SpamStrangler: A Chord-Based Distributed Spam Detection Tool

Andrew Chung, Igor Tarashansky, Murali Vajapeyam, Ryan Wagner

December 6, 2002

Abstract

The ever increasing volume of unsolicited commercial e-mail (also known as *spam*) has prompted the creation of several spam filtering applications [spa02], [Ham02]. This paper describes a new spam filter that relies on a distributed database of spam e-mail content. The approach is based on the fact that spammers try to address many different recipients with messages of similar content. For each detected spam e-mail, the database contains hashes of selected parts of the e-mail. To check for spam, the filter checks the hashes of selected parts of new e-mail messages against the hashes in the database. The distributed nature of our system permits sharing of spam information. The database is built on top of Chord[SMK+01], an efficient and scalable distributed hash table. Chord permits the database to be very large without significantly degrading spam filter performance. Our results show that our system has good detection rate and low misfire rate.

1 Introduction

Unsolicited commercial e-mail, or "spam", is an expensive problem. According to a BBC news article, the network costs of spam distribution have been estimated as $8.8 billion annually [Shi02]. The time wasted in reading or deleting spam e-mails also represents a significant cost in labor productivity; according to the article, such costs are estimated as $1 for each piece of spam.

Spam is difficult to control. Many spam e-mail headers are forged, leaving authorities with little information as to where spammers are located. Technology alone is no panacea, either. Just as spam detection technology evolves, so do spammers, manipulating headers and e-mail bodies to pass through even the most sophisticated filters.

A good spam filter should satisfy the following properties:

accuracy: the filter should detect spam with high probability. On the other hand, the filter should not tag non-spam e-mail as spam.

robustness: it should not be easy for spammers to write spam specifically designed to defeat the system.

efficiency: checking and submitting e-mails should be fast operations. Besides, the filter should keep its storage requirements low.

This paper proposes a distributed system to detect spam. The approach is based on the assumption that spammers send similar e-mails to many different users. The idea is to maintain a distributed database of known spam content. When a user manually detects a spam and submits it to the database, some of the information from the e-mail is placed into the database. Mail clients/servers, on the other hand, can use the database to check received e-mails for spam before taking any further action, such as forwarding the e-mail ahead or dropping it in the user's mailbox.

The power of having a distributed system comes from the shared database: different users do not have to each individually decide whether a given e-mail is spam. Having users share spam information should reduce the numbers of users who receive a specific piece of spam, and should also reduce storage costs of spam. We built our database on top of Chord, an efficient, robust, distributed and scalable database system that exhibits logarithmic-time key lookup[SMK+01].

Our system is intended as a toolkit for detecting spam. Clients/servers have ample flexibility in specifying policies for detecting spam and dealing with detected spam. One possibility includes keeping lists of trusted e-mail addresses and only checking for spam on e-mails from senders not in this list. Mail clients may immediately delete a detected spam or keep the e-mail in a special "spam" folder, depending on its user configuration. A mail server might decide not to deliver a detected spam to the client, saving network costs. Both approaches can save human users from the burden of reading/receiving spam.

Our system may also be integrated with other existing techniques for spam detection, such as blacklist blocking and e-mail content analysis, to provide a more robust and reliable method for detecting spam.

The rest of this paper is structured as follows: section 2 describes our design; section 3 describes the architecture and implementation of our system; section 4 presents our tests and subsequent analysis of the results; section 5 describes related work; section 6 describes planned future work; section 7 lists our conclusions.

2 Design

2.1 Design Considerations

A spam e-mail is usually sent to thousands or maybe millions of users. If we can maintain a public database of known spam, mail programs with access to the database might be able to tag received e-mail as spam before it reaches the end user. This can drastically reduce the number of users who receive a given piece of spam e-mail.

Spammers might try to defeat the system by sending different e-mail to different people, for example by including the user's name in the e-mail. Our assumption is that, despite their efforts, spammers still send essentially the same spam message to many different users. The reason is that it is infeasible for spammers to generate the same meaningful English content in thousands of different ways. Different versions of the same spam will be very similar to each other, except for small changes such as personalizations. In particular, different versions of the same spam will contain many of the same sentences.

