suspicious pool flow into its own cluster and generates a signature for each cluster. It then iteratively merges the two clusters that, when combined, will yield a signature with the lowest false positive rate in the innocuous pool. When no two clusters can be merged, Polygraph outputs the signatures for all remaining clusters. Hierarchical Clustering allows Polygraph to generate more specific signatures than would otherwise be possible if the suspicious pool were considered as a whole. This is important when either instances of multiple different worms or noise are present in the suspicious pool. The suspicious pools used in our tests contained only instances of our polymorphic worm without any noise flows.

Finally, Polygraph is also capable of generating Bayesian signatures. We do not consider Polygraph's Bayesian signatures in this work, however, because a number of very effective ways to evade this type of signature have already been presented in other papers [11, 10].

4 Experimental Evaluation

The purpose of our experiments is to show how a polymorphic web-based worm can successfully avoid the current state-of-the-art signature generation tools. However, it is important to note that our goal is not to develop (neither to show how to develop) a full-fledged web worm. In fact, the existence of this type of malware has already been observed "in the wild" in the past few years. For example, the Santy worm [5], which was first detected in December 2004, infected the phpBB discussion forums using the Google search engine to find its victims.

We started our evaluation by identifying a vulnerability inside a PHP application that allows for arbitrary code execution. We selected a vulnerability in the Limbo CMS system [2]. In this case, an attacker can execute arbitrary PHP code on the computer running the vulnerable application, due to a failure in the application to properly sanitize the user-supplied `Itemid` parameter to the `index.php` script.

We then wrote a proof-of-concept worm that exploited the Limbo vulnerability. This simple memory-resident hit-list worm [13] uploads a copy of itself in Limbo's `Itemid` parameter where it gains execution on the remote system and exploits its portion of the hit-list. In order to successfully exploit Limbo, the worm formats itself as a single PHP expression that Limbo includes in a call to the PHP `eval` function.

This basic implementation was then transformed into a polymorphic worm using our PHolyP tool. In addition to the set of general purpose mutation techniques applied by our polymorphic engine, we also added a few specific transformations to increase the randomness of the worm instances. For example, the worm non-deterministically chooses the request method and payload encoding format. It non-deterministically url-encodes certain characters to reduce the presence of certain tokens. Also, it chooses non-deterministically whether or not to apply a content encoding to the request body, it pads the length of the request in order to avoid trivial signatures on the size of the worm's representation, it randomizes the order and appearance of HTTP headers, it non-deterministically places some of the parameters in the query string, and it prefixes the payload with random sub-expressions.

Finally, in order to test the signature generation tools, we needed to create a dataset containing some attack-free Limbo traffic and a malicious dataset containing instances of our worm. We created the clean dataset as a composition of outgoing web requests originating from a local LAN, of traffic collected by manually using the Limbo application, and of traffic automatically generated by running a set of scripts that simulated realistic user activity. The scripts used in the experiments are based on a browser (the KHTML component that is part of the KDE library) controlled by a python program. Two sets of scripts, one simulating a registered user and one simulating a casual visitor, were programmed to navigate through the web pages and to randomly submit realistic data to the various application forms. This traffic was coalesced into innocuous training and test pools containing 9,393 and 32,286 flows, respectively.

The malicious traffic was generated by using a worm instance to repeatedly generate requests designed to exploit a web server running the vulnerable application. Each request included a new worm variant randomly generated by our polymorphic engine.

When testing Polygraph and Hamsa, we used exactly the same settings presented in the original papers [9, 8]. We conducted 5 trials, each testing suspicious training pools containing 5, 10, 25, 50, 100, and 200 worm variants along with a suspicious test pool containing 15,049 variants. In each case, the suspicious pool contained only variants of a single polymorphic worm without any noise – a best-case scenario for the signature generation systems.

Unfortunately, in order to make Polygraph's runtimes tractable, we were forced to truncate the worm payload in all flows to limit the maximum size of each flow to approximately 1KB. This gave Polygraph an unfair advantage over Hamsa, however, it was necessary in order to be able to conduct our experiments in a timely fashion. Also due to performance constraints, we do not give results for Polygraph's Hierarchical Clustering (HC) signatures with a training pool size of 200 flows.

