Botnet Judo: Fighting Spam with Itself

Andreas Pitsillidis* Kirill Levchenko* Christian Kreibich† Chris Kanich*
Geoffrey M. Voelker* Vern Paxson† Nicholas Weaver† Stefan Savage*

†International Computer Science Institute
Berkeley, USA
christian@icir.org
{vern,nweaver}@cs.berkeley.edu

*Dept. of Computer Science and Engineering
University of California, San Diego, USA
{apitsill,klevchen,ckanich}@cs.ucsd.edu
{voelker,savage}@cs.ucsd.edu

Abstract

We have traditionally viewed spam from the receiver's point of view: mail servers assaulted by a barrage of spam from which we must pick out a handful of legitimate messages. In this paper we describe a system for better filtering spam by exploiting the vantage point of the spammer. By instantiating and monitoring botnet hosts in a controlled environment, we are able to monitor new spam as it is created, and consequently infer the underlying template *used to generate polymorphic e-mail messages. We demonstrate this approach on mail traces from a range of modern botnets and show that we can automatically filter such spam precisely and with virtually no false positives.*

1. Introduction

Reactive defenses, in virtually any domain, depend on the currency of their intelligence. How much do you know about your opponent's next move and how quickly can you act on it? Maximizing the time advantage of such information is what drives governments to launch spy satellites and professional sports teams to employ lip readers. By the same token, a broad range of competitors, from commodities traders to on-line gamers, all seek to exploit small time advantages to deliver large overall gains. As Benjamin Franklin famously wrote, "Time is money."

Today's spammers operate within this same regime. Receivers install filters to block spam. Spammers in turn modify their messages to evade these filters for a time, until the receiver can react to the change and create a new filter rule in turn. This pattern, common to spam, anti-virus and intrusion detection alike, dictates that the window of vulnerability for new spam depends on how quickly it takes the receiver to adapt.

We advocate shrinking this time window by changing the vantage point from which we fight spam. In particular, it is widely documented that all but a small fraction of today's spam e-mail is transmitted by just a handful of distributed botnets [10, 18], and these, in turn, use template-based macro languages to specify how individual e-mail messages should be generated [14, 27]. Since these templates describe *precisely* the range of polymorphism that a spam campaign is using at a given point in time, a filter derived from these templates has the potential to identify such e-mails *perfectly* (i.e., never generating false positives or false negatives).

In this paper we describe *Judo*, a system that realizes just such an approach—quickly producing precise mail filters essentially equivalent the very templates being used to create the spam. However, while spam templates can be taken directly from the underlying botnet "command and control" channel, this approach can require significant manual effort to reverse engineer each unique protocol [12, 14]. Instead, we use a *black-box* approach, in which individual botnet instances are executed in a controlled environment and the spam messages they attempt to send are analyzed online. We show that using only this stream of messages we can quickly and efficiently produce a comprehensive regular expression that captures all messages generated from a template while avoiding extraneous matches. For example, in tests against live botnet spam, we find that by examining roughly 1,000 samples from a botnet (under a minute for a *single* energetic bot, and still less time if multiple bots are monitored) we can infer precise filters that produce zero false positives against non-spam mail, while matching virtually all subsequent spam based on the same template. While template inference is by no means foolproof (and accordingly we examine the anticipated arms race), we believe that this approach, like Bayesian filtering, IP blacklisting and sender reputation systems, offers a significant new tool

for combating spam while being cheap and practical to deploy within existing anti-spam infrastructure.

In particular, we believe our work offers three contributions: First, we produce a general approach for inferring e-mail generation templates in their entirety, subsuming prior work focused on particular features (e.g., mail header anomalies, subject lines, URLs). Second, we describe a concrete algorithm that can perform this inference task in near real-time (only a few seconds to generate an initial high-quality regular expression, and fractions of a second to update and refine it in response to subsequent samples), thereby making this approach feasible for deployment. Finally, we test this approach empirically against live botnet spam, demonstrate its effectiveness, and identify the requirements for practical use.

The remainder of the paper is organized as follows. In Section 2 we provide background and related work followed by a description of our model for spam generation in Section 3. We describe the Judo system in Section 4 and then evaluate its effectiveness on live data in Section 5. We discuss our results and further applications in Section 6 followed by our overall conclusions.

2. Background and Related Work

Since the first reported complaint in 1978, unsolicited bulk e-mail, or *spam*, has caused untold grief among users, system administrators, and researchers alike. As spam grew, so too did anti-spam efforts—today comprising a commercial market with over $1B in annual revenue. In this section we briefly review existing anti-spam technology and discuss the botnet work most directly related to this paper.

2.1. Current Anti-spam Approaches

Broadly speaking, anti-spam technologies deployed today fall into two categories: content-based and sender-based. Content-based approaches are the oldest and perhaps best known, focusing on filtering unwanted e-mail based on features of the message body and headers that are either anomalous (e.g., date is in the future) or consistent with undesirable messages (e.g., includes words like Rolex or Viagra). Early systems were simple heuristics configured manually, but these evolved into systems based on supervised learning approaches that use labeled examples of spam and non-spam to train a classifier using well-known techniques such as Bayesian inference [22, 25] and Support Vector Machines [6, 32]. These techniques can be highly effective (see Cormack and Lynam for an empirical comparison [5]) but are also subject to adversarial chaff and poisoning attacks [8, 17], and require great care to avoid false positives as spammers become more sophisticated at disguising their mail as legitimate.

Another class of content-based filtering approaches involves blacklisting the URLs advertised in spam [1, 3, 4, 31]. Because URL signatures are simple and require no complex training, they are more easily integrated into a closed-loop system: for example, in one study of spam sent by the Storm botnet, domains observed in templates were on average subsequently found on a URL blacklist only 18 minutes afterwards [15]. Unfortunately, URL-based systems also require comprehensive, up-to-date whitelists to avoid poisoning. They are also generally rigid in blocking *all* appearances of the URL regardless of context, and, of course, they do nothing for spam not containing a URL (e.g., stock schemes, image-based spam, and some types of phishing). Our system provides a fast, closed-loop response, while generating a more selective signature based on the URL (if present) and text of the spam instances.

Sender-based systems focus on the means by which spam is delivered. The assumption is that any Internet address that sends unsolicited messages is highly likely to repeat this act, and unlikely to send legitimate, desired communication. Thus, using a range of spam oracles, ranging from e-mail honeypots to user complaints, these systems track the IP addresses of Internet hosts being used to send spam. Individual mail servers can then validate incoming e-mail by querying the database (typically via DNS) to see if the transmitting host is a known spam source [11, 20]. Blacklists dealt very effectively with open e-mail relays and proxies, and forced spammers to move to botnet-based spam distribution, in which many thousands of compromised hosts under central control relay or generate spam on behalf of a single spammer [23]. As the number of hosts grows, this both reduces blacklist freshness and places scalability burdens on blacklist operators.

A related approach is sender reputation filtering, conceptually related to Internet address blacklisting. These schemes attempt to provide stronger ties between the nominal and true sender of e-mail messages in order to allow records of individual domains' communications to be employed to filter spam based on past behavior. Thus, using authentication systems such as SPF [30] or DomainKeys [16] (or heuristics such as greylisting [9]), a mapping can be made between origin e-mail's domain (e.g., foo.com) and the mail servers authorized to send on behalf of these addresses. Having bound these together, mail receivers can then track the reputation for each sending domain (i.e., how much legitimate mail and how much spam each sends) and build filtering policies accordingly [28].

In practice, these techniques are used in combination, with their precise formulation and mixture tuned to new spam trends and "outbreaks" (e.g., image spam). We view our system as a new component in this arsenal.

2.2. Spamming Botnets

Since roughly 2004, bot-based spam distribution has emerged as the platform of choice for large-scale spam campaigns. Conceptually, spam botnets are quite simple—the spammer generates spam and then arranges to send it through thousands of compromised hosts, thereby laundering any singular origin that could be blacklisted. However, an additional complexity is that spammers also need to generate content that is sufficiently polymorphic so that at least some of it will evade existing content filters. To describe this polymorphism, while ensuring that the underlying "messaging" is consistent, spammers have developed template-based systems. While original template-based spam generation from such templates was centralized, modern spammers now broadcast templates to individual bot hosts that in turn generate and send distinct message instances.

We have previously described the template-based spamming engine of the Storm botnet [14], while Stern analyzed that of the Srizbi botnet [27] and first observed the opportunity for filtering techniques that "exploit the regularity of template-generated messages." Our work is a practical realization of this insight.

Closest to our own system is the Botlab system of John *et al.* [10]. Their system, contemporaneously built, also executes bots in a virtual machine environment and extracts the outbound spam e-mail messages. Indeed, our work builds on their preliminary successes (and their data, which the authors have also graciously shared), which included using exact-matching of witnessed URL strings as a filter for future botnet spam. Our work is a generalization in that we do not assume the existence of any particular static feature (URL or otherwise), but focus on inferring the underlying template used by each botnet and from this generating comprehensive and precise regular expressions.

The system of Göbel *et al.* [7] uses a similar approach. Their system generates signatures by analyzing spam messages collected from compromised hosts as well. However, the Judo template inference algorithm supports key additional elements, such as *dictionaries*, which make it significantly more expressive.

The AutoRE system of Xie *et al.* clusters spam messages into campaigns using heuristics about how embedded URLs are obfuscated [31]. This effort has algorithmic similarities to our work, as it too generates regular expressions over spam strings, but focuses on a single feature of spam e-mail (URLs). By contrast to these efforts, our work is distinguished both by its generality (for example, generating signatures for images spam or spam with "tinyurl" links for which URLs are not discriminatory) and by its design for on-line real-time use.