Spammers might also try to insert random content in their e-mails, so that different users receive different messages. Since spammers still want to convey intelligible advertisements, they will not try to insert large amounts of random content in the middle of their messages. On the contrary, spammers would insert random content either where it is not visible to the user, such as in HTML comments; or at a section of the e-mail that users normally ignore (such as headers or at the end of e-mails).

Our system also rests on the assumption that spam e-mail must contain some type of contact information; after all, the purpose of the e-mail is to cause the user to buy a product or use some service offered by the spammers. We define contact information as the set of URLs, phone numbers and e-mail addresses that the e-mail recipient should use in order to contact the spammers. This contact information cannot be easily changed, and can be used as a valuable resource for spam detection. Finally, contact information is very specific: if spam detection is mainly tied to the contact information present in the e-mail, the probability of false alarms becomes smaller.

Finally, we assume that non-spam has essentially unique content. For one, minimum-length sentences in a non-spam e-mail are unlikely to be found in other spam or even non-spam messages. Besides, analysis of spam and non-spam has showed us that non-spam has significantly less contact information than spam e-mail. We are thus led to believe that both submission of spam and malicious submission of non-spam are unlikely to cause our system to incorrectly filter non-spam.

2.2 Our Approach

Based on our assumptions, we designed a spam filter based on a distributed database of spam e-mail content. Each e-mail user runs a database server, with the set of different servers constituting a distributed database accessible to everyone using the spam-detection system.

For a given spam e-mail, the database stores selected portions of the e-mail that are likely to be encountered in modified versions of the same spam. In order to facilitate comparison with pieces extracted from other e-mails, the database actually stores collision-resistant hashes of the pieces instead of the pieces themselves. Storing hashes also reduces the amount of storage in the database.

When a user decides that a received e-mail is spam, he submits the spam e-mail to the database. Our filter parses the e-mail, and selects certain parts to store in the database. When checking a received e-mail for spam, the filter parses the e-mail and selects certain parts to check in the database. Depending on which and how many of the parts are found in the database, the received e-mail is tagged as spam.

Our filter parses the e-mails to obtain three types of elements: user-visible text content, contact information and a list of sentences.

2.2.1 User-visible text content

Our filter extracts the entire user-visible text-content of the e-mail, both when submitting known spam or checking received e-mail. The parser should use different algorithms to extract this text content, depending on the type of encoding in the e-mail (plaintext, HTML, etc.). This approach permits reliable spam detection in case spam e-mails with identical text content are sent to different users; it is meant to permit effective and efficient detection of less sophisticated forms of spam.

Because the stored hash corresponds only to the user-visible body-text, a reasonable level of robustness can be achieved as early as this stage. In particular, the filter ignores any (potentially forged) header content or any (potentially random) content inserted as HTML comments.

2.2.2 List of sentences

We assume that, for different versions of the same spam e-mail, the user-visible body content contains mostly the same sentences. The filter parses the e-mail to obtain the list of all sentences that appear in the user-visible text content of the e-mail. It selects a random sample of all the sentences with a certain minimum length, and stores them in the database. The rationale for only considering sentences with a minimum length is to prevent the occurrences of false alarms: short sentences are more likely to also occur in non-spam e-mail.

Such an approach could potentially cause many misses because the sentences chosen for submission generally do not coincide with the sentences chosen for checking. However, this is a minor issue: if the first few checks result in misses because of the randomness, the corresponding users will manually detect the spam and submit it to the system. This will cause the number of stored sentences to increase, rendering it more likely that future checks will detect the spam. Eventually, enough sentences get stored, and subsequent checks effectively detect the spam.