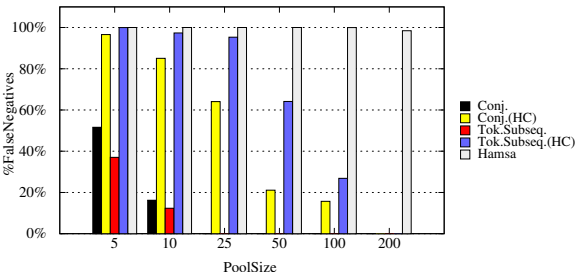

Figure 1: False Negatives by signature type

Figures 1 and 2 depict the median performance of the different types of signatures over the 5 trials. For Polygraph, results are shown with Hierarchical Clustering both enabled and disabled. Hamsa's Multiset signature and Polygraph's signatures with Hierarchical Clustering

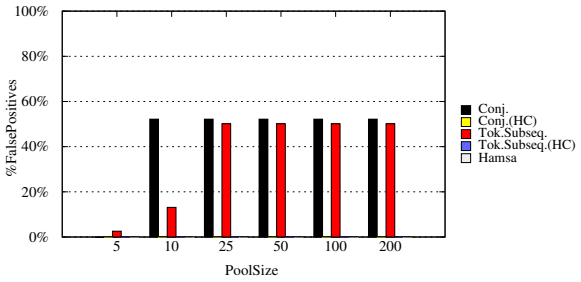

Figure 2: False Positives by signature type

enabled all exhibit non-negligible false negative rates because they are overly specific. On the other hand, Polygraph's signatures without Hierarchical Clustering have no false negatives because they are too general. The Conjunction signatures have a false positive rate of 52.17% while the Token Subsequence signatures exhibit a false positive rate of 50.13% (far too high to be useful).

Hamsa's signatures pick up a large number of tokens that are present coincidentally in the encrypted payload of the worm variants in the suspicious pool causing matching to fail on many worm variants that do not contain one or more of the tokens. A representative Hamsa signature can be found in Appendix A.

The Polygraph signatures without Hierarchical Clustering err in the other direction. They effectively block *all* requests to the `/index.php` script – creating a denial of service condition for any URLs that contain the substring `/index.php`. Due to their extreme generality, the signatures match a significant number of innocuous flows as well. Examples of Polygraph signatures are shown in Appendix B.

In the initial version of this paper, we reported the false negative rates for Polygraph's HC enabled Conjunction and Token Subsequence signatures as 82.67% and 91.57% respectively. While preparing the final version of this paper, we discovered that Polygraph was reporting inaccurate results when HC was enabled. After fixing Polygraph we determined that the actual false negative rates were 15.93% and 18.18%. While these results are far more modest, they are still non-negligible success probabilities – more than 1 in 6 worm instances evades the signatures. Through preliminary additional testing we have created variants of our worm that achieve median false negative rates of 15.71% against Conjunction signatures and 26.85% against Token Subsequence signatures with HC enabled. We believe that these results could be improved even further with additional development effort as we have not fully explored all the possibilities for polymorphism available to worms targeting web applications.

5 Related Work

In [11], the authors present attacks against Polygraph's algorithms both for deriving Conjunction and Token Subsequence signatures and for deriving Bayesian signatures. The authors demonstrate that an attacker can use noise injected into Polygraph's suspicious pool to cause Polygraph's clustering algorithm to exclude a worm's invariants from the signatures that are generated. The authors then proceed to demonstrate how including substrings of tokens that are moderately common in innocuous traffic can be used to defeat Polygraph's Bayesian learner. A worm containing substrings of tokens found in innocuous traffic can artificially decrease the matching threshold set by Polygraph. Polygraph's Bayesian learner is then forced to choose between unacceptably high false positive rates or unacceptably high false negative rates.

Newsome et al. [10] strengthen and generalize the attack on Polygraph's Bayesian learner presented by [11] in what they refer to as the Correlated Outlier Attack. They demonstrate that an attacker can force the learner to choose between high false positives and high false negatives without needing to inject noise into the suspicious pool. They also demonstrate that the attack may be strengthened further by poisoning of the innocuous pool (perhaps long before the vulnerability is discovered).