3. Template-based Spam

Spam templates are akin to form letters, consisting of text interspersed with substitution macros that are instantiated differently in each generated message. Unlike form letters, spam templates use macros more aggressively, not only to personalize each message but also to avoid spam filtering based on message content. Figure 1 shows a typical template from the Storm botnet (circa Feb. 2008) together with an instance of spam created from the template and a Judo regular expression signature generated from 1,000 message instances (Section 4 below describes our algorithm for generating such signatures). Note that this regular expression was created by observing the messages sent by the botnet, and *without any prior knowledge of the underlying template* (we show the template for reference only).

Template Elements. We can divide the bulk of macros used in templates into two types: *noise* macros that produce a sequence of random characters from a specified character alphabet, and *dictionary* macros that choose a random phrase from a list given in a separate "dictionary" included with the template. For example, in the Storm template shown in Figure 1, the "`%^P ... ^%`" macro generates a string of a given length using a given set of characters, while the "`%^F ... ^%`" macro inserts a string from a list in the specified dictionary. Similar functionality exists in other template languages (e.g., the $rndabc$ and $rndsyn$ macros in Reactor Mailer [27]). In general, noise macros are used to randomize strings for which there are few semantic rules (e.g., message-ids), while dictionary macros are used for content that must be semantically appropriate in context (e.g., subject lines, sender e-mail addresses, etc). In addition to these two classes, there are also special macros for dates, sender IP addresses, and the like. We deal with such macros specially (Section 4.2).

Thus, to a first approximation, our model assumes that a message generated from a template will consist of three types of strings: invariant text, *noise* macros producing random characters as described above, and *dictionary* macros producing a text fragment from a fixed list (the dictionary).

Real-time Filtering. The nature of template systems documented by our earlier work [14] as well as Stern [27] suggests that templates can be used to *identify*—and thus filter—any mail instances generated from a template. It is relatively straightforward to convert a Storm template, for example, into a regular expression by converting macros to corresponding regular expressions: noise macros become repeated character classes, and dictionary macros become a disjunction of the dictionary elements. Such a regular expression signature will then match all spam generated from the template. Unfortunately, obtaining these templates requires reverse-engineering the botnet "command and control" channel—a highly time-consuming task. Instead, our

```
Received: from %^C0%^P%^R2-6^%:qwertyuiopasdfghjklzxcvbnm^%.%^P%^R2-6^%:qwertyuiopasd ▷
          fghjklzxcvbnm^%^% ([%^C6%^I^%.%^I^%.%^I^%.%^I^%^%]) by ▷
          %^A^% with Microsoft SMTPSVC(%^Fsvcver^%); %^D^%
Message-ID: <%^Z^%.%^R1-9^%0%^R0-9^%0%^R0-9^%0%^R0-9^%@%^C1%^Fdomains^%^%>
Date: %^D^%
From: <%^Fnames^%@%^V1^%>
User-Agent: Thunderbird %^Ftrunver^%
MIME-Version: 1.0
To: %^0^%
Subject: %^Fstormline^%
Content-Type: text/plain; charset=ISO-8859-1; format=flowed
Content-Transfer-Encoding: 7bit

%^G%^Fstormline^% http://%^Fstormlink2^%/^%
```

```
Received: from auz.xwzww ([132.233.197.74]) by dsl-189-188-79-63.prod-infinitum.com.mx ▷
          with Microsoft SMTPSVC(5.0.2195.6713); Wed, 6 Feb 2008 16:33:44 -0800
Message-ID: <id012345.99066044044@experimentalist.org>
Date: Wed, 6 Feb 2008 16:33:44 -0800
From: <katiera@experimentalist.org>
User-Agent: Thunderbird 2.0.0.14 (Windows/20080421)
MIME-Version: 1.0
To: victim@spam-target.com
Subject: Get Facebook's FBI Files
Content-Type: text/plain; charset=ISO-8859-1; format=flowed
Content-Transfer-Encoding: 7bit

FBI may strike Facebook http://GoodNewsGames.com/
```

```
From: <.+@.+\..+>
User-Agent: Thunderbird 2\.0\.0\.14 \(Windows/200(80421\)|70728\))
MIME-Version: 1\.0
To: .+@.+\..+
Subject: (Get Facebook's FBI Files|...|The F\.B\.I\. has a new way of tracking Facebook)
Content-Transfer-Encoding: 7bit

(FBI may strike Facebook|FBI wants instant access to Facebook|...|The F\.B\.I\. ▷
has a new way of tracking Facebook) http://(GoodNewsGames|...|StockLowNews)\.com/
```

Figure 1. Fragment of a template from the Storm template corpus, together with a typical instantiation, and the regular expression produced by the template inference algorithm from 1,000 instances. The subject line and body were captured as dictionaries (complete dictionaries omitted to save space). This signature was generated without any prior knowledge of the underlying template.

template inference algorithm, described in the next section, creates such signatures by observing multiple instances of the spam from the same template.

Figure 2 shows a diagram of a Judo spam filtering system. The system consists of three components: a "bot farm" running instances of spamming botnets in a contained environment; the signature generator; and the spam filter. The system intercepts all spam sent by bots and provides the specimens to a signature generator. The signature generator maintains a set of regular expression signatures for spam sent by each botnet, updating the set in real time if necessary. We can then immediately disseminate new signatures to spam filters.

System Assumptions. Our proposed spam filtering system operates on a number of assumptions. First and foremost, of course, we assume that bots compose spam using a

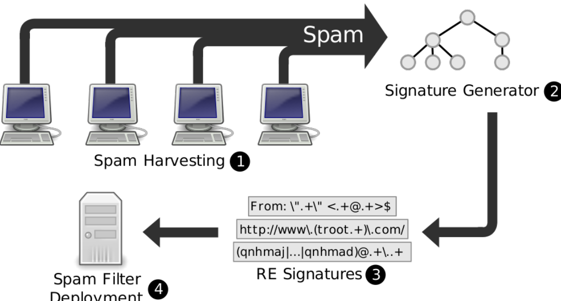

Figure 2. Automatic template inference makes it possible to deploy template signatures as soon as they appear "in the wild:" bots (❶) running in a contained environment generate spam processed by the Judo system (❷); signatures (❸) are generated in real time and disseminated to mail filtering appliances (❹).

template system as described above. We also rely on spam campaigns employing a small number of templates at any point in time. Thus, we can use templates inferred from one or a few bot instances to produce filters effective at matching the spam being sent by all other bot instances. This assumption appears to hold for the Storm botnet [15]; in Section 5.3 we empirically validate this assumption for other botnets. Finally, as a simplification, our system assumes that the first few messages of a new template do not appear intermixed with messages from other new templates. Since we infer template content from spam output, rather than extracting it directly from botnet command and control messages, interleaved messages from several new templates will cause us to infer an amalgamated template—the product of multiple templates—which is consequently less precise than ideal. This assumption could be relaxed by more sophisticated spam pre-clustering, but we do not explore doing so in this paper.

4. The Signature Generator

In this section we describe the Judo system, which processes spam generated by bots and produces regular expression signatures. The system operates in real time, updating the set of signatures immediately in response to new messages. We begin with a description of the template inference algorithm—the heart of Judo—which, given a set of messages, generates a matching regular expression signature. We then describe how we incorporate domain knowledge, such as the header-body structure of e-mail messages, into the basic algorithm. Finally, we show how we use the template inference algorithm to maintain a *set* of signatures matching all messages seen up to a given point in time.

4.1. Template Inference

The template inference algorithm produces a regular expression signature from a set of messages assumed to be generated from the same spam template. As described in Section 3, a template consists of invariant text and macros of two types: *noise* macros which generate a sequence of random characters, and *dictionary* macros which choose a phrase at random from a list (the dictionary). Proceeding from this model, the algorithm identifies these elements in the text of the input messages, and then generates a matching regular expression. Throughout this section, we use the simple example in Figure 3 to illustrate the steps of our algorithm.

4.1.1. Anchors

The first step of the algorithm is to identify invariant text in a template, that is, fragments of text that appear in every message. We call such fragments *anchors* (because they "anchor" macro-generated portions of a message). Invariant text like "Best prices!" and "http://" in Figure 3 are examples of anchors.

We start by extracting the longest ordered set of substrings having length at least q that are common to every message. Parameter q determines the minimum length of an anchor. Setting $q = 1$ would simply produce the longest common subsequence of the input messages, which, in addition to the anchors, would contain common characters, such as spaces, which do not serve as useful anchors. Large minimum anchor lengths, on the other hand, may exclude some useful short anchors. In our experiments with a variety of spamming botnets, we found that $q = 6$ produces good results, with slightly smaller or larger values also working well.

We note that finding this ordered set of substrings is equivalent to computing the longest common subsequence (LCS) over the q-gram sequence of the input, i.e., the sequence of substrings obtained by sliding a length-q window over each message. Unfortunately, the classic dynamic programming LCS algorithm is quadratic in the length of its inputs.

As an optimization we first identify all substrings of length at least q that are common to all input messages (using a suffix tree constructed for these messages). We then find longest common subsequence of substrings (i.e., treating each substring as a single "character") using the classic dynamic programming algorithm. Typical messages thus collapse from several thousand bytes to fewer than ten common substrings, resulting in essentially linear-time input processing.