If the e-mail contains few sentences that exceed the minimum length, all sentences get stored. Otherwise, only a random sample of messages is chosen to be stored. The main reason for this approach is efficiency: storing too many sentences will cause the submit procedure to slow down. Besides, having each user submit different parts of the spam message contributes to distribute the workload on each user. The randomness, on the other hand, is important to make the system robust. If the spam filter selects sentences according to a deterministic algorithm, spammers might tailor their messages so that precisely those sentences get small variations (such as personalizations).

A potential problem with this approach is that spammers might insert a large amount of variable random content at places that are ignored by the user, such as the end of the e-mail. This would reduce the probability of sentence-based detection. A solution is to consider only sentences in the first few paragraphs of the e-mail– we assume that that's where the important information in the e-mail lies.

2.2.3 Contact Information

The filter parses the e-mail for three types of contact information: phone numbers, URLs and e-mail addresses. Presumably, the user should call these numbers, visit these URLs, or send e-mails to these addresses if he wishes to contact the spammers. All such instances of contact information are stored in the database. As noted in section 2.1, this information should provide the most useful component of our spam detection algorithm. For one, contact information is non-forgeable: users must be able to correctly contact the spammers if the spam is to be of any use to the spammers. The spammers' contact information is also very specific, and not likely to be found in non-spam e-mail.

One might think that if the parser is precise in differentiating the real contact information from random URLs, phones and e-mails inserted in the message, the filter would not need to rely on sentences to detect spam. As was argued in section 2.1, contact information is a more reliable technique for spam detection. However, a practical implementation would probably not be completely accurate in isolating the real contact information from random URLs, phone numbers and e-mail addresses contained in the e-mail. Thus, the sentence-based technique should be used in conjunction with contact information to improve the reliability of detection. Another issue regards deliberate attacks: since our database is distributed and decentralized, it is hard to control who can submit spam to the system. A malicious user may, for example, create artificial spam messages by replacing the contact infos in real spam messages with the contact infos of normal users. If such artificial spam messages are submitted to the database, non-spam messages containing the contact infos of those users might be filtered as spam. In section 2.1 we argued that, even in face of such attacks, non-spam is unlikely to be tagged as spam, due to the sentence uniqueness of non-spam messages. This provides an additional reason to use e-mail sentences, and not just contact information, to reliably detect spam.

3 Architecture and Implementation

3.1 Overview

Our system is composed mainly of three parts: Parser, DHT Interface and SpamStrangler (see Figure 1). SpamStrangler interacts with the mail client/server, and is responsible for submitting spam and checking for spam. If a user decides that an e-mail he received is spam, the mail client submits the e-mail using SpamStrangler.

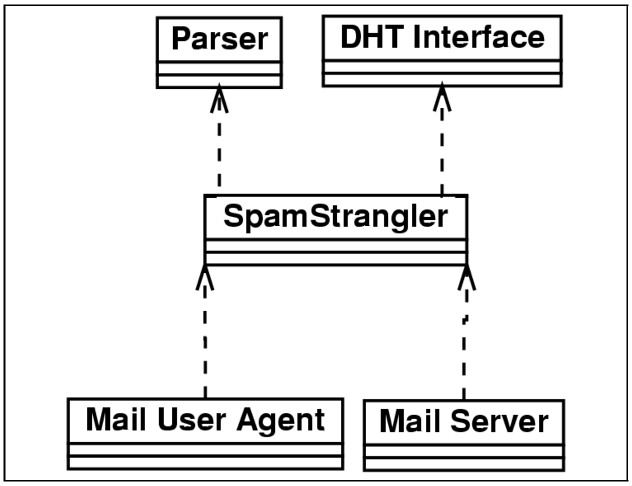

Figure 1: High level class hierarchy of the SpamStrangler system.

SpamStrangler uses methods from the Parser module to extract the body-text, contact information and individual sentences from the e-mail. When submitting a spam, SpamStrangler uses the methods in the DHT interface to store the body, as well as a sample of the contact information and sentences found in the e-mai, in the databasel. When checking for spam, SpamStrangler uses the methods in the DHT interface to check whether the body of the e-mail is in the database; otherwise, it checks whether a sample of the contact information and sentences are in the database. If the body of the e-mail is found in the database, SpamStrangler reports the e-mail as a very likely spam. Otherwise, SpamStrangler reports spam-like quality of an e-mail, depending on the contact information and sentences actually found on the database.