Newsome et al. also present several attacks against Polygraph's Conjunction and Token Subsequence signatures known as Red Herring attacks. These attacks use coincidental patterns, or pseudo-invariants, that are removed over time in order to cause Polygraph to derive signatures that are too specific to match most instances of a worm. The authors note that, while not immune to their Red Herring attacks, Hamsa is much less susceptible than Polygraph.

Rather than attacking Polygraph or Hamsa's techniques directly, as is the case with previous work, we demonstrate that the degree of polymorphism available to worms that do not exploit memory corruption vulnerabilities can prevent these systems from deriving precise signatures despite being trained in an idealized (noise-free) setting. In some sense, the high degree of randomness present in our worm could be considered to be a coincidental Red Herring attack. However, we have made no attempt to directly attack either system, only to pro-

duce worm variants with the highest degree of polymorphism possible.

6 Conclusions

In this paper we presented an analysis of the ability of state-of-the-art polymorphic worm detection systems to detect worms that do not exploit memory corruption vulnerabilities. To this end, we developed a novel PHP-based worm that exploits an arbitrary PHP code execution vulnerability and we tested the ability of two worm detection systems to classify this kind of worm. The results show that many of the assumptions that are at the basis of existing detection techniques (e.g., the fact that the address used to overwrite a pointer must contain some constant part) do not hold for this type of worm.

References

[1] Piotr Bania. TAPiON Polymorphic Decryptor Generator. `http://www.piotrbania.com/all/tapion/`, 2005.

[2] BID-16902. Limbo CMS Frontpage Arbitrary PHP Command Execution Vulnerability. `http://www.securityfocus.com/bid/16902/`.

[3] V. Bontchev. Future Trends in Virus Writing. White Paper, 1994.

[4] CAN-2003-0245. Apache APR_PSPrintf Memory Corruption Vulnerability. `http://www.securityfocus.com/bid/7723`.

[5] SANS Internet Storm Center. Santy worm. `http://isc.sans.org/diary.html?storyid=399`.

[6] T. DeTristan, T. Ulenspiegel, Y. Malcom, and M. von Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. `http://www.phrack.org/show.php?p=61&a=9`.

[7] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, volume 3858 of *LNCS*, pages 207–226, Seattle, WA, September 2005. Springer-Verlag.

[8] Z. Li, M. Sanghi, Y. Chen, M.Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P06)*, pages 32–47, 2006.

[9] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Symposium on Security and Privacy*, 2005.

[10] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting Signature Learning by Training Maliciously. In *Proceedings of RAID 2006*, pages 81–105, September 2006.

[11] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading Worm Signature Generators Using Deliberate Noise Injection. In *IEEE Symposium on Security and Privacy*, pages 17–31, May 2006.

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

[13] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the internet in your spare time. In *Proceedings of the 11th USENIX Security*, 2002.

[14] Symantec. Symantec internet security threat report, March 2007.

[15] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A Taxonomy of Computer Worms. In *ACM Workshop on Rapid Malcode*, October 2003.

[16] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *13th Usenix Security Symposium*, 2004.

A Representative Hamsa Signature

The following (truncated) signature was generated by Hamsa for a suspicious pool size of 200 variants. It yields a test pool false negative rate of 91.84% with no false positives. It consists primarily of tokens that occur coincidently within the encrypted payload of the worm variants.

{'G7': 1, 'G6': 1, 'G5': 1, 'G3': 1, 'G1': 1, 'G9': 1, 'G8': 1, 'G%': 1, 'GW': 1, 'GV': 1, 'GU': 2, 'GS': 1, 'GP': 1, 'GZ': 1, 'GY': 1, 'GX': 1, 'GF': 1, 'GE': 1,