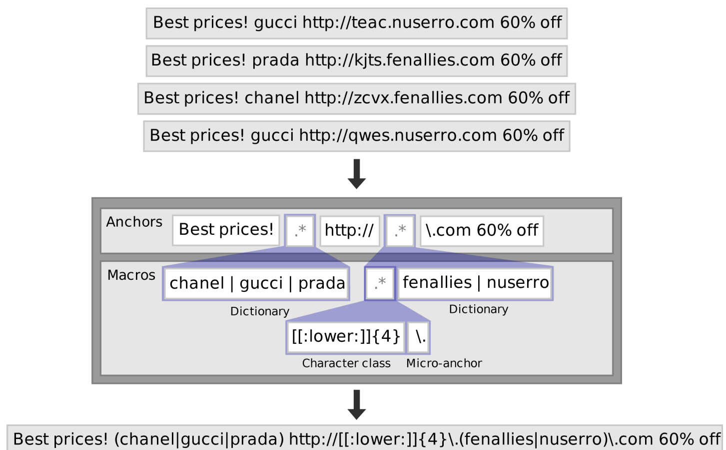

Figure 3. Template inference algorithm example showing excerpts from template-based spam messages, the invariant text and macros inferred from the spam, and the resulting regular expression signature.

4.1.2. Macros

Following anchor identification, the next step is to classify the variant text found *between* anchors. In the simplest case this text corresponds to a single macro, but it may also be the back-to-back concatenation of several macros. In general, our algorithm does not distinguish between a single macro and the concatenation of two or more macros, unless the macros are joined by "special" characters — non-alphanumeric printing characters such as "@" and punctuation. We call these special characters *micro-anchors* and treat them as normal anchors. Our next step, then, is to decide whether text between a pair of anchors is a dictionary macro, a noise macro, or a micro-anchor expression, itself consisting of a combination of dictionary and noise macros separated by micro-anchor characters.

Dictionary Macros. In a template, a dictionary macro is instantiated by choosing a random string from a list. In Figure 3, for instance, the brand names "gucci", "prada", and "chanel" would be generated using such a macro. Given a set of strings found between a pair of anchors, we start by determining whether we should represent these instances using a dictionary macro or a noise macro. Formally speaking, every macro can be represented as a dictionary, in the sense that it is a set of strings. However, if we have not seen every possible instance of a dictionary, a dictionary representation will be necessarily incomplete, leading to false positives. Thus we would like to determine whether what we have observed is the *entirety* of a dictionary or not. We formulate the problem as a hypothesis test: the null hypothesis is that there is an unobserved dictionary element. We take the probability of such an unobserved element to be at least the empirical probability of the least-frequent observed element. Formally, let n be the number of distinct strings in m samples, and let f_n be the empirical probability of the least-frequent element (i.e., the number of times it occurs in the sample, divided by m). Then the probability of observing fewer than n distinct strings in m samples drawn from a dictionary containing $n + 1$ elements is at most $(1 - f_n/(1 + f_n))^m$. For the brand strings in Figure 3, this value is at most $(1 - 0.25/1.25)^4 \approx 0.41$. In practice, we use a 99% confidence threshold; for the sake of example, however, we will assume the confidence threshold is much lower. If the dictionary representation is chosen, we group the distinct strings $\alpha_1, \ldots, \alpha_n$ into a disjunction regular expression $(\alpha_1 \mid \ldots \mid \alpha_n)$ to match the variant text. Otherwise, we check whether it might be a micro-anchor expression.

Micro-Anchors. A micro-anchor is a substring that consists of non-alphanumeric printing characters too short to be a full anchor. Intuitively, such strings are more likely to delimit macros than ordinary alphanumeric characters, and are thus allowed to be much shorter than the minimum anchor length q. We again use the LCS algorithm to identify micro-anchors, but allow only non-alphanumeric printing characters to match. In Figure 3, the domain names in the URLs are split into smaller substrings around the "." micro-anchor. Once micro-anchors partition the text, the algorithm performs the dictionary test on each set of strings delimited by the micro-anchors. Failing this, we represent these strings as a noise macro.

Noise Macros. If a set of strings between a pair of anchors or micro-anchors fails the dictionary test, we consider those strings to be generated by a noise macro (a macro that generates random characters from some character set). In Figure 3, the host names in the URLs fail the dictionary test and are treated as a noise macro. The algorithm chooses the smallest character set that matches the data from the set of

POSIX character classes `[:alnum:]`, `[:alpha:]`, etc., or a combination thereof. If all strings have the same length, the character class expression repeats as many times as the length of the string, i.e., the regular expression matches on both character class and length. Otherwise, we allow arbitrary repetition using the "$*$" or "$+$" operators.[1]

When generating each signature, we also add the constraint that it must contain at least one anchor or dictionary node. If this constraint is violated, we consider the signature as *unsafe* and discard it.

4.2. Leveraging Domain Knowledge

As designed, the template inference algorithm works on arbitrary text. By exploiting the structure and semantics of e-mail messages, however, we can "condition" the input to greatly improve the performance of the algorithm. We do two kinds of "conditioning," as described next.

Header Filtering. The most important pre-processing element of the Judo system concerns headers. We ignore all but the following headers: "MIME-Version," "Mail-Followup-To," "Mail-Reply-To," "User-Agent," "X-MSMail-Priority," "X-Priority," "References," "Language," "Content-Language," "Content-Transfer-Encoding," and "Subject." We specifically exclude headers typically added by a mail transfer agent. This is to avoid including elements of the spam collection environment, such as the IP address of the mail server, in signatures. We also exclude "To" and "From" headers; if the botnet uses a list of e-mail addresses in alphabetical order to instantiate the "To" and "From" headers, portions of the e-mail address may be incorrectly identified as anchors.

The resulting headers are then processed individually by running the template inference algorithm on each header separately. A message must match all headers for a signature to be considered a match.

Special Tokens. In addition to dictionary and noise macros, bots use a small class of macros for non-random variable text. These macros generate dates, IP addresses, and the like. If the output of a date macro, for example, were run through the template inference algorithm, it would infer that the year, month, day, and possibly hour are anchors, and the minutes and seconds are macros. The resulting signature would, in effect, "expire" shortly after it was generated. To cope with this class of macros, we perform the following pre- and post-processing steps. On input, we replace certain well-known tokens (currently: dates, IP addresses, and multi-part message delimiters) with special fixed strings that the template inference algorithm treats as anchors. After the algorithm produces a regular expression signature, it replaces these fixed strings with regular expressions that capture all instances of the macro.

4.3. Signature Update

The Judo system processes e-mail messages generated by a botnet, creating a set of signatures that match those messages. The need for a signature *set*, rather than a single signature, arises because several templates may be in use at the same time. Recall that the template inference algorithm relies heavily on anchors (common text) to locate macros. If the template inference algorithm were given messages generated from different templates, only strings common to all templates would be identified as anchors, leading the algorithm to produce a signature that is too general. Ideally, then, we would like to maintain one signature per template.

Unfortunately, because we do not know which template was used to generate a given message, we cannot simply group messages by template and apply the template inference algorithm separately to each. The situation is not as dire as it seems, however. If we already *have* a good signature for a template, we can, by definition, easily identify messages generated by the template. Thus, if new templates are deployed incrementally, we can use the template inference algorithm only on those messages which do not already match an existing signature.

On receiving a new message, the algorithm first checks if the message matches any of its existing signatures for the botnet in question. If it does, it ignores the message, as there is already a signature for it. Otherwise, it places the message into a *training buffer*. When the training buffer fills up, it sends the message to the template inference algorithm to produce a new signature. The size of the training buffer is controlled by a parameter k, which determines the trade-off between signature selectivity and training time. If the training buffer is too small, some dictionaries may be incomplete—the template inference algorithm will emit a noise macro instead. On the other hand, a very large training buffer means waiting longer for a usable signature. A very large training buffer increases the chances of mixing messages from two different templates, decreasing signature accuracy. Thus we would like to use a training buffer as small as necessary to generate good signatures.

In experimenting with the signature generator, we found that no single value of k gave satisfactory results. The Storm botnet, for example, tended to use large dictionaries requiring many message instances to classify, while in the extreme case of a completely invariant template (containing no macros), signature generation would be delayed unnecessarily, even though the first few messages are sufficient to produce a perfect signature. We developed two additional mechanisms to handle such extreme cases more gracefully: the *second chance mechanism* and *pre-clustering*.

Second Chance Mechanism. In many cases, a good sig-

[1] We also experimented with the alternative of allowing a range of lengths, but found such an approach too restrictive in practice.

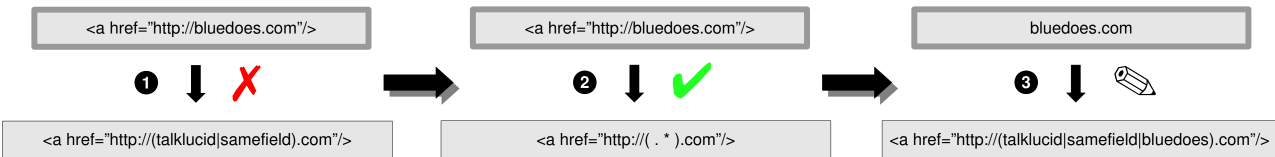

Figure 4. The second chance mechanism allows the updating of signatures: when a new message fails to match an existing signature (❶), it is checked again only against the anchor nodes (❷); if a match is found, the signature is updated accordingly (❸).

nature can be produced from a small number of messages, even though many more are required to fully capture dictionaries. Furthermore, the dictionary statistical test is rather conservative (to avoid false negatives). To combine the advantage of fast signature deployment with the eventual benefits of dictionaries, we developed a "second chance" mechanism allowing a signature to be updated after it has been produced. When a new message fails to match an existing signature, we check if it would match any existing signatures consisting of anchors only. Such *anchor signatures* are simply ordinary regular expression signatures (called *full signatures*) with the macro-based portions replaced by the ".$*$" regular expression. If the match succeeds, the message is added to the training buffer of the signature and the signature is updated. This update is performed incrementally without needing to rerun a new instance of the inference algorithm.