The entire system was implemented in C++. The next sections describe the implementation of the individual parts of the system in more detail.

3.2 Parser

Currently the Parser supports plaintext, HTML, and q-encoding. The Parser module contains methods for returning various parts of the e-mail, such as body, header, contact information (phone numbers, URLs and e-mail addresses), individual sentences and individual words. To aid in HTML parsing, we used a library that can be downloaded online[Con02].

The body returned by the Parser consists only of the text portion of the e-mail: HTML content, attachments and header are ignored. Case and non-alphanumeric characters are also ignored: the body returned by Parser consists of a string of uppercase alphanumeric characters.

Both sentences and individual words are extracted only from the body-text of the message. Sentences are separated via common English punctuation marks, whereas words are separated via spaces and tab characters.

Contact information, on the other hand, is extracted from both the body-text and the HTML content of the message. The contact information parsing currently uses regular expressions to detect all phone numbers, URLs and e-mail addresses that appear in the body of the given e-mail. The list of contact information returned by the Parser does not contain duplicates.

3.3 DHT Interface

The DHT Interface consists of two methods: put and get. The first method inserts a string in the database, the second method determines whether a string is in the database . The DHT interface is responsible for computing a collision-resistant hash of the string, instead of the string itself.

We chose SHA-1 as the collision-resistant hash function. The distributed database was built using Chord, an efficient, scalable, robust peer-to-peer database system. Chord was chosen primarily for its lack of centralization and scalability, making it easy for new users to join the distributed database. In Chord, the expected number of lookups for a given key is logarithmic in the number of nodes in the distributed system. A low number of network lookups is crucial for keeping our system efficient.

In order to ensure data integrity, the values inserted in the database are keyed by their corresponding SHA-1 hashes. Since in our system the values are themselves SHA-1 values, our keys consist of hashes of hashes of strings in the e-mails.Since the keys are bound to the values in the database entries, methods put and get need only one argument, the string value.

The DHT interface provides both synchronous and asynchronous versions of the put and get methods. In the synchronous interface, both methods are blocking; the asynchronous interface, on the other hand, permits multiple pending put and get requests. In order to implement the asynchronous interface, we used libasync, an event-driven interface[Maz01]. We expect the asynchronous interface to be the one used in practice–because it permits overlapping of requests, it will probably be more efficient.

Both the synchronous and asynchronous interfaces are implemented via RPCs, and make use of the RPC interface available for interaction with Chord. The communication between the filter and the Chord server is established via Unix sockets; as a consequence, the mail filter and the Chord server must be on the same machine.

3.4 SpamStrangler

The interface to SpamStrangler consists of methods `submitSpam` and `checkSpam`, both of which take an e-mail (represented as a string) as argument. The SpamStrangler module interacts directly with the mail client/server. Method `submitSpam` is used to add spam to the database; method `checkSpam` checks an e-mail against the database to determine if it is spam. The checking method returns a integer indicating the spam-like quality of the checked e-mail.

When SpamStrangler receives an e-mail for submission, it uses the Parser to obtain the user-visible text content, the list of contact information, and the list of sentences. Its first step is to store the body of the e-mail in the database, using method "put" in the DHT interface.

The filter selects the first N phone numbers, N e-mail addresses, and N URLs found in the e-mail, and submits them to the database (N is an adjustable parameter in our filter). If less than N instances of a certain contact info (such as phone number) are present in the e-mail, all are submitted to the database. As argued before, we judged that the advertisement content–in particular contact information–would be located in the beginning of the e-mail. Variable N is currently set to 15.

SpamStrangler only submits sentences with a minimum length (currently 48 characters). Among all such sentences in the e-mail, it selects M random sentences to submit to the database, or all of them if there are less than M. Currently, M is set to 15.

