

To Catch a Predator: A Natural Language Approach for Eliciting Malicious Payloads

Sam Small
Johns Hopkins University

Joshua Mason
Johns Hopkins University

Fabian Monroe
Johns Hopkins University

Niels Provos
Google Inc.

Adam Stubblefield
Johns Hopkins University

Abstract

We present an automated, scalable, method for crafting dynamic responses to real-time network requests. Specifically, we provide a flexible technique based on natural language processing and string alignment techniques for intelligently interacting with protocols trained directly from raw network traffic. We demonstrate the utility of our approach by creating a low-interaction web-based honeypot capable of luring attacks from search worms targeting hundreds of different web applications. In just over two months, we witnessed over 368,000 attacks from more than 5,600 botnets targeting several hundred distinct webapps. The observed attacks included several exploits detected the same day the vulnerabilities were publicly disclosed. Our analysis of the payloads of these attacks reveals the state of the art in search-worm based botnets, packed with surprisingly modular and diverse functionality.

1 Introduction

Automated network attacks by malware pose a significant threat to the security of the Internet. Nowadays, web servers are quickly becoming a popular target for exploitation, primarily because once compromised, they open new avenues for infecting vulnerable clients that subsequently visit these sites. Moreover, because web servers are generally hosted on machines with significant system resources and network connectivity, they can serve as reliable platforms for hosting malware (particularly in the case of server farms), and as such, are enticing targets for attackers [25]. Indeed, lately we have witnessed a marked increase in so-called “search worms” that seek out potential victims by crawling the results returned by malevolent search-engine queries [24, 28]. While this new change in the playing field has been noted for some time now, little is known about the scope of this growing problem.

To better understand this new threat, researchers and practitioners alike have recently started to move towards the development of low-interaction, *web-based* honeypots [3]. These differ from traditional honeypots in that their only purpose is to monitor automated attacks directed at vulnerable web applications. However, web-based honeypots face a unique challenge—they are ineffective if not broadly indexed under the same queries used by malware to identify vulnerable hosts. At the same time, the large number of different web applications being attacked poses a daunting challenge, and the sheer volume of attacks calls for efficient solutions. Unfortunately, current web-based honeypot projects tend to be limited in their ability to easily simulate diverse classes of vulnerabilities, require non-trivial amounts of manual support, or do not scale well enough to meet this challenge.

A fundamental difference between the type of malware captured by traditional honeypots (e.g., Honeyd [23]) and approaches geared towards eliciting payloads from search-based malware stems from how potential victims are targeted. For traditional honeypots, these systems can be deployed at a network telescope [22], for example, and can simply take advantage of the fact that for random scanning malware, any traffic that reaches the telescope is unsolicited and likely malicious in nature. However, search-worms use a technique more akin to instantaneous hit-list automation, thereby only targeting authentic and vulnerable hosts. Were web-based honeypots to mimic the passive approach used for traditional honeypots, they would likely be very ineffective.

To address these limitations, we present a method for crafting dynamic responses to on-line network requests using sample transcripts from observed network interaction. In particular, we provide a flexible technique based on natural language processing and string alignment techniques for intelligently interacting with protocols trained directly from raw traffic. Though our approach is application-agnostic, we demonstrate its util-

ity with a system designed to monitor and capture automated network attacks against vulnerable web applications, without relying on static vulnerability signatures. Specifically, our approach (disguised as a typical web server) elicits interaction with search engines and, in turn, search worms in the hope of capturing their illicit payload. As we show later, our dynamic content generation technique is fairly robust and easy to deploy. Over a 72-day period we were attacked repeatedly, and witnessed more than 368,000 attacks originating from 28,856 distinct IP addresses.

The attacks target a wide range of web applications, many of which attempt to exploit the vulnerable application(s) via a diverse set of injection techniques. To our surprise, even during this short deployment phase, we witnessed several attacks immediately after public disclosure of the vulnerabilities being exploited. That, by itself, validates our technique and underscores both the tenacity of attackers and the overall pervasiveness of web-based exploitation. Moreover, the relentless nature of these attacks certainly sheds light on the scope of this problem, and calls for immediate solutions to better curtail this increasing threat to the security of the Internet. Lastly, our forensic analysis of the captured payloads confirms several earlier findings in the literature, as well as highlights some interesting insights on the post-infection process and the malware themselves.

The rest of the paper is organized as follows. Section 2 discusses related work. We provide a high-level overview of our approach in Section 3, followed by specifics of our generation technique in Section 4. We provide a validation of our approach based on interaction with a rigid binary protocol in Section 5. Additionally, we present our real-world deployment and discuss our findings in Section 6. Finally, we conclude in Section 7.

2 Related Work

Generally speaking, honeypots are deployed with the intention of eliciting interaction from unsuspecting adversaries. The utility in capturing this interaction has been diverse, allowing researchers to discover new patterns and trends in malware propagation [28], generate new signatures for intrusion-detection systems and Internet security software [16, 20, 31], collect malware binaries for static and/or dynamic analysis [21], and quantify malicious behavior through widespread measurement studies [26], to name a few.

The adoption of virtual honeypots by the security community only gained significant traction after the introduction of *low-interaction* honeypots such as *Honeyd* [23]. *Honeyd* is a popular tool for establishing multiple virtual hosts on a single machine. Though *Honeyd* has proved to be fairly useful in practice, it is important to recognize

that its effectiveness is strictly tied to the availability of accurate and representative protocol-emulation scripts, whose generation can be fairly tedious and time consuming. *High-interaction* honeypots use a different approach, replying with authentic and unscripted responses by hosting sand-boxed virtual machines running common software and operating systems [11]¹.

A number of solutions have been proposed to bridge the separation of benefits and restrictions that exist between high and low-interaction honeypots. For example, Leita et al. proposed *ScriptGen* [18, 17], a tool that automatically generates *Honeyd* scripts from network traffic logs. *ScriptGen* creates a finite state machine for each listening port. Unfortunately, as the amount and diversity of available training data grows, so does the size and complexity of its state machines. Similarly, *RolePlayer* (and its successor, *GQ* [10]) generates scripts capable of interacting with live traffic (in particular, worms) by analyzing series of similar application sessions to determine static and dynamic fields and then replay appropriate responses. This is achieved by using a number of heuristics to remove common contextual values from annotated traffic samples and using byte-sequence alignment to find potential session identifiers and length fields.