Pre-Clustering. Whereas the second chance mechanism helps mitigate the effects of a small training buffer, pre-clustering helps mitigate the effects of a large training buffer. Specifically, a large training buffer may intermix messages from different templates, resulting in an amalgamated signature. With pre-clustering, unclassified messages are clustered using *skeleton signatures*. A skeleton signature is akin to an anchor signature used in the second chance mechanism, but is built with a larger minimum anchor size q, and as such is more permissive. In our experiments, we set $q = 6$ for anchor signatures and $q = 14$ for skeleton signatures. Further evaluation indicated that slight variations of these values only have a minimal impact on the overall performance.

The pre-clustering mechanism works as follows. Each skeleton signature has an associated training buffer. When a message fails to match a full signature or an anchor signature (per the second chance mechanism), we attempt to assign it to a training buffer using a skeleton signature. Failing that, it is added to the unclassified message buffer. When this buffer has sufficient samples (we use 10), we generate a skeleton regular expression from them and assign them to the skeleton's training buffer. When a training buffer reaches k messages ($k = 100$ works well), a full signature is generated. The signature and its training buffer are moved to the signature set, and the signature is ready for use in a production anti-spam appliance. In effect, the pre-clustering mechanism mirrors the basic signature update procedure (with skeleton signatures instead of full and anchor signatures).

As noted in Section 3, our system does not currently support a complete mechanism for pre-clustering messages into different campaigns. Instead, our current mechanism relies on the earlier assumption that the first few messages of a new template do not appear intermixed with messages from other new templates—hence our decision to group together every 10 new unclassified messages and treat them as a new cluster. Note that it is perfectly acceptable if these first 10 messages from a new template are interleaved with the messages of a template for which we already have generated a signature. In this case, the messages of the latter will be filtered out using the existing regular expression, and only the messages of the former will enter the unclassified message buffer. From our experience, this choice has provided us with very good results, although a more sophisticated clustering method could be a possible future direction.

4.4. Execution Time

Currently the execution time of the template inference algorithm observed empirically is linear in the size of the input. Based on our experience, the length of messages generated by different botnets varies significantly. The largest average length we have observed among all botnets was close to 6,000 characters. The selected training buffer size k (introduced in Section 4.3), along with the average length of e-mails, determine the total size of the input. Under this worst-case scenario, the algorithm requires 2 seconds for $k = 50$ and 10 seconds for $k = 500$, on a modest desktop system. Signature updates execute much faster, as they are performed incrementally. The execution time in this case depends on a wide range of factors, but an average estimate is between $50 - 100$ ms. The focus of our implementation has always been accuracy rather than execution time, thus we expect several optimizations to be possible.

Corpus	Messages
SpamAssassin 2003 [19]	4,150
TREC 2007[29] (non-spam only)	25,220
lists.gnu.org [2] (20 active lists)	479,413
Enron [13]	517,431

Table 1. Legitimate mail corpora used to assess signature safety throughout the evaluation.

5. Evaluation

The principal requirements of a spam filtering system are that it should be both *safe* and *effective*, meaning that it does not classify legitimate mail as spam, and it correctly recognizes the targeted class of spam. Our goal is to experimentally demonstrate that Judo is indeed safe and effective for filtering botnet-originated spam.

Our evaluation consists of three sets of experiments. In the first, we establish the effectiveness of the template inference algorithm on spam generated synthetically from actual templates used by the Storm botnet. Next, we run the Judo system on *actual* spam sent by four different bots, measuring its effectiveness against spam generated by the same bot. In our last set of experiments, we execute a real deployment scenario, training and testing on *different* instances of the same bot. In all cases, Judo was able to generate effective signatures.

In each set of experiments, we also assess the safety of the Judo system. Because Judo signatures are so specific, they are, by design, extremely safe; signatures generated in most of our experiments generated no false positives. We start by describing our methodology for evaluating signature safety.

5.1. Signature Safety Testing Methodology

By their nature, Judo signatures are highly specific, targeting a single observed campaign. As such, we expect them to be extremely *safe*: Judo signatures should never match legitimate mail. We consider this to be Judo's most compelling feature.

The accepted metric for safety is the *false positive rate* of a signature with respect to a corpus of legitimate (non-spam) mail, i.e., the proportion of legitimate messages incorrectly identified as spam. Throughout the evaluation we report the false positive rate of the generated signatures; Section 5.5 presents a more detailed analysis.

Corpora. We used four corpora of legitimate mail, together totaling over a million messages, summarized in Table 1: the SpamAssassin "ham" corpus dated February 2003 [19], the 2007 TREC Public Spam Corpus restricted to messages labelled non-spam [29], 20 active mailing lists from lists.gnu.org spanning August 2000 to April 2009 [2], and the Enron corpus [13].

Age bias. Recall that Judo signatures consist of regular expressions for a message's header as well as the body. To avoid potential age bias, we tested our signatures with all but the subject and body regular expressions removed. This is to safeguard against age-sensitive headers like "User-Agent" causing matches to fail on the older corpora. It is worth noting that using only subject and body is a significant handicap because the remaining headers can act as additional highly discriminating features.

5.2. Single Template Inference

The template inference algorithm is the heart of the Judo system. We begin by evaluating this component in a straightforward experiment, running the template inference algorithm directly on training sets generated from single templates. By varying the size of the training set, we can empirically determine how much spam is necessary to achieve a desired level of signature effectiveness. Our metric of effectiveness is the *false negative rate* with respect to instances of spam generated from the same template. In other words, the false negative rate is the proportion of spam test messages that do not match the signature. Because the template is known, we can also (informally) compare it with the generated signature. Figure 1 from Section 3 shows an example.

5.2.1. Methodology

We generated spam from real templates and dictionaries, collected during our 2008 study of Storm botnet campaign orchestration [15]. The templates covered three campaigns: a self-propagation campaign from August 1–2, 2008 (4,210 templates, 1,018 unique), a pharmaceutical campaign from the same time period (4,994 templates, 1,271 unique), and several low-priced stock campaigns between June and August 2008 (1,472 templates, all unique). Each one of these templates had its own unique set of dictionary files. Both the self-propagation and pharmaceutical templates contained URLs; the stock campaign templates did not.[2]

For convenience, we generated Storm spam directly from these templates (rather than having to operate actual Storm bots) by implementing a Storm template instantiation tool based on our earlier reverse-engineering work on this botnet [14]. For each of the 10,676 templates, we generated

[2] Although less common in terms of Storm's overall spam volume [15], we included non-URL spam to determine how much of our system's effectiveness stems from learning the domains of URLs appearing in the spam, compared to other features.

k	False Negative Rate 95%	99%	Max	Avg
1000	0%	0%	0%	0%
500	0%	0%	2.53%	<0.01%
100	0%	0%	0%	0%
50	0%	0%	19.15%	0.06%
10	45.45%	58.77%	81.03%	14.16%

(a) Self-propagation and pharmaceutical spam.

k	s	False Negative Rate 95%	99%	Max	Avg
1000	99.8%	0%	0.22%	100%	0.21%
500	81.8%	100%	100%	100%	18.21%
100	55.0%	100%	100%	100%	45.04%
50	42.9%	100%	100%	100%	57.25%
10	40.9%	100%	100%	100%	62.13%

(b) Stock spam.

Table 2. False negative rates for spam generated from Storm templates as a function of the training buffer size k. Rows report statistics over templates. The stock spam table also shows the number of templates s for which a signature was generated (for self-propagation and pharmaceutical templates, a signature was generated for every template); in cases where a signature was not generated, every instance in the test set was counted as a false negative. At $k = 1000$, the false positive rate for all signatures was zero.

1,000 training instances and an additional 4,000 instances for testing.

We ran the template inference algorithm on the 1,000 training messages and assessed the false negative rate of the resulting signature using the 4,000-message test set.[3] To better understand the performance of the Judo system, we then pushed it to the "breaking point" by using smaller and smaller training sets to generate a signature. We use k to denote the training set size.

5.2.2. Results

As expected, the template inference algorithm generated effective signatures. Both self-propagation and pharmaceutical campaigns were captured perfectly, with no false negatives. For the stock campaign, 99% of the templates had a false negative rate of 0.22% or lower.

Table 2 also shows Judo's performance as we decrease the number of training instances k. In effect, k is a measure of how "hard" a template is. We separate results for templates with URLs (self-propagation and pharmaceutical) and without (stock) to establish that our algorithm is effective for both types of spam. Rows correspond to different numbers of training messages, and columns to summary statistics of the range of the false negative rates. For example, when training a regular expression on just $k = 10$ messages from a URL-based campaign template, the signature yielded a false negative rate of 45.45% or less on 95% of such templates, and a false negative rate of 81.03% for the template that produced the worst false negative rate. Such high false negative rates are not surprising given just 10 training instances; with just 50 training messages, it exhibits no false negatives for 99% of such Storm templates.[4]

Column s in Table 2b also shows the number of templates for which a signature was generated (all templates resulted in a signature for self-propagation and pharmaceutical spam). Recall from Section 4 that we discard a signature if it is found to be unsafe. This was the only case in our evaluation where this occurred. For such templates for which we do not generate a signature, we calculate a 100% false negative rate. For this test, our input was stock templates which did not contain URLs. These messages were also very short and consisted entirely of large dictionaries: characteristics that make messages particularly difficult to characterize automatically from a small number of samples.