'GD': 1, 'GA': 1, 'GM': 1, 'GJ': 1, 'GI': 1, 'Gw': 1, 'Gt': 1, 'Gq': 1, 'Gp': 1, 'Gz': 1, 'Gg': 2, 'Gd': 1, 'Gb': 1, 'Gj': 1, 'Gh': 1, 'Z8': 1, 'Z6': 2, 'Z0': 1, 'Z1': 1, 'Z2': 1, 'Z3': 1, 'ZL': 1, 'ZM': 1, 'ZN': 1, 'ZH': 1, 'ZJ': 1, 'ZD': 1, 'ZE': 1, 'ZF': 1, 'ZG': 1, 'ZB': 1, 'ZC': 1, 'ZT': 1, 'ZU': 1, 'ZV': 1, 'ZW': 1, 'ZS': 1, 'Zl': 1, 'Zm': 1, 'Zb': 1, '3f': 1, 'Zy': 1, 'Zv': 1, 'Zw': 1, 'Zp': 1, 'Zq': 1, 'Zr': 1, '9i': 1, '9h': 1, '9j': 1, '9l': 1, '9d': 1, '9y': 1, '9x': 1, '9p': 1, '9s': 1, '9r': 1, '9u': 1, '9t': 1, '9w': 1, '9v': 1, '9H': 1, '9M': 2, '9N': 1, '9C': 1, '9B': 1, '9F': 1, 'm4': 1, 'm6': 1, 'm1': 1, 'm0': 1, 'm3': 1, '9U': 1, '9T': 1, '9W': 1, 'mE': 1, 'mD': 1, 'mG': 1, 'mF': 1, 'mA': 1, '9%': 5, 'mU': 1, 'mW': 1, 'mV': 1, 'mQ': 2, 'mP': 1, 'mR': 1, '91': 1, '93': 1, '92': 2, 'mY': 1, 'mZ': 1, 'me': 1, 'mg': 1, 'mb': 2, 'mj': 1, 'mt': 1, 'mw': 1, 'ms': 1, 'mx': 1, 'mz': 1, 'Lg': 1, 'Ld': 1, 'Le': 2, 'Ln': 2, 'Lm': 1, 'Lj': 1, 'Lv': 1, 'Lw': 1, 'Lr': 1, 'Ls': 1, 'LG': 1, 'LD': 1, 'LC': 1, 'LA': 1, 'LM': 1, 'LH': 1, 'LI': 1, 'LV': 1, 'LT': 1, 'LU': 1, 'LR': 1, 'L7': 1, 'L4': 1, 'L8': 1, '1P': 1, '1V': 1, 'V2': 1, 'V5': 1, '1Z': 2, 'iN': 1, '%2F%2': 1, 'rT': 1, 'rU': 1, 'rW': 1, 'rP': 2, 'rQ': 1, 'rR': 1, 'rX': 1, 'rY': 1, 'rZ': 2, 'rC': 1, 'rL': 1, 'rM': 1, 'rO': 2, 'rH': 1, 'rI': 1, 'rJ': 1, 'rK': 1, 'rt': 1, 'rq': 1, 'rr': 1, 'rx': 1, 'i1': 1, 'rc': 1, 'ro': 2, 'rh': 1, 'ri': 1, 'i2': 1, '1p': 1, '1s': 1, '1r': 1, 'i7': 2, 'r6': 1, 'r2': 1, '1v': 1, '1b': 1, '1h': 1, 'Q1': 1, 'Q3': 1, 'Q5': 1, 'Q4': 1, 'Q7': 1, 'Q6': 1, 'Q9': 1, 'Q8': 1, 'Qp': 1, 'Qs': 1, 'Qu': 2, 'Qt': 1, ...

B Representative Polygraph Signatures

B.1 Conjunction (HC disabled)

The following signature was generated by Polygraph for a suspicious pool of 200 variants. It has a 52.16% false positive rate with no false negatives.

{ 'T /', '/index.php', ' HTTP/1.1', '\nHost: ' }

This signature will block all HTTP 1.1 requests that contain the string `/index.php` – creating a denial of service condition for all URLs with `/index.php` as a substring.

B.2 Token Subsequence (HC disabled)

The following signature was generated by Polygraph for a suspicious pool of 200 variants. It has a false positive rate of 50.13% with no false negatives. Like the Conjunction signature above, this signature blocks all HTTP 1.1 requests to any URL containing `/index.php`.