While neither of these systems specifically target search-based malware, they represent germane approaches and many of the secondary techniques they introduce apply to our design as well. Also, their respective designs illustrate an important observation—the choice between using a small or large set of sample data manifests itself as a system tradeoff: there is little diversity to the requests recognized and responses transmitted by *RolePlayer*, thereby limiting its ability to interact with participants whose behavior deviates from the training session(s). On the other hand, the flexibility provided by greater state coverage in *ScriptGen* comes at a cost to scalability and complexity.

Lastly, since web-based honeypots rely on search engines to index their attack signatures, they are at a disadvantage each time a new attack emerges. In our work, we sidestep the indexing limitations common to static signature web-based honeypots and achieve broad query representation prior to new attacks by proactively generating “signatures” using statistical language models trained on common web-application scripts. When indexed, these signatures allow us to monitor attack behavior conducted by search worms without explicitly deploying structured signatures *a priori*.

3 High-level Overview

We now briefly describe our system architecture. Its setup follows the description depicted in Figure 1, which is conceptually broken into three stages: pre-processing,

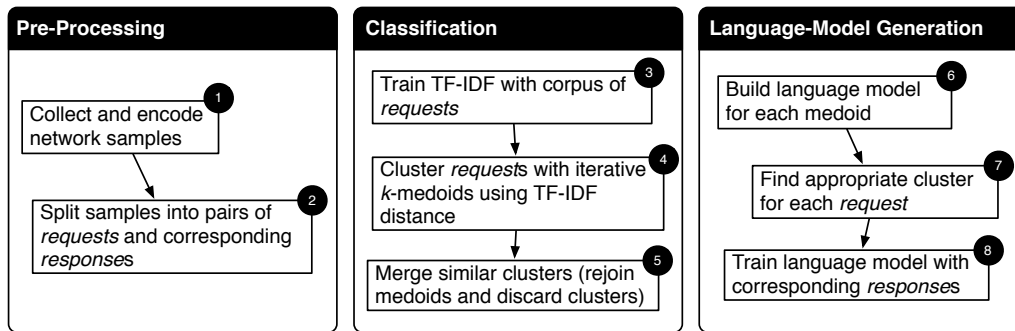


Figure 1: Setup consists of three distinct stages conducted in tandem in preparation for deployment.

classification, and language-model generation. We address each part in turn. We note that although our methodology is not protocol specific, for pedagogical reasons, we provide examples specific to the DNS protocol where appropriate. Our decision to use DNS for validation stems from the fact that validating the correctness of an HTTP response is ill-defined. Likewise, many ASCII-based protocols that come to mind (e.g., HTTP, SMTP, IRC) lack strict notions of correctness and so do not serve as a good conduit to demonstrate the correctness of the output we generate.

To begin, we pre-process and sanitize all trace data used for training. Network traces are stripped of transport protocol headers and organized by session into pairs of requests and responses. Any trace entries that correspond to protocol errors (e.g., HTTP 404) are omitted. Next, we group request and response pairs using a variant of iterative k -means clustering with TF/IDF (i.e., term frequency-inverse document frequency) cosine similarity as our distance metric. Formally, we apply a k -medoids algorithm for clustering, which assigns samples from the data as cluster medoids (i.e., centroids) rather than numerical averages. For reasons that should become clear later, pair similarity is based solely on the content of the request samples. Upon completion, we then generate and train a collection of smoothed n -gram language-models for each cluster. These language-models are subsequently used to produce dynamic responses to online requests. However, because message formats may contain session-specific fields, we also post-process responses to satisfy these dependencies whenever they can be automatically inferred. For example, in DNS, a session identifier uniquely identifies each record request with its response.

During a live deployment, online-classification is used to deduce the response that is most similar to the incoming request (i.e., by mapping the response to its best medoid). For instance, a DNS request for an MX record will ideally match a medoid that maps to other MX re-

quests. The medoid with the minimum TF/IDF distance to an online request identifies which language model is used for generating responses. The language models are built in such a way that they produce responses influenced by the training data. The overall process is depicted in Figure 2. For our evaluation as a web-based honeypot (in Section 6), this process is used in two distinct stages: first, when interacting with search engines for site indexing and second, when courting malware.

4 Under the Hood

In what follows, we now present more specifics about our design and implementation. Recall that our goal is to provide a technique for automatically providing valid responses to protocols interactions learned directly from raw traffic.

In lieu of semantic knowledge, we instead apply classic pattern classification techniques for partitioning a set of observed requests. In particular, we use the iterative k -medoids algorithm. As our distance metric we choose to forgo byte-sequence alignment approaches that have been previously used to classify similarities between protocol messages (e.g. [18, 9, 6]). As Cui et. al. observed, while these approaches are appropriate for classifying requests that only differ parametrically, byte-sequence alignment is ill-suited for classifying messages with different byte-sequences [8]. Therefore, we use TF/IDF cosine similarity as our distance metric.

Intuitively, *term frequency-inverse document frequency* (TF/IDF) is the measure of a term’s significance to a string or document given its significance among a set of documents (or corpus). TF/IDF is often used in information retrieval for a number of applications including automatic text retrieval and approximate string matching [29]. Mathematically, we compute TF/IDF in the following way: let τ_{d_i} denote how often the term τ appears in document d_i such that $d_i \in D$, a collection of documents. Then $\text{TF/IDF} = \text{TF} \cdot \text{IDF}$ where

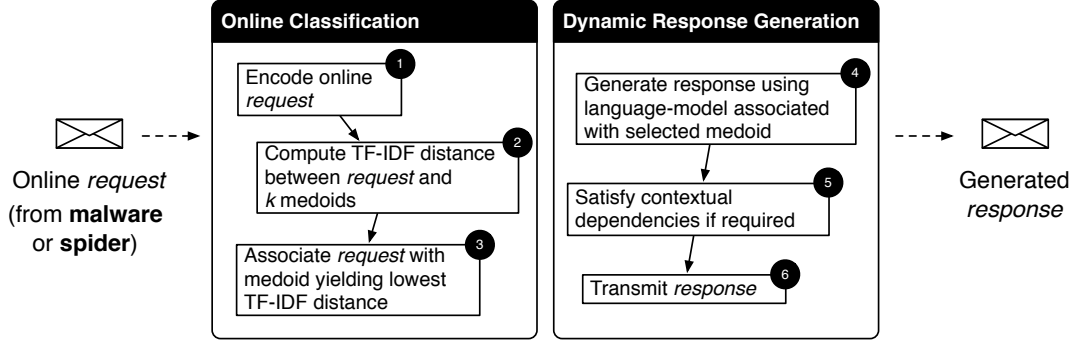


Figure 2: Online classification of requests from malware and search engine spiders influences which language model is selected for response generation.

$$TF = \sqrt{\tau_{d_i}}$$

and

$$IDF = \sqrt{\log \frac{|D|}{|\{d_j : d_j \in D \text{ and } \tau \in d_j\}|}}$$

The term-similarity between two strings from the same corpus can be computed by calculating their TF/IDF *distance*. To do so, both strings are first represented as multi-dimensional vectors. For each term in a string (e.g., word), its TF/IDF value is computed as described previously. Then, for a string with n terms, an n -dimensional vector is formed using these values. The cosine of the angle between two such vectors representing strings indicates a measure of their similarity (hence, its complement is a measure of distance).