Signatures from the self-propagation and pharmaceutical templates produced no false positives in three of the four legitimate mail corpora, regardless of the value of k. For the stock templates, the result was again zero for $k = 1000$. We present a detailed breakdown of these results in Section 5.5.

[3] We choose 1,000 as the training set size in part because Storm generated roughly 1,000 messages for each requested work unit. We note that 1,000 messages represents a very small training set compared with the amount of spam generated by bots from a given template: in our 2008 study of spam conversion rates [12] we observed almost 350 million spam messages for one spam campaign generated from just 9 templates. Thus, we can generate potent signatures nearly immediately after a new template is deployed, and use that signature for the duration of a large spam campaign.

[4] One peculiarity in Table 2a merits discussion: the 2.53% maximum false negative rate for $k = 500$ arises due to a single template out of the 9,204 total; every other template had a false negative rate of 0%. For this template, when transitioning from $k = 100$ to $k = 500$ training samples the algorithm converted the "Subject" header from a character class to a dictionary. The mismatches all come about from a single dictionary entry missing from the signature because it did not appear in the 500 training messages.

Botnet	k \ d	0	50	100	500	Sigs
		Cumulative False Negative Rate				
Mega-D	50	0.11%	0.09%	0.07%	0.05%	5
	100	0.16%	0.13%	0.12%	0.08%	5
	500	0.54%	0.52%	0.50%	0.34%	5
Pushdo	50	0.17%	0.13%	0.10%	0.05%	8
	100	0.23%	0.20%	0.17%	0.08%	6
	500	0.72%	0.69%	0.66%	0.45%	6
Rustock	50	0.19%	0.12%	0.06%	0.05%	9
	100	0.28%	0.22%	0.15%	0.08%	9
	500	1.01%	0.95%	0.88%	0.40%	9
Srizbi	50	0.22%	0.11%	0%	0%	11
	100	0.33%	0.22%	0.11%	0%	11
	500	1.21%	1.10%	1.05%	0.79%	11

Table 3. Cumulative false negative rate as a function of training buffer size k and classification delay d for spam generated by a single bot instance. The "Sigs" column shows the number of signatures generated during the experiment (500,000 training and 500,000 testing messages). All signatures produced zero false positives with the only exception being the signatures for Rustock.

5.3. Multiple Template Inference

In the previous section we examined the case of spam generated using a single template. In practice, a bot may be switching between multiple templates without any indication. In this part of the evaluation we test the algorithm on a "live" stream of messages generated by a single bot, classifying each message as it is produced.

5.3.1. Methodology

Our spam corpus consists of bot-generated spam collected by the Botlab [10] project from the University of Washington. One instance each of the Mega-D, Pushdo, Rustock, and Srizbi bots was executed and their output collected. We split the first 1 million messages from each bot into a training and testing set by sequentially assigning messages to each set in alternation. The Judo system was then used to create signatures from the training data. In parallel with running the Judo system, we processed the testing corpus in chronological order, classifying each message using signatures generated up to that point. In other words, we consider a test message *matched* (a *true positive*) if it matches some signature generated chronologically *before* the test message; otherwise we count it as a *false negative*. Our measure of effectiveness is the false negative rate over the testing message set.

It is important to note that in practice one could employ some *delay* when matching messages against the filters: either holding up messages for a short period to wait for the generation of updated filters, or by retroactively testing messages already accepted, but not yet presented to the receiving user, against any filter updates. To simulate such a scenario, we buffered testing messages in a *classification buffer*, allowing us to delay classification. We denote the length of the classification buffer by d. The case $d = 0$ corresponds to no message buffering; in other words, messages must be classified immediately upon being received. We also evaluated signature performance with the classification delay d set to 50, 100 and 500. In a real-world deployment, we can think of this buffer as a very short delay introduced by e-mail providers before delivering incoming e-mails to inboxes.

5.3.2. Results

Our results confirm the effectiveness of the Judo system in this "live" setting as well. Table 3 shows the cumulative results for each combination of k and d, as well as the number of signatures generated for each botnet during the experiment. Two trends are evident. First, the false negative rate decreases as the classification delay d increases. This is not surprising, since the delay gives Judo time to build a signature. The second trend, an *increasing* false negative rate as k increases, may seem counterintuitive because in our previous experiment, increasing k led to a decrease in the false negative rate. This increase occurs because all spam in the testing set generated before Judo produces the signature is

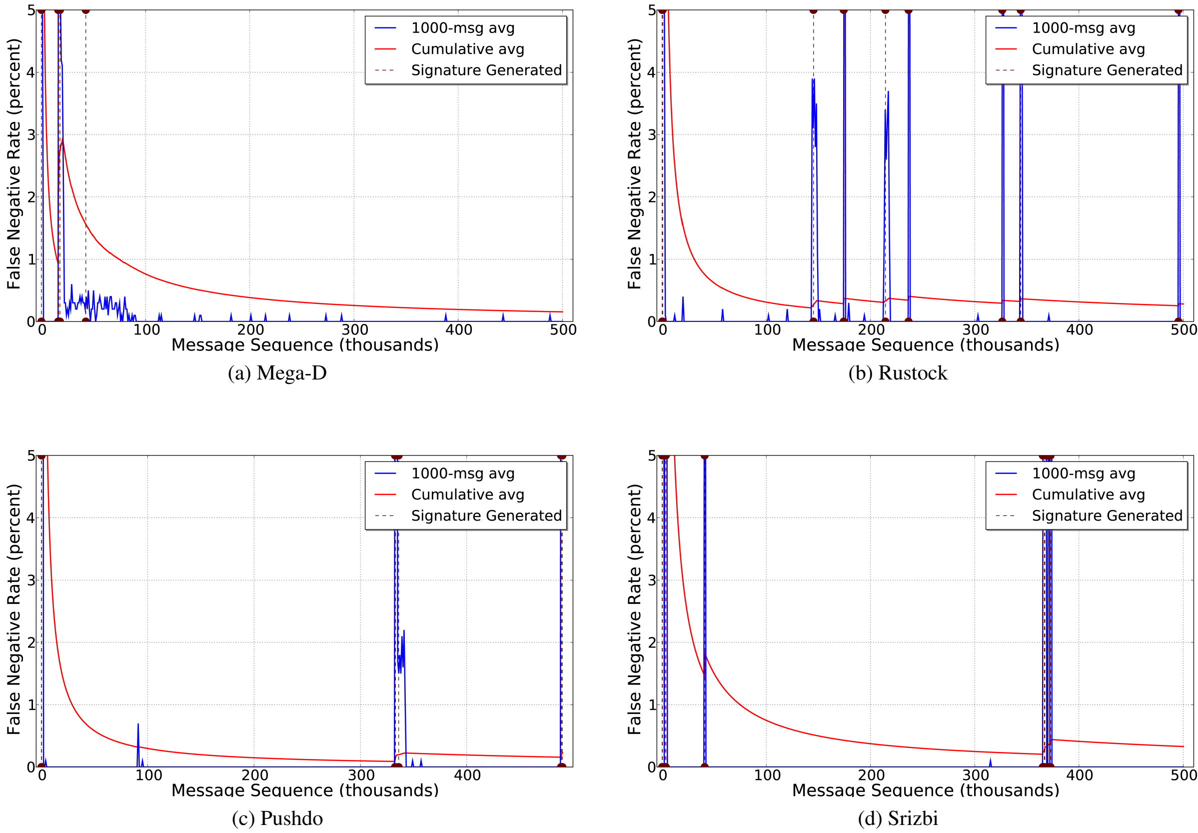

Figure 5. Classification effectiveness on Mega-D, Rustock, Pushdo, and Srizbi spam generated by a single bot, as a function of the testing message sequence. Experiment parameters: $k = 100$, $d = 0$ (that is, 100 training messages to generate each new signature, and immediate classification of test messages rather than post facto).

counted as a false negative. Classification delay helps, but even with $d = 500$, a new signature is not produced until we collect 500 messages *that match no other signature*.

Dynamic Behavior. We can better understand Judo by looking at its dynamic behavior. Figure 5 shows the average and cumulative false negative rate as a function of the testing messages. Dashed vertical lines indicate when Judo generated a new signature. Looking at the Mega-D plot (Figure 5a), we see that Judo is generating signatures during the first 20,000 messages in the testing set. After the initial flurry of signature generation, the false negative rate hovers just under 0.5%. After 100,000 testing messages, the false negative rate drops to nearly zero as the signatures are refined.

There are also two interesting observations here. First, peaks sometimes disappear from the graphs without the creation of a new signature. Normally we would expect that mismatches would be eliminated by inferring a new, previously missed underlying template. The effect in question here, though, is a result of using the second chance mechanism. For example, missing entries from a dictionary node can cause some false negative hits to occur until eventually the signature gets updated. This is done incrementally without the need to deploy a new signature, hence the absence of a dashed vertical line in the graph. The second observation is similar and relates to the creation of new signatures without any peaks appearing in the graph indicating the need for performing such an action. In this case, we need to remember that the training and testing track operate independently of each other. Thus it is sometimes the case that the training track observes a new template slightly before the testing track, and of course immediately generates a new signature. In this way, false negative hits are eliminated since the signature is already available for use when messages from the new template appear on the testing track.