('T ', '/index.php', ' HTTP/1.1', '\nHost: ')

B.3 Conjunction (HC enabled)

The following Conjunction signatures were generated by Polygraph for a suspicious pool of 100 variants with Hierarchical Clustering enabled. The total false negative rate (i.e. instances missed by all signatures) is 15.71% with no false positives. In all, eight individual signatures were generated. Four are shown below.

The two following signatures both capture data that is posted to `/index.php` with a Content-Encoding applied.

{ 'ww', '\nHost: ', ' HTTP/1.1', '/index.php', 'gzip', '\nContent-', 'ength: ', '\nContent-Type: application/x-www-form-urlencoded', '\nContent-Encoding: ', 'POST /', '\n\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x03', ...}

{ 'ww', 'com', 'limbo', '/index.php', 'ength: ', '\nHost: www.li', '\nContent-Encoding: ', '\n\n', 'POST /', '\nContent-Type: application/x-www-form-urlencoded\n', ' HTTP/1.1\nContent-', '\x10\xfc\x1a' }

The following two signatures result from GET and POST requests without any Content-Encoding. The first signature captures escape sequences found in the plain url-encoded payload. The second signature captures sequences found in a payload that has been encoded using quoted-printable encoding.

{ '%3', '%2', 'li', 'T /', '\nHost: ', ' HTTP/1.1', '/index.php', 'B%', '09', '%0', '%2F', '%4', '%0A', '%6', '%23', '9%2', '%2b', '%2f', '%2C%', '%2f%' }

{ '%2', 'A%3', 'T /', '\nHost: ', ' HTTP/1.1', '20', '%7', '/index.php', '1%3', 'B%', 'a%3', 'd%3', '2C', 'c%3', 'dA', '%5', 'e%3D', 'b%3', 'f%3', 'DE', 'E%', 'C%3', '2%3d', '2%3D', '%3de', '%3d8', '%3db', '%3D1', '%3D0', ...}

B.4 Token Subsequence (HC enabled)

The following signatures were generated by Polygraph with for a suspicious pool of 100 worm variants with Hierarchical Clustering enabled. The total false negative rate (i.e. instances missed by all signatures) is 26.85% with no false positives. Polygraph emitted 14 signatures for the test pool, four of which are shown below. As with the Conjunction signatures above, these signatures primarily capture requests to `/index.php` that either have a Content-Encoding applied or incorporate a significant number of url-encoding escape sequences.

('POST', 'index', 'ph', 'p', 'HTTP', '\nContent-Type: application/x-www-form-urlencoded', '\n', '\n\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x03')

('T', '/index.php', 'It%65mi', '%', '%2', '%', '%2', '%2', '%0', '%2', '%2', '%2', '%2', '%2', '2', '%2', '%2', '%2', '%', '%', '%2', '9', '%2', '%2', '%2', '%2', '%2', '%2', '%2', 'B', '%2', '%2', 'F', '%2', '%2f', '%2', '%2', '%2', '%2', '%2', '%2', '%2', '%2', '%2', '%2', '%2')

('T ', '/index.php', 't', 'e', 'id', 'V', 'L%2', '%09m', 'r', '_', '%2', '%2', '%', '%2', '%2', 'a', '%', '%2', '%2', '%2', '%2', '2', 'A%2', '%2', '%2', '%2', '%2', 'E', '2', '%2', '2', '%2', '%2', 'B', '4', '8', '%2', '%2', '%', '2', '%', '0', '%', '2', '%0', '9%09', '%2', '%2', '%2', '%', '0', '%2', 'B', '%29%', '0', '2', '9')

('GET', '/index.php?', 'I', '%6', '%28', 'rY', '_', '%2', '_', 'e', '%2c', '%2', 'E', 'E', '%28', '0', '%2', '%2', '8', '%2', 'W', '%2', '%2', '%2', '2', 'C', 'e', '%2', '%2', '2', 'D', 'P', 'e', 'S', '%2f', '%2', '%2B', '%2', 'F', '%2', 'B', '%2', '%2', '%2', '%29%2', '%2C', '0', '%2', '%29', ' HTTP/1.1\nHost: www.limbof', 'r', '.', 'o', '\n\n')