In the context of our implementation, terms are delineated by tokenizing requests into the following classes: one or more spaces, one or more printable characters (excluding spaces), and one or more non-printable characters (also excluding spaces).² We chose the space character as a primary term delimiter due to its common occurrence in text-based protocols; however, the delimiter could have easily been chosen automatically by identifying the most frequent byte in all requests. The collection of all requests (and their constituent terms) form the TF/IDF corpus.

Once TF/IDF training is complete we use an iterative k -medoids algorithm, shown in Algorithm 1, to identify similar requests. Upon completion, the classification algorithm produces a k (or less) partitioning over the set of all requests. In an effort to rapidly classify online requests in a memory-efficient manner, we retain only the medoids and dissolve all clusters. For our deployment, we empirically choose $k = 30$, and then perform a trivial cluster-collapsing algorithm: we iterate through the

k clusters and, for each cluster, calculate the mean and standard deviation of the distance between the medoid and the other members of the cluster. Once the k -means and standard deviations are known, we collapse pairs of clusters if the medoid requests are no more than one standard deviation apart.

4.1 Dynamic Response Generation

Since one of the goals of our method is to generate not only valid but also *dynamic* responses to requests, we employ natural language processing techniques (NLP) to create models of protocols. These models, termed *language models*, assign probabilities of occurrence to sequences of tokens based on a corpus of training data. With natural languages such as English we might define a token or, more accurately, a *1-gram* as a string of characters (i.e., a word) delimited by spaces or other punctuation. However, given that we are not working with natural languages, we define a new set of delimiters for protocols. The 1-gram token in our model adheres to one of the following criteria: (1) one or more spaces, (2) one or more printable characters, (3) one or more non-printable characters, or (4) the beginning of message (BOM) or end of message (EOM) tokens.

The training corpora we use contain both requests and responses. Adhering to our assumption that similar requests have similar responses, we train k response language models on the responses associated with each of the k request clusters. That is, each cluster’s response language model is trained on the packets seen in response to the requests in that cluster. Recall that to avoid having to keep every request cluster in memory, we keep only the medoids for each cluster. Then, for each (*request*, *response*) tuple, we recalculate the distance to each of the k request medoids. The medoid with the minimal distance to the tuple’s request identifies which of the k language models is trained using the response. After train-

```

1:  $MedoidSet \leftarrow SelectKRandomElements(ObservedRequests)$ 
2:  $RequestMap < RequestType, MedoidType > \leftarrow \perp$  // for mapping requests to medoids
3: repeat
4:   for all  $R \in (ObservedRequests - MedoidSet)$  do
5:     for all  $M \in MedoidSet$  do
6:        $Distance \leftarrow TF/IDF(R, M)$ 
7:       if  $RequestMap[R] = \perp$  or  $Distance < TF/IDF(RequestMap[R], M)$  then
8:          $RequestMap[R] \leftarrow M$ 
9:     for all  $M \in MedoidSet$  do
10:       $M \leftarrow FindMemberWithLowestMeanDistance(M, RequestMap)$ 
11: until  $HasConverged(MedoidSet)$ 
12: for all  $(M_i, M_j) \in MedoidSet$  s.t.  $i \neq j$  do
13:   if  $TF/IDF(M_i, M_j) < FindThresholdDistance(M_i, M_j)$  then
14:      $MedoidSet \leftarrow MedoidSet - \{M_i, M_j\}$ 
15:      $MedoidSet \leftarrow MedoidSet \cup Merge(M_i, M_j, RequestMap)$ 

```

Algorithm 1: Iterative k -Medoids Classification for Observed Requests

ing concludes, each of the k response language models has a probability of occurrence associated with each observed sequence of 1-grams. A sequence of two 1-grams is called a 2-gram, a sequence of three 1-grams is called a 3-gram, and so on. We cut the maximum n -gram length, n , to eight.

Since it is unlikely that we have witnessed every possible n -gram during training, we use a technique called *smoothing* to lend probability to unobserved sequences. Specifically, we use parametric Witten-Bell back-off smoothing [30], which is the state of the art for n -gram models. This smoothing method estimates, if we consider 3-grams, the 3-gram probability by interpolating between the naive count ratio $C(w_1w_2w_3)/C(w_1w_2)$ and a recursively smoothed probability estimate of the 2-gram probability $P(w_3|w_2)$. The recursively smoothed probabilities are less vulnerable to low counts because of the shorter context. A 2-gram is more likely to occur in the training data than a 3-gram and the trend progresses similarly as the n -gram length decreases. By smoothing, we get a reasonable estimate of the probability of occurrence for all possible n -grams even if we have never seen it during training. Smoothing also mitigates the possibility that certain n -grams dominate in small training corpora. It is important to note that during generation, we only consider the states seen in training.

To perform the response generation, we use the language models to define a Markov model. This Markov model can be thought of as a large finite state machine where each transition occurs based on a *transition probability* rather than an input. As well, each “next state” is conditioned solely on the previous state. The transition probability is derived directly from the language models. The transition probability from a 1-gram, w_1 to a 2-gram, w_1w_2 is $P(w_2|w_1)$, and so on. Intuitively, generation is accomplished by conducting a probabilistic simulation

from the start state (i.e., BOM) to the end state (i.e., EOM).

More specifically, to generate a response, we perform a random walk on the Markov model corresponding to the identified request cluster. From the BOM state, we randomly choose among the possible next states with the probabilities present in the language model. For instance, if the letters ($\mathcal{B}, \mathcal{C}, \mathcal{D}$) can follow \mathcal{A} with probabilities (70%, 20%, 10%) respectively, then we will choose the \mathcal{AB} path approximately 70% of the time and similarly for \mathcal{AC} and \mathcal{AD} . We use this random walk to create responses similar to those seen in training not only in syntax but also in frequency. Ideally, we would produce the same types of responses with the same frequency as those seen during training, but the probabilities used are at the 1-gram level and not the response packet level.