We also observe that Pushdo exhibits different behavior in terms of the number of generated signatures. One would expect that the number of such signatures should be

Bots	Training	Testing
Xarvester	184,948	178,944
Mega-D	174,772	171,877
Gheg	48,415	207,207
Rustock	252,474	680,000

Table 4. Number of training and testing messages used in the real-world deployment experiment.

the same regardless of the parameters used. Although this holds for the other botnets, it does not for Pushdo due to the dictionary statistical test. Recall from Section 4 that we declare something as a dictionary only if Judo believes that it has seen every entry of it. This decision is based on the occurrences of the least-frequently observed element in the set under question. Hence, in the cases where we observe the same elements repeated over an extended number of messages, we can sometimes mis-conclude that we have seen the dictionary in its entirety. The use of a high threshold ensures that we keep such cases to a minimum. While processing Pushdo, the algorithm mistakenly classified a node as a dictionary before capturing all of its entries. As a result, Judo eventually generated multiple regular expressions for the same underlying template, with each one including a different subset of the underlying dictionaries.

5.4. Real-world Deployment

The previous experiments tested regular expressions produced by the template inference system against spam produced by a single bot instance. Doing so illuminates how quickly and how well the system learns a new template, but does not fully match how we would operationally deploy such filtering. We finish our evaluation with an assessment using *multiple* bot instances, one to generate the training data and the others to generate the test data. This configuration tells us the degree to which signatures built using one bot's spam are useful in filtering spam from multiple other instances. It also tests to a certain degree our assumption regarding the small number of templates actively used by botnets.

5.4.1. Methodology

We ran two instances of Xarvester and two of Mega-D in a contained environment akin to Botlab [10]. One of the bots was arbitrarily selected to provide the training corpus and the other the testing corpus. We also ran four instances of Rustock and six instances of Gheg. In a similar manner, one of the bots was arbitrarily selected to provide the training message set, and the remaining bots, combined, the testing set. Table 4 shows the number of messages generated by each bot.

As in the previous experiment, we "played back" both message streams chronologically, using the training data to generate a set of signatures incrementally as described in Section 4.3. Again, our metric of effectiveness is the false negative rate on the testing set. To maintain the accuracy of the results we preserved the chronological order of messages in the testing track. This ordering was a consideration for both Rustock and Gheg where we merged the output of multiple bots, as described earlier.

5.4.2. Results

Our results show Judo performed extremely well in this experiment, achieving false negative rates under 1% in most cases and generating no false positives. Table 5 shows the cumulative results for each combination of k and d, as well as the number of signatures generated for each botnet during the experiment. Although in all cases the training track was selected arbitrarily, in the case of Gheg we executed the experiment six times. Each time we used a different bot as the training track and the results show the worst false negative rate over these six choices.

Figure 6 shows the dynamic behavior of the Xarvester and Rustock bots in this experiment. Despite the fact of now using independent bots as the training and testing tracks, we see that the behavior is quite similar to the previous experiment, where only one bot was used. However, we observe slightly higher false negative rates in the cases where the training track consists of multiple bots. The reason for this higher rate is that the bots are not completely "synchronized", i.e., they do not switch to a new template or dictionary at the exact same time. What is important to note is that in all cases, even after a slight delay, such a switch does indeed occur across all hosts. Gheg exhibited similar behavior when we ran six different instances of the botnet. Recall that for our evaluation, we run the experiment six times and evaluated each one of the bots as being the provider of the training set. Even when taking into consideration the results from the worst run, as presented in Table 5, we can still see that monitoring just a single bot suffices for capturing the output of multiple other spamming hosts. Ultimately, the only reason for the differences in these executions was the precise ordering of the messages, and how early Judo was able to deploy a new signature each time. Our real-world experience verifies to a certain extent our original assumption that spam campaigns use only a small number of templates at any point in time in current practice. Of course, spammers could modify their behavior in response; we discuss this issue further in Section 6.

5.5 False Positives

One of the most important features of Judo is the unusual safety that the generated signatures offer. When deal-

Cumulative False Negative Rate						
Botnet	k \ d	0	50	100	500	Sig
Xarvester	50	0.07%	0.04%	0.02%	0%	6
	100	0.13%	0.06%	0.03%	0%	6
	500	1.00%	0.89%	0.78%	0.02%	6
Mega-D	50	0.09%	0.06%	0.03%	0%	1
	100	0.13%	0.10%	0.07%	0%	1
	500	0.92%	0.90%	0.87%	0.64%	1
Gheg	50	0.88%	0.86%	0.84%	0.64%	3
	100	1.13%	1.11%	1.08%	0.89%	3
	500	3.56%	3.54%	3.51%	3.33%	3
Rustock	50	0.99%	0.97%	0.95%	0.75%	6
	100	1.03%	1.01%	0.98%	0.78%	6
	500	1.49%	1.47%	1.44%	1.20%	6

Table 5. Cumulative false negative rate as a function of training buffer size k and classification delay d for spam generated by a multiple bot instances, one generating the training spam and the others the testing spam. The "Sig" column shows the number of signatures generated during the experiment. Signatures generated in this experiment produced no false positives on our corpora.

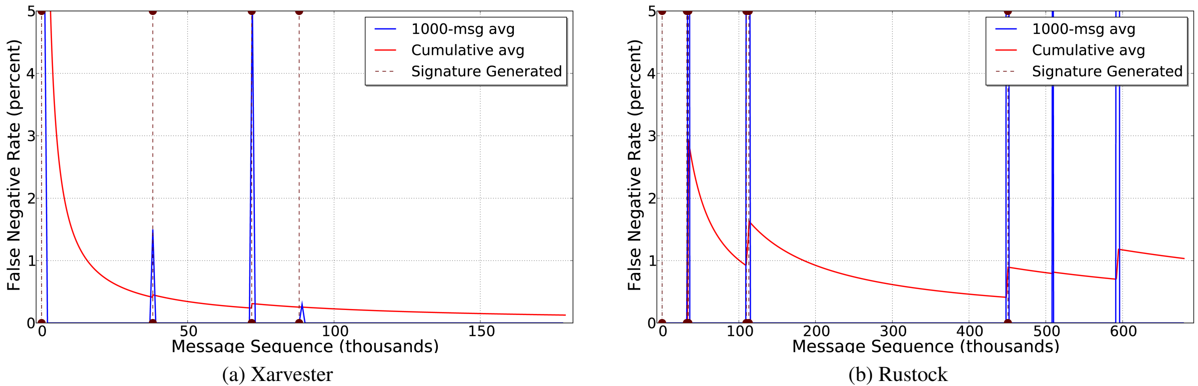

Figure 6. Classification effectiveness on Xarvester and Rustock spam with multiple bots: one bot was used to generate training data for the Judo system and the remaining bots to generate the testing data (1 other for Xarvester, 3 others for Rustock). Experiment parameters: $k = 100$, $d = 0$ (that is, 100 training messages to generate each new signature, and immediate classification of test messages rather than post facto).

ing with spam filtering, the biggest concern has always been falsely identifying legitimate messages as spam. Messages that fall under this category are known as false positives. In this section, we try to verify our claims and validate the safety of the signatures.

There are two main factors that affect the false positive rate of our system. The first is the fact that the generated signatures include information for both the headers and the body of the messages. In contrast to more naive methods of simply using URLs or subject lines for identifying spam e-mails, we use the additional information for minimizing the possibility of accidental matches. For the purpose of the current evaluation though, every header besides "Subject" was removed from the signatures. We made this choice for two reasons. First, we want to examine the system under a worst-case scenario, since it is straightforward to see

that the presence of additional headers can only improve our false positive rates. Second, we want to remove any possible temporal bias that would pollute our results. Such bias might be the result of age-sensitive headers like "User Agent".

The second factor which contributes to the system's strong false positive results is the existence of anchor and dictionary nodes. Recall that dictionary nodes are only created when Judo estimates that it has observed every single dictionary entry. As described in Section 4, these nodes impose strong limitations as to what messages a signature can match. Hence we add the constraint that all final signatures must contain at least one anchor or dictionary node. If this constraint is violated, we consider the signature *unsafe* and discard it. Although this heuristic can potentially hinder the false negative rates, it also makes sure that false positives remain very small. We confirmed that all signatures used in our evaluation were *safe*. There was only one case where it became necessary to discard signatures, as described in Section 5.2.

We first look at the Storm templates. Based on our dataset description in Section 5.2, we split this analysis into URL and non-URL (stock) templates. For the former category, which included signatures from the self-propagation and pharmaceutical templates, we had no false positives in three of the four legitimate mail corpora. In the lists.gnu.org corpus, signatures produced from 100 or fewer training messages resulted in a false positive rate of 1 in 50,000. This rate arose from a small number of cases in which dictionaries were not constructed until $k = 500$. For the remaining values of k the result was again zero matches.

Storm templates that did not contain a URL proved to be a harder workload for our system, and the only scenario where *unsafe* signatures were generated and discarded. Although URLs are not a requirement for producing good signatures, the problem was amplified in this case due to the very small length of messages generated by the Storm botnet. Hence it is sometimes the case that the system cannot obtain enough information for smaller numbers of training messages. This issue, though, is eliminated when moving to higher values of k.