Method `checkSpam` uses the same algorithm as `submitSpam` to obtain a list of contact information and a list of sentences to check against the database. In the initial step, however, `checkSpam` simply verifies if the body of the e-mail is in the database. If so, it returns 5–indicating highest degree of spam content (message identical to a known spam). Otherwise, it checks the selected sentences and contact information against the database. Method `checkSpam` returns 0 if no matches were found, and returns a number from 1 to 4 (depending on the number of matches) if the number of matches is positive.

Matches for different types of contact info may be weighted differently (for example, it seems natural to weight phone numbers higher than e-mail addresses), and matches for sentences may be weighted differently from contact information. The current weights are 1 for all these types–this is equivalent to simply counting the number of matches in the database for all selected contact information and sentences. Future work includes determining a more optimal set of weights for the filter.

4 System Evaluation

SpamStrangler underwent several extensive tests for accuracy and performance. The tests were performed with spam e-mails found at online spam repositories [Gra]. Non-spam e-mails were collected from the inboxes of the authors of the paper. When our system checks a given message using `checkSpam`, the result can be one of the four possibilities: *positive spam* $>= 5$, *high probability of spam* $>= 3$, *low probability of spam* $>= 1$, and *negative spam* $<= 0$. The accuracy test results are organized in accordance with these ranges.

We attempted to estimate how often the contact information is encountered in spam and in non-spam. Table 1 shows the average number of URLs, phone numbers and email addresses per message for both spam and non-spam, as computed from our databases. These don't include duplicates. The table also shows the total average number of contact information per message, for both spam and non-spam (including duplicates).

According to our system, only 8% of spam do not contain any contact information, whereas the same metric for non-spam is 30%. Also, the total count of contact information items (including duplicates) in spam is almost 6 times of that in non-spam. These numbers corroborate our original assumption that contact information is found in spam more often than non-spam.

The performance tests were conducted on 3 i386 machines running FreeBSD 4.5: a 1.2GHz AMD Athlon dual processor with 1 GB RAM, a 300 MHz P2/P2 Xeon/Celeron single processor with 400 MB RAM, and a 200 MHz Pentium Pro single processor with 64 MB RAM. Each server ran a Chord server, and the AMD Athlon ran SpamStrangler software.

4.1 Tests

The accuracy tests consisted of submitting random e-mails from our spam repository and running `checkSpam` on our full repositories of spam and non-spam e-mails. The tests reported the probability that an e-mail was a spam. Identical tests were performed by submitting 10%, 20%,

	# of mesgs	# w/ no contacts	Avg # of URL/mesg	Avg # of e-mail/mesg	Avg # of phone #/mesg	Avg total # of contacts/mesg
spam	6049	519	2.0372	1.82047	1.1101	18.3072
non-spam	3023	959	0.475356	1.28746	0.496857	3.65862

Table 1: Contact Information Analysis

30%, and 50% of spam randomly selected from the repository. A 100% test was also performed, but the submitted messages were obviously not random.

The performance tests were designed to calculate the time required to separately submit and check an e-mail. For this purpose, we artificially generated sets of emails with specified sizes and specified number of contact infos. The sets of e-mails consisted of 100 e-mails, whose bodies all differed from each other due to the insertion of different contact infos. All e-mails had different bodies in order to make sure that the SpamStrangler operations checked and submitted all the contact infos and sentences. The number of sentences in each e-mail was proportional to the size of the e-mail. For the sizes, values were chosen from 100 bytes, 1 Kbyte, 10 Kbytes, or 100 Kbytes. For the number of contact infos, 0, 6, and maximum allowed, 15 were used. The 100 byte messages with 15 contact infos were considered unrealistic due to their small size.

4.2 Analysis

The accuracy of detecting spam is summarized in Table 2. The most interesting result is a high detection rate of 76% while submitting only 10% of the spam e-mails – where our detection rate includes positive, high and low results. As the number of spams submitted increases, the detection rate does not increase drastically. The likely reason is that many of the unsubmitted messages in the 10% test are very similar to previously submitted messages; thus, the marginal contribution of submitting additional messages decreases with the number of messages submitted. Nonetheless, the number of spam messages caught always increases as the submission rate increases. Many of the negative were due to e-mail messages in MIME encodings that our parser does not yet support.