The Markov models used to generate responses attempt to generate valid responses based on the training data. However, because the training is over the entire set of responses corresponding to a cluster, we cannot recognize contextual dependencies between requests and responses. Protocols will often have session identifiers or tokens that necessarily need to be mirrored between request and response. DNS, for instance, has a two byte session identifier in the request that needs to appear in any valid response. As well, the DNS name or IP requested also needs to appear in the response. While the NLP engine will recognize that *some* session identifier and domain name should occupy the correct positions in the response, it is unlikely that the *correct* session identifier and name will be chosen. For this reason, we automatically post-process the NLP generated response to appropriately satisfy contextual dependencies.

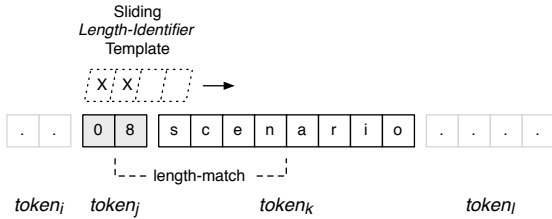


Figure 3: A sliding window template traverses request tokens to identify variable-length tokens that should be reproduced in related responses.

4.1.1 Detecting Contextual Dependencies

Generally speaking, protocols have two classes of contextual dependencies: invariable length tokens and variable length tokens. Invariable length tokens are, as the name implies, tokens that always contain the same number of bytes. For the most part, protocols with variable length tokens typically adhere to one of two standards: tokens preceded by a length field and tokens separated using a special byte delimiter. Overwhelmingly, protocols use length-preceded tokens (DNS, Samba, Netbios, NFS, etc.). The other less-common type (as in HTTP) employ variable length delimited tokens.

Our method for handling each of these token types differs only slightly from techniques employed by other active responder and protocol disassembly techniques ([8, 9]). Specifically, we identify contextual dependencies using two techniques. First, we apply the Needleman-Wunsch string alignment algorithm [19] to align requests with their associated responses during training. Since the language models we use are not well suited for this particular task, this process is used to identify if, and where, substrings from a request also appear in its response. If certain bytes or sequences of bytes match over an empirically derived threshold (80% in our case), these bytes are considered invariable length tokens and the byte positions are copied from request to response after the NLP generation phase.

To identify variable length tokens, we make the simplifying assumption that these types of tokens are preceded by a length identifier; we do so primarily because we are unaware of any protocols that contain contextual dependencies between request and response through character-delimited variable length tokens. As depicted in Figure 3, we iterate over each request and consider each set of up to four bytes as a length identifier if and only if the token that follows it belongs to a certain character class³ for the described length. In our example, $Token_i$ is identified as a candidate length-field based upon its value. Since the next immediate token is of the

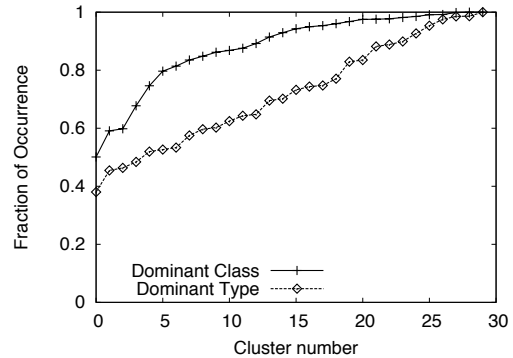


Figure 4: Frequency of dominant type/class per request cluster (with $k = 30$), sorted from least to most accurate.

length described by $Token_i$ (i.e., 8), $Token_j$ is identified as a variable length token. For each variable length token discovered, we search for the same token in the observed response. We copy these tokens after NLP generation if and only if this matching behavior was common to more than half of the request and response pairs observed throughout training.

As an aside, the content-length header field in our HTTP responses also needs to accurately reflect the number of bytes contained in each response. If the value of this field is greater than the number of bytes in a response, the recipient will poll for more data, causing transactions to stall indefinitely. Similarly, if the value of the content-length field is less than the number of bytes in the response, the recipient will prematurely halt and truncate additional data. While other approaches have been suggested for automatically inferring fields of this type, we simply post-process the generated HTTP response and automatically set the content-length value to be the number of bytes after the end-of-header character.

5 Validation

In order to assess the correctness of our dynamic response generation techniques, we validate our overall approach in the context of DNS. Again, we reiterate that our choice for using DNS in this case is because it is a rigid binary protocol, and if we can correctly generate dynamic responses for this protocol, we believe it aptly demonstrates the strength (and soundness) of our approach. For our subsequent evaluation, we train our DNS responder off a week’s worth of raw network traces collected from a public wireless network used by approximately 50 clients. The traffic was automatically par-

tioned into request and response tuples as outlined in Section 4.

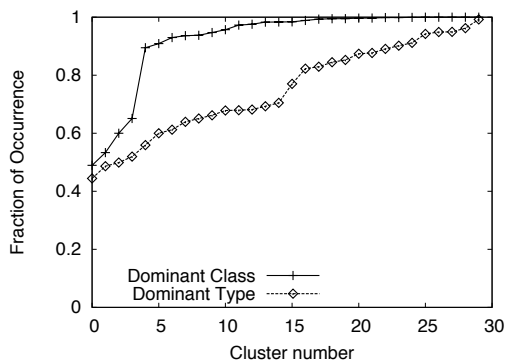


Figure 5: Frequency of dominant type/class per *response* cluster (with $k = 30$), sorted from least to most accurate.

To validate the output of our clustering technique, we consider clustering of requests successful if for each cluster, one type of request (A, MX, NS, etc.) and the class of the request (IN) emerges as the most dominant member of the cluster; a cluster with one type and one class appearing more frequently than any other is likely to correctly classify an incoming request and, in turn, generate a response to the correct query type and class. We report results based on using 10,000 randomly selected flows for training. As Figures 4 and 5 show, nearly all clusters have a dominating type and class.

To demonstrate our response generation’s success rate, we performed 20,000 DNS requests on randomly generated domain names (of varying length). We used the UNIX command `host`⁴ to request several types of records. For validation purposes, we consider a response as strictly faithful if it is correctly interpreted by the requesting program with no warnings or errors. Likewise, we consider a response as valid if it processes correctly with or without warnings or errors. The results are shown in Figure 6 for various training flow sizes. Notice that we achieve a high success rate with as little as 5,000 flows, with correctness ranging between 89% and 92% for strictly faithful responses, and over 98% accuracy in the case of valid responses.