For these stock templates, the 99th percentile false positive rate for $k \leq 100$ was under 0.1% across all templates, and with a maximum false positive rate of 0.4% at $k = 10$. For $k \geq 500$, the maximum false positive rate was 1 in 50,000 on the Enron corpus, and zero for the remaining three corpora. We emphasize again that we are using stripped down versions of the signatures (subject and body patterns only); including additional headers ("MIME-Version" and "Content-Transfer-Encoding") eliminated all false positives. We further validated these numbers by making sure that these additional headers were indeed included in the messages of our legitimate mail corpora. Hence we confirmed that all mismatches arose due to the corresponding header regular expressions failing to match, and not due to bias of our specific dataset.

The Botlab workload (Section 5.3) produced zero matches against all corpora for the MegaD, Pushdo and Srizbi botnets. The only exception was the signatures generated for Rustock. We had zero matches against the TREC 2007 corpus. When testing against the remaining corpora, signatures for this botnet produced an average false positive rate between 0.00021% to 0.01%. The 99th percentile false positive rate was at most 0.24% and 95th percentile at most 0.04%. When looking into this issue further we identified the source of the problem as the inability of these signatures to produce all possible dictionaries. One reason was the very high threshold used for allowing the conversion of a character class to a dictionary node. We are currently looking into this particular problem for further improvement. Once again, though, we note that incorporating additional headers gives a worst-case false positive rate of 1 in 12,500 due to signature mismatches and not because of the absence of these headers in our corpora.

For all other signatures generated from messages captured in our sandbox environment, the result was zero false positive matches across all corpora for all botnets. Note that these results correspond to our real-world deployment experiment, with signatures being generated for the very latest spam messages sent by the botnets. The structure of these messages allowed for the creation of very precise regular expressions, such as the example presented in Figure 7 for the MegaD botnet.

5.6 Response Time

One reasonable concern can be the time Judo requires for generating a new signature. Since we aim to shrink the window between the time a new campaign starts and the time we deploy a filter, being able to quickly infer the underlying template is crucial. As already shown in Section 4.4, execution time is not a concern for the system, as it takes under 10 seconds in almost all cases to run the algorithm. Thus the only question left to answer is how long it takes to build up the required training sets.

Obviously such a metric is dependent on the spamming rate of botnets. It is also possible that messages from multiple campaigns might be interleaved in a corpus, further complicating the exact time it takes to create the training set for each one. Despite this, knowing the spamming rate at which bots operate can give us a good high-level estimate of the time requirements imposed by Judo. Focusing on the datasets used in our real-world deployment scenario, we identified that the six Gheg bots took less than 6 hours to send all 255,622 e-mails. This translates to about 118 messages per minute, for each bot. In a similar way, the four

```
Subject ^(RE: Message|new mail|Return mail|Return Mail|Re: Order status|no-reply| ▷
Your order|Delivery Status Notification|Delivery Status Notification \(Failure\))$

^<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD>
<META http-equiv=Content-Type content="text/html; charset=(us-ascii|iso-8859-2| ▷
iso-8859-1|windows-1250|Windows-1252)">
</HEAD>
<BODY><a href="http://(talklucid|samefield|famousfall|bluedoes|meekclaim).com/" ▷
target="_blank">
<img src="http://(talklucid|samefield|famousfall|bluedoes|meekclaim).com/dyuwqlk.jpg" ▷
border="0" alt="Show picture and go to site now!"></a></BODY></HTML>$
```

Figure 7. Fragment of a template generated for the Mega-D botnet on August 26, 2009. Only the subject and body are displayed with full dictionaries, exactly as they were captured. Recall that templates are inferred from only the output of bots, without any access to the C&C channel and without any information regarding the underlying mechanisms used.

Rustock bots we had deployed required only 20 hours to send 932,474 messages, which gives us a spamming rate of 194 messages per minute, for each bot. Considering that for modern botnets, Judo showed excellent performance for any values of $k \leq 500$, we conclude that the system requires only a few minutes for the deployment of a new signature. In fact, the time it takes to do so directly affects all results of our evaluation in Section 5.3 and Section 5.4. Recall that while the system is creating the required training set for a campaign, any mismatches on the testing tracks are registered as false negatives. Despite this fact, we can see that Judo is still able to maintain excellent performance, hence satisfying our initial goal of a fast response time.

5.7. Other Content-Based Approaches

The efficacy of Judo rests on two essential characteristics: its unique vantage point at the *source* of the spam, and the template inference algorithm for creating signatures from this spam. Regarding the latter, one may naturally ask if a *simpler* mechanism suffices. In short, we believe the answer is "No." To our knowledge two such simple approaches have been advanced: subject-line blacklisting and URL domain blacklisting. We consider these in turn.

Subject-line Blacklisting. Filtering based on message subject is one of the earliest spam filtering mechanisms in deployment. Most recently, it was used by the Botlab project [10] to attribute spam to specific botnets.[5] Unfortunately, it is very easy (and in some cases desirable for the spammer) to use subject lines appearing in legitimate mail. One of the Mega-D templates found in our previous experiment used a subject-line dictionary containing "RE: Message," "Re: Order status,", "Return mail," and so on, and can be seen in Figure 7. Such a template cannot be effectively characterized using the message subject alone.

URL Domain Blacklisting. URL domain blacklists (e.g. [3, 4]) are lists of domains appearing in spammed URLs. In our experience, domain names appearing in spam do indeed provide a strong signal. However, there are at least two cases where domain names *alone* are not sufficient. The first case, spam not containing URLs (stock spam for example), simply cannot be filtered using URL domain signatures. (Section 5.2 shows that Judo is effective against this type of spam).

The second type of spam uses "laundered" domains, that is, reputable services which are used to redirect to the advertised site. Most recently, for example, spam sent by the Gheg botnet was using groups.yahoo.com and google.com domains in URLs. We suspect this trend will continue as URL domain blacklisting becomes more widely adopted.

"Focused" Bayesian Signatures. An intriguing possibility is using a Bayesian classifier to train on a single campaign or even the output of a single bot, rather than a large universal corpus of spam as is conventional. Our cursory evaluation using SpamAssassin's [26] Bayesian classifier showed promise; however, in addition to a formal evaluation, a number of technical issues still need to be addressed (the larger size of Bayesian signatures, for example).

We also trained SpamAssassin's Bayesian filter on a generic spam corpus of over one thousand recent messages along with the SpamAssassin 2003 "ham" corpus. It fared poorly: from a sample of 5,180 messages from the Waledac botnet, 96% received a score of 0 (meaning "not spam") due to the complete absence of tokens seen in the generic spam corpus; none were given a score above 50 out of 100.

[5] Note that John *et al.* do not suggest that subject lines alone should be used for *identifying* spam, but for "classifying spam messages as being sent by a particular botnet" after being classified as spam.

Enterprise Spam Filtering Appliances. We ran a subset of the spam corpora from the Gheg, MegaD, Rustock, and Waledac botnets through a major spam filtering appliance deployed on our university network. Spam from the Waledac and Rustock botnets was completely filtered based upon the URLs appearing in the message bodies, and MegaD spam was correctly filtered via a generic "pharmaceutical spam" rule. However, only 7.5% of the spam from the Gheg botnet was correctly identified as such; these messages were German language pharmaceutical advertisements laundering URL reputation through google.com via its RSS reader application.

IP Reputation. IP reputation filtering is a widely deployed technique for identifying incoming spam messages. It is orthogonal to content-based approaches, such as Judo; today reputation-based and content-based approaches are used in tandem in major spam filtering systems. This is because IP reputation filtering alone does not provide a complete spam-filtering solution. In addition to the operational overhead of dealing with false positives (e.g., deblacklisting), they are also vulnerable to reputation laundering. By sending spam using webmail accounts, for example, spammers can effectively hide behind reputable, high-volume sources and increase the deliverability of their e-mails. In 2008, according to [21], around 10% of all spam originated by web-based e-mail and application service providers. Our own informal analysis at a high-volume mail server indicated that around 16% of all messages classified as spam, originated from "high reputation" mail servers. In fact, nearly 50% of those were sent by a single major webmail provider. This analysis was performed on September 14, 2009. With IP reputation systems becoming targetted, approaches like Judo can offer another valuable tool in the fight against spam. Our system does not rely on IP features for achieving high effectiveness and more importantly, it can do so by offering close to zero false positives.

6. Discussion

There are four questions invariably asked about any new anti-spam system: how well does it filter spam, how often does it misclassify good e-mail in turn, how easy or expensive is it to deploy and how will the spammers defeat it? We discuss each of these points briefly here.

As we have seen, template inference can be highly effective in producing filters that precisely match spam from a given botnet. Even in our preliminary prototype we have been able to produce filters that are effectively perfect for individual campaigns after only 1,000 samples. To a certain extent this result is unsurprising: if our underlying assumptions hold, then we will quickly learn the regular language describing the template. Even in less than ideal circumstances we produce filters that are very good at matching subsequent spam. The catch, of course, is that each of our filters is over-constrained to only match the spam arising from one particular botnet and thus they will be completely ineffective against any other spam.

The hidden benefit of this seeming drawback is that filters arising from template inference are unusually *safe*. Their high degree of specialization makes them extremely unlikely to match any legitimate mail and thus false positive rates are typically zero or extremely close thereto. To further validate this hypothesis, we provided the regular expressions corresponding to the data for Xarvester to a leading commercial provider of enterprise anti-spam appliances. They evaluated these filters against their own "ham" corpus and found no matches. Given this evidence, together with our own results, we argue that template inference can be safely used as a pre-filter on any subsequent anti-spam algorithm and will generally only improve its overall accuracy.