# of spams submitted	Positive	High	Low	Negative
470	610	1854	1115	1118
940	1124	1781	877	915
1409	1600	1559	773	765
2349	2520	1029	538	610
4697	4623	0	2	72

Table 2: Spam Check Accuracy

SpamStrangler's accuracy of avoiding false-positives is quite impressive as shown in Table 3. Up until the 100% submission rate, the false-detection rate (which includes the number of positives, highs and lows returned by `checkSpam`) is about 0.1%. In the case of 100% submission, we discovered that the high number of "highs" was due to a single message containing "http://web.mit.edu". It was submitted only during the 100% run and caused relatively many "high" results because our non-spam repository consisted of e-mails accumulated at MIT. Upon further analysis of the falsely detected e-mails, we found that many false detections (including the one "positive" result) consisted of e-mails from a legitimate mailing list that had been classified as spam by other users.

We also tested the detection accuracy of a version of SpamStrangler which does not utilize the message sentences. The results were very close to the ones presented in here.

# of spams submitted	Positive	High	Low	Negative
470	0	1	7	3015
940	0	0	27	2996
1409	0	1	38	2984
2349	0	1	33	2989
4697	1	12	210	2800

Table 3: Non-spam Check Accuracy

Tables 4 and 5 summarize the performance of checkSpam and submitSpam for different messages in the 3 Chord servers (1 per host) configuration. Even though we were unable to test our system on a real world WAN network, we still decided to use the maximum number of hosts available to us, so that we could observe more realistic timings that included the UDP/TCP stack and LAN latencies. As the number of SpamStrangler participants increases, we expect the network time to increase logarithmically, due to the underlying properties of Chord.

Our main goal was to decrease the number of Chord requests because we expect these requests to dominate the execution time of SpamStrangler. As described earlier in the paper, we achieved this by storing a limited number of SHA1 hashes for each submitted e-mail.

The total processing time of our system increases with the size of the e-mail because more information needs to be parsed. To decrease this effect, we assume that very long messages are unlikely to be spam because a large size makes a message harder to deliver and harder for people to read. For example, the average message size of our spam repository is approximately 9KB. Therefore, our system currently does not parse messages longer than 100KB.

The results in Tables 4 and 5 were obtained using SpamStrangler's blocking interface to better estimate the Chord lookup time.

We also estimated the average time per message when submitting our entire repository of spam emails. The average submit time using the non-blocking interface was 18.25 msec, and the average check time (using an empty Chord database) was 18.39 msec. The numbers were almost twice as high when performing this same test using the blocking interface, justifying our original hypothesis that the non-blocking interface is the one likely to be used in practice.

These results suggest that SpamStrangler's overhead is perfectly acceptable for mail clients. Our system performs a limited number of Chord lookups, which should take logarithmic time in the number of computers in the network. We also hope that with increasing network speeds the network time will not be a bottleneck in our system.

Though it might not be feasible for mail servers to parse the contents of large volumes of emails in order to detect spam, mail servers can still use a stripped-down version of SpamStrangler that only checks for matching email bodies. As our results showed, that test by itself can block a significant number of spam emails.

message size	0 contacts msec/msec	6 contacts msec/msec	15 contacts msec/msec
100b	3.28/4.01	10.63/125.6	N/A
1Kb	2.02/4.75	9.60/13.18	19.82/24.08
10Kb	2.23/19.44	9.23/31.97	19.29/43.00
100Kb	2.32/170.86	9.44/227.68	19.99/239.21

Table 4: checkSpam avg execution time per message (Chord lookup time / total time)

message size	0 contacts msec/msec	6 contacts msec/msec	15 contacts msec/msec
100b	3.24/3.96	22.29/25.24	N/A
1Kb	2.01/4.68	20.60/25.28	41.18/47.99
10Kb	2.26/19.08	19.56/43.29	41.43/67.42
100Kb	2.40/167.64	19.86/238.07	44.11/264.34

Table 5: submitSpam avg execution time per message (Chord lookup time / total time)

As far as the storage usage is concerned, SpamStrangler is quite efficient. For each submitted e-mail, SpamStrangler stores no more than 2KB worth of data. In fact, our spam repository occupies approximately 40 MB of space while SpamStrangler's database for all of the e-mails is only 1.3 MB. The average spam size is about 9 KB while the average storage space per message consumed by SpamStrangler is about 272 bytes.