In summary, this demonstrates that the overall design depicted in Figures 1 and 2—that embodies our training phase, classification phase, model generation, and reposing phase to detect contextual dependencies and correctly mirror the representative tokens in their correct location(s)—produces faithful responses. More importantly, these responses are learned automatically, and re-

quire little or no manual intervention. In what follows, we further substantiate the utility of our approach in the context of a web-based honeypot.

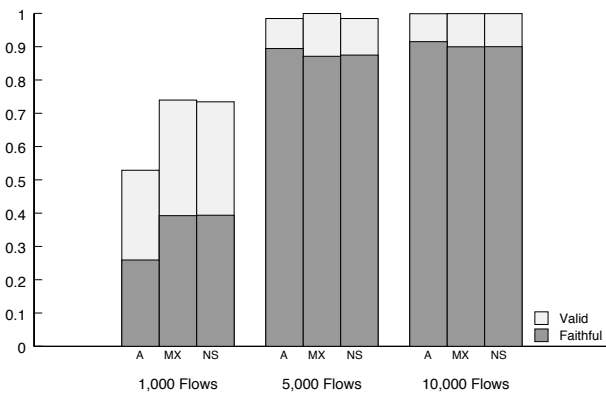


Figure 6: Faithful, valid, or erroneous response success rate for 20,000 random DNS requests under different numbers of training flows.

6 Evaluation

Our earlier assertion was that the exploitation of web-apps now pose a serious threat to the Internet. In order to gauge the extent to which this is true, we used our dynamic generation techniques to build a lightweight HTTP responder — in the hope of snatching attack traffic targeted at web applications. These attackers query popular search engines for strings that fingerprint the vulnerable software and isolate their targets.

| Type | Appearances |
|--------|-------------|
| .PHP | 3165 |
| .PL | 29 |
| .CGI | 49 |
| .HTML | 15 |
| .PHTML | 2 |

Table 1: Query Types

With this in mind, we obtained a list of the 3,285 of the most searched queries on Google by known botnets attempting to exploit web applications.⁵ We then queried Google for the top 20 results associated with each query. Although there are several bot queries that are ambiguous and are most likely not targeting a specific web application, most of the queries were targeted. However, automatically determining the number of different web applications being attacked is infeasible, if not impossible. For this reason, we provide only the break down in

the types of web applications being exploited (i.e., PHP, Perl, CGI, etc.) in Table 1.

Nearly all of the bots searching for these queries exploit command injection vulnerabilities. The PHP vulnerabilities are most commonly exploited through remote inclusion of a PHP script, while the Perl vulnerabilities, are usually exploited with UNIX delimiters and commands. Since CGI/HTML/PHTML can house programs from many different types of underlying languages, they encompass a wide range of exploitation techniques. The collected data contains raw traces of the interactions seen when downloading the pages for each of the returned results. Our corpus contained 178,541 TCP flows, of which we randomly selected 24,000 flows as training data for our real-world deployment (see Section 6.1).

Since our primary goal here is to detect (and catch) bots using search engines to query strings present in vulnerable web applications, our responder must be in a position to capture these prey — i.e., it has to be broadly indexed by multiple search engines. To do so, we first created links to our responder from popular pages,⁶ and then expedited the indexing process by disclosing the existence of a minor bug in a common UNIX application to the Full-Disclosure mailing list. The bug we disclosed cannot be leveraged for privilege escalation. Bulletins from Full-Disclosure are mirrored on several high-ranking websites and are crawled extensively by search-engine spiders; less than a few hours later, our site appeared in search results on two prominent search engines. And, right on queue, the attacks immediately followed.

6.1 Real-World Deployment

For our real-world evaluation, we deployed our system on a 3.0 GHz dual-processor Intel Xeon with 8 GB of RAM. At runtime, memory utilization peaked at 960 MB of RAM when trained with 24,000 flows. CPU utilization remained at negligible levels throughout operation and on average, requests are satisfied in less than a second. Because our design was optimized to purposely keep all data RAM during runtime, disk access was unnecessary.

Shortly after becoming indexed, search-worms began to attack at an alarming rate, with the attacks rapidly increasing over a two month deployment period. During that time, we also recorded the number of indexes returned by Google per day (which totaled just shy of 12,000 during the deployment). We choose to only show PHP attacks because of their prominence. Figure 7 depicts the number of attacks we observed per day. For reference, we provide annotations of our Google index count in ten day intervals until the indices plateau.

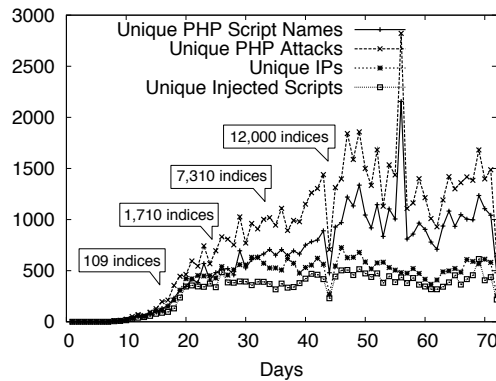


Figure 7: Daily PHP attacks. The valley on day 44 is due to an 8 hr power outage. The peak on day 56 is because two bots launched over 2,000 unique script attacks.

For ease of exposition, we categorize the observed attacks into four groups. The first denotes the number of attacks targeting vulnerabilities that have distinct file structures in their names. The class “Unique PHP attacks”, however, is more refined and represents the number of attacks against scripts but using unique injection variables (i.e., `index.php?page=` and `index.php?inc=`). The reason we do so is that the file names and structures can be ubiquitous and so by including the variable names we glean insights into attacks against potentially distinct vulnerabilities. We also attempt to quantify the number of distinct botnets involved in these attacks. While many botnets attack the same application vulnerabilities, (presumably) these botnets can be differentiated by the PHP script(s) they remotely include. Recall that a typical PHP remote-include exploit is of the form “`vulnerable.php?variable=http://site.com/attack_script?`”, and in practice, botnets tend to use disjoint sites to store attack scripts. Therefore, we associate bots with a particular botnet by identifying unique injection script repositories. Based on this admittedly loose notion of uniqueness [27], we observed attacks from 5,648 distinct botnets. Lastly, we record the number of unique IP addresses that attempt to compromise our responder.

The results are shown in Figure 7. An immediate observation is the sheer volume of attacks—in total, well over 368,000 attacks targeting just under 45,000 unique scripts before we shutdown the responder. Interestingly, notice that there are more unique PHP attacks than unique IPs, suggesting that unlike traditional scanning attacks, these bots query for and attack a wide variety of web applications. Moreover, while many bots attempt

to exploit a large number of vulnerabilities, the repositories hosting the injected scripts remain unchanged from attack to attack. The range of attacks is perhaps better demonstrated not by the number of unique PHP scripts attacked but by the number of unique PHP web-applications that are the target of these attacks.