There are three aspects to the "cost" of deploying a system such as ours. The first is the complexity of capturing, executing and monitoring the output of spam bots. As more bot instances can be maintained in a contained environment, new filters can be generated more quickly. While this is by no means trivial, it is routinely done in both academia and industry and there is a broad base of tools and technology being developed to support this activity. The second issue concerns the efficiency of the template inference process itself. Here we believe the concern is moot since the algorithm is linear time and our untuned template extraction algorithm is able to generate regular expressions from 1000 messages in under 10 seconds, and update the expression in 50-100 ms. Next, there is the issue of integration complexity since it is challenging to mandate the creation of new software systems and interfaces. However, since our approach generates standard regular expressions—already in common use in virtually all anti-spam systems—the integration cost should be minimal in practice.

Finally, we recognize that spam is fundamentally an adversarial activity, and successful deployment of our system would force spammers to react in turn to evade it. We consider the likely path of such evolution here. There are three obvious ways that spammers might attempt to stymie the template inference approach.

First, they can use technical means to complicate the execution of bots within controlled environments. A number of bots already implement extensive anti-analysis actions such as the detection of virtual machine environments and the specialization of bot instances to individual hosts (to complicate the sharing of malware samples). Moreover, some botnets require positive proof of a bot's ability to send external spam e-mail before providing spam template data. While this aspect of the botnet arms race seems likely to continue, it also constitutes the weakest *technical* strategy against template inference since there is no fundamental test

to distinguish a host whose activity is monitored from one whose is not.

A more daunting countermeasure would be the adoption of more complex spam generation languages. For example, multi-pass directives (e.g., shuffling word order after the initial body is generated) could easily confound the algorithm we have described. While there is no doubt that our inference approach could be improved in turn, for complex languages the general learning problem is untenable. However, there are drawbacks in pursuing such complexity for spammers as well. Template languages emerged slightly over 5 years ago as a way to bypass distributed spam hash databases [24] and they have not changed significantly over that time. Part of the reason is that they are easy for spammers to use and reason about; a new spam campaign does not require significant testing and analysis. However, a more important reason is that there are limits to how much polymorphism can be encoded effectively in a spam message while still preserving the underlying goal. To be effective, pitches and subject lines must be roughly grammatical, URLs must be properly specified, and so on. Randomizing the letters across such words would defeat template inference but also would likely reduce the underlying conversion rate significantly.

Finally, spammers might manage the distribution of templates in a more adversarial fashion. In particular, were each bot instance given templates with unique features then the regular expressions learned from the output of one bot would suffer from overtraining; they would be unable to generalize to spam issued from another bot in the same botnet. Depending precisely on how such features were generated, this could add significant complexity to the underlying inference problem at relatively low cost to spammers, and without significantly changing the overall "look and feel" of such messages to potential customers. We leave the challenge of joint learning across bot instances to future work should the spam ecosystem evolve in this manner.

7. Conclusion

In starting this paper we observed that strong defenses benefit from obtaining current and high quality intelligence. This point is hardly lost on the anti-spam community and over time there have been many efforts to share information among sites, precisely to shrink the window of vulnerability between when a new kind of spam appears and a corresponding e-mail filter is installed. Historically, these efforts have been successful when the information gathering itself can be centralized and have floundered when they require bilateral sharing of mail samples (even in a statistical sense). Thus, IP-based blacklists constitute intelligence that, upon being learned, is shared quickly and widely, while content-based filter rules continue to be learned independently by each defender.

To put it another way, the receiver-oriented learning approach makes it challenging to automatically share new spam intelligence (for reasons of privacy, logistics, scale, etc.). However, given that a small number of botnets generate most spam today, this problem can be neatly sidestepped. We have shown that it is practical to generate high-quality spam content signatures simply by observing the output of bot instances and inferring the likely content of their underlying template. Moreover, this approach is particularly attractive since the resulting regular expressions are highly specialized and thus produce virtually no false positives. Finally, while we recognize that there are a range of countermeasures that an adversary might take in response, we argue that they are not trivial for the attacker and thus that the template inference approach is likely to have value for at least a modest period of time.

8. Acknowledgments

We would like to thank John P. John *et al.* for the Botlab project data. We would also like to acknowledge Brandon Enright and Brian Kantor for their invaluable assistance. Some of the experiments were conducted on the UCSD FWGrid cluster, which is funded in part by National Science Foundation Research Infrastructure grant EIA-0303622. This work was supported in part by National Science Foundation grant awards NSF-0433668, NSF-0433702, NSF-0905631, and NSF-6783527, as well as in-kind support from Cisco, Microsoft, Google, Yahoo!, ESnet, and UCSD's Center for Networked Systems. Opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the funding sources.

References

[1] jwSpamSpy Spam Domain Blacklist. `http://www.joewein.net/spam/spam-bl.htm`.
[2] lists.gnu.org. `ftp://lists.gnu.org`.
[3] SURBL. `http://www.surbl.org`.
[4] URIBL – Realtime URI Blacklist. `http://www.uribl.com`.
[5] G. V. Cormack and T. R. Lynam. Online supervised spam filter evaluation. *ACM Trans. Inf. Syst.*, 25(3), July 2007.
[6] H. Drucker, D. Wu, and V. N. Vapnik. Support vector machines for spam categorization. *Neural Networks, IEEE Transactions on*, 10(5):1048–1054, 1999.
[7] J. Göbel, T. Holz, and P. Trinius. Towards proactive spam filtering. In *Proceedings of the 6th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2009.
[8] G. Hulten, A. Penta, G. Seshadrinathan, and M. Mishra. Trends in Spam Products and Methods. In *Proceedings of*

the First Conference on Email and Anti-Spam (CEAS), 2004.

[9] A. G. K. Janecek, W. N. Gansterer, and K. A. Kumar. Multi-Level Reputation-Based Greylisting. In *Availability, Reliability and Security (ARES)*, pages 10–17, 2008.

[10] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying Spamming Botnets Using Botlab. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[11] J. Jung and E. Sit. An empirical study of spam traffic and the use of DNS black lists. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 370–375, New York, NY, USA, 2004. ACM Press.

[12] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: An Empirical Analysis of Spam Marketing Conversion . In *ACM CCS*, pages 3–14, Alexandria, Virginia, USA, October 2008.

[13] B. Klimt and Y. Yang. Introducing the Enron Corpus. In *Proceedings of the First Conference on Email and Anti-Spam (CEAS)*, 2004.

[14] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. On the Spam Campaign Trail. In *Proceedings of the 1st USENIX LEET Workshop*, 2008.

[15] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamcraft: An Inside Look At Spam Campaign Orchestration. In *Proceedings of the 2nd USENIX LEET Workshop*, 2009.

[16] B. Leiba and J. Fenton. DomainKeys Identified Mail (DKIM): Using Digital Signatures for Domain Verification. In *Proceedings of the Fourth Conference on Email and Anti-Spam (CEAS)*, 2007.

[17] D. Lowd and C. Meek. Good Word Attacks on Statistical Spam Filters. In *Proceedings of the Second Conference on Email and Anti-Spam (CEAS)*, 2005.

[18] Marshal8e6 TRACELabs. Marshal8e6 security threats: Email and web threats. `http://www.marshal.com/newsimages/trace/Marshal8e6_TRACE_Report_Jan2009.pdf`, 2009.

[19] J. Mason. SpamAssassin public corpus. `http://spamassassin.apache.org/publiccorpus`, 2003.

[20] R. McMillan. What will stop spam?, December 1997.

[21] MessageLabs Intelligence. 2008 annual security report. `http://www.messagelabs.com/mlireport/MLIReport_Annual_2008_FINAL.pdf`, 2008.

[22] T. A. Meyer and B. Whateley. SpamBayes: Effective open-source, Bayesian based, email classification system. In *Proceedings of the First Conference on Email and Anti-Spam (CEAS)*, 2004.

[23] A. Ramachandran, D. Dagon, and N. Feamster. Can DNSBLs Keep Up with Bots? In *Proceedings of the Third Conference on Email and Anti-Spam (CEAS)*, 2006.

[24] Rhyolite Corporation. Distributed checksum clearinghouse, 2000.

[25] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A Bayesian Approach to Filtering Junk E-Mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin, 1998. AAAI Technical Report WS-98-05.

[26] M. Sergeant. Internet Level Spam Detection and SpamAssassin 2.50. In *MIT Spam Conference*, 2003.

[27] H. Stern. A Survey of Modern Spam Tools. In *Proceedings of the 5th Conference on Email and Anti-Spam*, 2008.

[28] B. Taylor. Sender Reputation in a Large Webmail Service. In *Proceedings of the Third Conference on Email and Anti-Spam (CEAS)*, 2006.

[29] 2007 TREC Public Spam Corpus. `http://plg.uwaterloo.ca/~gvcormac/treccorpus07`, 2007.

[30] M. W. Wong. Sender Authentication: What To Do, 2004.

[31] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming Botnets: Signatures and Characteristics. In *Proceedings of ACM SIGCOMM*, pages 171–182, 2008.

[32] L. Zhang, J. Zhu, and T. Yao. An evaluation of statistical spam filtering techniques. *ACM Transactions on Asian Language Information Processing (TALIP)*, 3(4):243–269, December 2004.