6.1.1 Unique WebApps

In general, classifying the number of unique web applications being attacked is difficult because some bots target PHP scripts whose filenames are ubiquitous (e.g., `index.php`). In these cases, bots are either targeting a vulnerability in one specific web-application that happens to use a common filename or arbitrarily attempting to include remote PHP scripts.

To determine if an attack can be linked to a specific web-application, we downloaded the directory structures for over 4,000 web-applications from SourceForge.net. From these directory structures, we matched the web application to the corresponding attacked script (e.g., `gallery.php` might appear only in the Web Gallery web application). Next, we associated an attack with a specific web application if the file name appeared in no more than 10 web-app file structures. We choose a threshold of 10 since SourceForge stores several copies of essentially the same web application under different names (due to, for instance, “skin” changes or different code maintainers). For non-experimental deployments aimed at detecting zero-day attacks, training data could be associated with its application of origin, thereby making associations between non-generic attacks and specific web-applications straightforward.

Based on this heuristic, we are able to map the 24,000 flows we initially trained on to 560 “unique” web-applications. Said another way, by simply building our language models on randomly chosen flows, we were able to generate content that approximates 560 distinct web-applications — a feat that is not as easy to achieve if we were to deploy each application on a typical web-based honeypot (e.g., the Google Hack Honeypot [3]). The attacks themselves were linked back to 295 distinct web applications, which is indicative of the diversity of attacks.

We note that our heuristic to map content to web-apps is strictly a lower bound as it only identifies web-applications that have a distinct directory structure and/or file name; a large percentage of web-applications use `index.php` and other ubiquitous names and are therefore not accounted for. Nonetheless, we believe this serves to make the point that our approach is effective and easily deployable, and moreover, provides insight into the amount of web-application vulnerabilities currently being leveraged by botnets.

6.1.2 Spotting Emergent Threats

While the original intention of our deployment was to elicit interaction from malware exploiting known vulnerabilities in web applications, we became indexed under broader conditions due to the high amount of variability in our training data. As a result, a honeypot or active responder indexed under such a broad set of web applications can, in fact, attract attacks targeting *unknown* vulnerabilities. For instance, according to `milw0rm` (a popular security advisory/exploit distribution site), over 65 PHP remote inclusion vulnerabilities were released during our two month deployment [1]. Our deployment began on October 27th, 2007 and used the *same* training data for its entire duration. Hence, any attack exploiting a vulnerability released after October 27th is an attack we did not explicitly set out to detect.

Nonetheless, we witnessed several emergent threats (some may even consider them “zero-day” attacks) because some of the original queries used to bootstrap training were generic and happened to represent a wide number of webapps. As of this writing, we have identified more than 10 attacks against vulnerabilities that were undisclosed at deployment time (some examples are illustrated in Table 2). It is unlikely that we witnessed these attacks simply because of arbitrary attempts to exploit random websites—indeed, we never witnessed many of the other disclosed vulnerabilities being attacked.

We argue that given the frequency with which these types of vulnerabilities are released, a honeypot or an active responder without dynamic content generation will likely miss an overwhelming amount of attack traffic—in the attacks we witnessed, botnets begin attacking vulnerable applications on the day the vulnerability was publicly disclosed! An even more compelling case for our architecture is embodied by attacks against vulnerabilities that have not been disclosed (e.g., the recent WordPress vulnerability [7]). We believe that the potential to identify these attacks exemplifies the real promise of our approach.

6.2 Dissecting the Captured Payloads

To better understand what the post-infection process entails, we conducted a rudimentary analysis of the remotely included PHP scripts. Our malware analysis was performed on a Linux based Intel virtual machine with the 2.4.7 kernel. We used a deprecated kernel version since newer versions do not export the system call table of which we take advantage. Our environment consisted of a kernel module and a preloaded library⁷ that serve to inoculate malware before execution and to log interesting behavior. The preloaded library captures calls

| Disclosure Date | Attack Date | Signature |
|-----------------|-------------|--|
| 2007-11-04 | 2007-11-10 | /starnet/themes/c-sky/main.inc.php?cmsdir= |
| 2007-11-21 | 2007-11-23 | /comments-display-tpl.php?language_file= |
| 2007-11-22 | 2007-11-22 | /admin/kfm/initialise.php?kfm_base_path= |
| 2007-11-25 | 2007-11-25 | /Commence/includes/db.connect.php?phproot_path= |
| 2007-11-28 | 2007-11-28 | /decoder/gallery.php?ccms_library_path= |

Table 2: Attacks targeting vulnerabilities that were unknown at time of deployment

to `connect()` and `send()`. The `connect` hook deceives the malware by faking successful connections, and the `send` function allows us to record information transmitted over sockets.⁸

Our kernel module hooks three system calls: (`open`, `write`, and `execve`). We execute every script under a predefined user ID, and interactions under this ID are recorded via the `open()` hook. We also disallow calls to `open` that request write access to a file, but feign success by returning a special file descriptor. Attempts to write to this file descriptor are logged via `syslog`. Doing so allows us to record files written by the malware without allowing it to actually modify the file system. Similarly, only commands whose file names contain a pre-defined random password are allowed to execute. All other command executions under the user ID fail to execute (but pretend to succeed), assuring no malicious commands execute. Returning success from failed executions is important because a script may, for example, check if a command (e.g., `wget`) successfully executes before requesting the target URL.

To determine the functionality of the individual malware scripts, we batched processed all the captured malware on the aforementioned architecture. From the transcripts provided by the kernel module and library, we were able to discern basic functionality, such as whether or not the script makes connections, issues IRC commands, attempts to write files, etc. In certain cases, we also conducted more in-depth analyses by hand to uncover seemingly more complex functionality. We discuss our findings in more detail below.

The high-level break-down for the observed scripts is given in Table 3. The challenge in capturing bot payloads in web application attacks stems from the ease with which the attacker can test for a vulnerability; unique string displays (where the malware echoes a unique token in the response to signify successful exploitation) accounts for the most prevalent type of injection. Typically, bots parse returned responses for their identifying token and, if found, proceed to inject the actual bot payload. Since these unique tokens are unlikely to appear in our generated response, we augment our responder to echo these tokens at run-time. While the use of random numbers as tokens seem to be the *soup du jour* for testing

| Script Classification | Instances |
|-----------------------|-----------|
| PHP Web-based Shells | 834 |
| Echo Notification | 591 |
| PHP Bots | 377 |
| Spammers | 347 |
| Downloaders | 182 |
| Perl Bots | 136 |
| Email Notification | 87 |
| Text Injection | 35 |
| Java-script Injection | 18 |
| Information Farming | 9 |
| Uploaders | 4 |
| Image Injection | 4 |
| UDP Flooders | 3 |

Table 3: Observed instances of individual malware

a vulnerability, we observed several instances where attackers injected an image. Somewhat comically, in many cases, the bot simply e-mails the IP address of the vulnerable machine, which the attacker then attempts to exploit at a later time. The least common vulnerability test we observed used a connect-back operation to connect to an attacker-controlled system and send vulnerability information to the attacker. This information is presumably logged server-side for later use.

Interestingly, we notice that bots will often inject simple text files that typically also contain a unique identifying string. Because PHP scripts can be embedded inside HTML, PHP requires begin and end markers. When a text file is injected without these markers, its contents are simply interpreted as HTML and displayed in the output. This by itself is not particularly interesting, but we observed several attackers injecting large lists of queries to find vulnerable web applications via search engines. The largest query list we captured contained 7,890 search queries that appear to identify vulnerable web applications — all of which could be used to bootstrap our content generation further and cast an even wider net.

Overall, the collected malware was surprisingly modular and offered diverse functionality similar to that reported elsewhere [26, 15, 13, 12, 25, 5, 4]. The captured scripts (mostly PHP-based command shells), are advanced enough that many have the ability to display

the output in some user-friendly graphical user interface, obfuscate the script itself, clean the logs, erase the script and related evidence, deface a site, crawl vulnerability sites, perform distributed denial of service attacks and even perform automatic self-updates. In some cases, the malware inserted tracking cookies and/or attempted to gain more information about a system's inner-workings (e.g., by copying `/etc/passwd` and performing local banner scans). To our surprise, only eight scripts contained functionality to automatically obtain `root`. In these cases, they all used C-based kernel vulnerabilities that write to the disk and compile upon exploitation. Lastly, IRC was used almost exclusively as the communication medium. As can be expected, we also observed several instances of spamming malware using e-mail addresses pulled from the web-application's MySQL database backend. In a system like phpBB, this can be highly effective because most forum users enter an e-mail address during the registration process. Cross-checking the bot IPs with data from the Spamhaus project [2] shows that roughly 36% of them currently appear in the spam black list.

One noteworthy functionality that seems to transcend our categorizations among PHP scripts is the ability to break out of PHP safe mode. PHP safe mode disables functionality for, among others, executing system commands, modifying the file system, etc. The malware we observed that bypass safe mode tend to contain a handful of known exploits that either exploit functionality in PHP, functionality in mysql, or functionality in web server software. Lastly, we note that although we observed what appeared to be over 5,648 unique injection scripts from distinct botnets, nearly half of them point to zombie botnets. These botnets no longer have a centralized control mechanism and the remotely included scripts are no longer accessible. However, they are still responsible for an overwhelming amount of our observed HTTP traffic.

6.3 Limitations

One might argue that a considerably less complex (but more mundane) approach for eliciting search worm traffic may be to generate large static pages that contain content representative of a variety of popular web-applications. However, simply returning arbitrary or static pages does not yield either the volume or diversity of attacks we observed. For instance, one of our departmental websites (with a much higher PageRank than our deployment site) only witnessed 437 similar attacks since August 2006. As we showed in Section 6, we witnessed well over 368,000 attacks in just over two months. Moreover, close inspection of the attacks on the university website show that they are far less varied or

interesting. These attacks seem to originate from either a few botnets that issue "loose" search queries (e.g., "in-url:index.php") and subsequently inject their attack, or simply attack ubiquitous file names with common variable names. Not surprisingly, these unsophisticated botnets are less widespread, most likely because they fail to infect many hosts. By contrast, the success of our approach lead to more insightful observations about the scope and diversity of attacks because we were able to cast a far wider net.

That said, for real-world honeypot deployments, detection and exploitation of the honeypot itself can be a concern. Clearly, our system is not a true web-server and like other honeypots [23], it too can be trivially detected using various fingerprinting techniques [14]. More to the point, a well-crafted bot that knows that a particular string always appears in pages returned by a given web-application could simply request the page from us and check for the presence of that string. Since we will likely fail to produce that string, our phony will be detected⁹.

The fact that our web-honeypot can be detected is a clear limitation of our approach, but in practice it has not hindered our efforts to characterize current attack trends, for several reasons. First, the search worms we witnessed all seemed to use search engines to find the identifying information of a web-application, and attacked the vulnerability upon the first visit to the site; presumably because verifying that the response contains the expected string slows down infection. Moreover, it is often times difficult to discern the web-application of origin as many web-applications do not necessarily contain strings that uniquely identify the software. Indeed, in our own analysis, we often had difficulty identifying the targeted web-application by hand, and so automating this might not be trivial.

Lastly, we argue that the limitations of the approach proposed herein manifests themselves as trade-offs. Our decision to design a stateless system results in a memory-efficient and lightweight deployment. However, this design choice also makes handling stateful protocols nearly impossible. It is conceivable that one can convert our architecture to better interact with stateful protocols by simply changing some aspects of the design. For instance, this could be accomplished by incorporating flow sequence information into training and then recalling its hierarchy during generation (e.g., by generating a response from the set of appropriate first round responses, then second round responses, etc.). To capture multi-stage attacks, however, *ScriptGen* [18, 17] may be a better choice for emulating multi-stage protocol interaction, and can be used in conjunction with our technique to cast a wider net to initially entice such malware.

7 Conclusion

In this paper, we use a number of multi-disciplinary techniques to generate dynamic responses to protocol interactions. We demonstrate the utility of our approach through the deployment of a dynamic content generation system targeted at eliciting attacks against web-based exploits. During a two month period we witnessed an unrelenting barrage of attacks from attackers that scour search engine results to find victims (in this case, vulnerable web applications). The attacks were targeted at a diverse set of web applications, and employed a myriad of injection techniques. We believe that the results herein provide valuable insights on the nature and scope of this increasing Internet threat.

8 Acknowledgments

We thank Ryan MacArthur for his assistance with the forensic analysis presented in Section 6.2. We are grateful to him for the time spent analyzing and cataloging the captured payloads. We also thank David Dagon for cross-referencing bot IP addresses with the Spamhaus Project black lists [2]. We extend our gratitude to our shepherd, George Danezis, and the anonymous reviewers for their invaluable comments and suggestions. We also thank Bryan Hoffman, Charles Wright, Greg MacManus, Moheeb Abu Rajab, Lucas Ballard, and Scott Coull for many insightful discussions during the course of this project. This work was funded in part by NFS grants CT-0627476 and CNS-0546350.

Data Availability

To promote further research and awareness of the malware problem, the data gathered during our live deployment is available to the research community. For information on how to get access to this data, please see http://spar.isi.jhu.edu/botnet_data/.

References

- [1] Milw0rm. See <http://www.milw0rm.com/>.
- [2] The Spamhaus Project. See <http://www.spamhaus.org/>.
- [3] The Google Hack Honeygot, 2005. See <http://ghh.sourceforge.net/>.
- [4] ANDERSON, D. S., FLEIZACH, C., SAVAGE, S., AND VOELKER, G. M. Spamscluster: Characterizing internet scam hosting infrastructure. In *Proceedings of the 16th USENIX Security Symposium*, pp. 135–148.
- [5] BARFORD, P., AND YEGNESWARAN, V. An inside look at botnets. In *Advances in Information Security* (2007), vol. 27, Springer Verlag, pp. 171–191.
- [6] BEDDOE, M. The protocol informatics project, 2004.
- [7] CHEUNG, A. Secunia’s wordpress GBK/Big5 character set ”S” SQL injection advisory. See <http://secunia.com/advisories/28005/>.
- [8] CUI, W., KANNAN, J., AND WANG, H. J. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of the 16th USENIX Security Symposium* (Boston, MA, August 2007), pp. 199–212.
- [9] CUI, W., PAXSON, V., WEAVER, N., AND KATZ, R. H. Protocol-independent adaptive replay of application dialog. In *Network and Distributed System Security Symposium 2006* (February 2006), Internet Society.
- [10] CUI, W., PAXSON, V., AND WEAVER, N. C. GQ: Realizing a system to catch worms in a quarter million places. Tech. Rep. TR-06-004, International Computer Science Institute, 2006.
- [11] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating systems design and implementation* (New York, NY, USA, 2002), ACM Press, pp. 211–224.
- [12] FREILING, F. C., HOLZ, T., AND WICHERSKI, G. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)* (September 2005), vol. 3679 of *Lecture Notes in Computer Science*, pp. 319–335.
- [13] GU, G., PORRAS, P., YEGNESWARAN, V., FONG, M., AND LEE, W. BotHunter: Detecting malware infection through IDS-driven dialog correlation. In *Proceedings of the 16th USENIX Security Symposium* (August 2007), pp. 167–182.
- [14] HOLZ, T., AND RAYNAL, F. Detecting honeypots and other suspicious environments. In *Proceedings of the Workshop on Information Assurance and Security* (June 2005).

- [15] Know your enemy: Tracking botnets. Tech. rep., The HoneyNet Project and Research Alliance, March 2005. Available from <http://www.honeynet.org/papers/bots/>.
- [16] KREIBICH, C., AND CROWCROFT, J. Honeycomb - Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (Hotnets II)* (Boston, November 2003).
- [17] LEITA, C., DACIER, M., AND MASSICOTTE, F. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *RAID* (2006), D. Zamboni and C. Krügel, Eds., vol. 4219 of *Lecture Notes in Computer Science*, Springer, pp. 185–205.
- [18] LEITA, C., MERMOUD, K., AND DACIER, M. ScriptGen: an automated script generation tool for honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference* (December 2005), pp. 203–214.
- [19] NEEDLEMAN, S. B., AND WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48 (1970), 443–453.
- [20] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 226–241.
- [21] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium* (2005).
- [22] PANG, R., YEGNESWARAN, V., BARFORD, P., PAXSON, V., AND PETERSON, L. Characteristics of Internet background radiation, October 2004.
- [23] PROVOS, N. A virtual honeypot framework. In *Proceedings of the 12th USENIX Security Symposium* (August 2004), pp. 1–14.
- [24] PROVOS, N., MCCLAIN, J., AND WANG, K. Search worms. In *Proceedings of the 4th ACM workshop on Recurring malware* (New York, NY, USA, 2006), ACM, pp. 1–8.
- [25] PROVOS, N., MCNAMEE, D., MAVROMMATIS, P., WANG, K., AND MODADUGU, N. The ghost in the browser: Analysis of web-based malware. In *Usenix Workshop on Hot Topics in Botnets (HotBots)* (2007).
- [26] RAJAB, M. A., ZARFOSS, J., MONROSE, F., AND TERZIS, A. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of ACM SIGCOMM/USENIX Internet Measurement Conference* (October 2006), ACM, pp. 41–52.
- [27] RAJAB, M. A., ZARFOSS, J., MONROSE, F., AND TERZIS, A. My botnet is bigger than yours (maybe, better than yours): Why size estimates remain challenging. In *Proceedings of the first USENIX workshop on hot topics in Botnets (HotBots '07)*. (April 2007).
- [28] RIDEN, J., MCGEEHAN, R., ENGERT, B., AND MUETER, M. Know your enemy: Web application threats, February 2007.
- [29] TATA, S., AND PATEL, J. Estimating the selectivity of TF-IDF based cosine similarity predicates. *SIGMOD Record* 36, 2 (June 2007).
- [30] WITTEN, I. H., AND BELL, T. C. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory* 37, 4 (1991), 1085–1094.
- [31] YEGNESWARAN, V., GIFFIN, J. T., BARFORD, P., AND JHA, S. An Architecture for Generating Semantics-Aware Signatures. In *Proceedings of the 14th USENIX Security Symposium* (Baltimore, MD, USA, Aug. 2005), pp. 97–112.

Notes

¹ The drawback, of course, is that high-interaction honeypots are a heavy-weight solution, and risk creating their own security problems [23].

² Protocol messages are tokenized similarly in [18, 17] and [8].

³ In practice, we use printable and non-printable.

⁴ The results are virtually the same for `nslookup`, and hence, omitted.

⁵ These initial queries were provided by one of the authors, but similar results could easily be achieved by crawling the WebApp directories in SourceForge and searching Google for identifiable strings (similar to what we outline in Section 6.1.1).

⁶ We placed links on 3 pages with Google PageRank ranking of 6, 2 pages with rank 5, 3 pages with rank 2, and 5 pages with rank 0.

⁷ A preloaded library loads before all other libraries in order to hook certain library functions

⁸ Because none of the malware we obtained use direct system calls to either `connect()` or `send()`, this setup suffices for our needs.

⁹ Notice however that if a botnet has n bots conducting an attack against a particular web-application, we only need to probabilistically return what the malware is seeking $1/n^{th}$ of the time to capture the malicious payload.