5 Related Work

SpamBouncer is a collection of rules for filtering e-mail in procmail. It determines spam through IP address blocking, usage of mass-mailing programs, and text filtering. SpamBouncer has profiles for different spamming companies, which allows for custom filtering. A limitation of this system is that there are no methods for automatically updating the spam company's profile[Ham02]. A new spam e-mail that doesn't fit a regular spam company's profile will go undetected.

The Distributed Checksum Clearinghouse system (DCC) is a set of distributed servers that store fuzzy hashes of spam e-mails[Sof02]. A fuzzy hash maps similar e-mails to similar hashes. The DCC servers exchange fuzzy hashes in order to keep a distributed database. Hashes are deleted after a certain time period of infrequent use. Unfortunately, we could not find a description of DCC's fuzzy hash algorithm. A drawback of this system is that DCC servers can cause high network traffic by transmitting checksums to each other. DCC mitigates this problem by broadcasting only frequent hashes. Our implementation avoids this problem because Chord does not need to broadcast e-mail hashes to keep a distributed database.

SpamAssassin is a set of API methods that can be integrated with many mail systems such as procmail, qmail, Exchange, Outlook, and Eudora. To detect spam, SpamAssassin employs blacklist blocking, text filtering (including e-mail addresses of known spammers), and a spam database called Vipul's Razor[spa02]. SpamAssassin's use of Vipul's Razor is efficient in detecting identical spam messages; SpamAssassin also employs the DCC system to check for slight variations on e-mails. This system is ideal for integrating with SpamStrangler: SpamStrangler could use SpamAssassin's API methods to extract relevant spam information and store it on the database. On its side, SpamAssassin could take advantage of SpamStrangler's decentralized and efficient distributed database.

SpamCop is an e-mail provider that filters spam when checking e-mail. It converts each e-mail to a set of words and then performs a Bayes algorithm to determine if the words in the e-mail are likely to be from a spam[PL98]. SpamCop also does blacklist blocking; when it finds a spam, SpamCop puts the e-mail in a special folder[Hai02]. SpamCop has a 92% detection rate and a 1.16% misfire rate[PL98]. In order to check for spam, user e-mails are forwarded to SpamCop servers, where they can be filtered and read. Such e-mail forwarding is the major drawback of SpamCop because it generates additional network traffic. Additionally, users have less privacy in their e-mail accounts.

The Mail Abuse Prevention System, MAPS, provides a listing of IP addresses from known spam senders [MAPS01]. Other networks can then filter out e-mails from those IP addresses. There are two drawbacks from this system: first, IP addresses can be easily forged. Second, IP addresses cannot always be traced to individual users: in the case of ISP providers, the same IP address may be used by both legitimate users and spammers at different times.

6 Future Work

Future work includes support for deleting entries. A possible initial approach would be to associate a "time-last-accessed" and "number of accesses" along with each entry in the database. Entries could be deleted once they became "cold", that is, once they started being accessed less and less often. The specific "coldness" thresholds could be dynamically adjusted, based on the amount of data stored in the database.

Our initial implementation of contact info parsing was used to illustrate a proof of concept. Now that our initial tests show that contact info is effective in detecting spam, we plan to augment the Parser to more reliably extract contact information. Instead of extracting every phone, URL and e-mail address, the Parser will additionally take into account the location of these elements in the message. In particular, it would pay attention to phone numbers, URLs and e-mail addresses located close to sentences like "Call us today", "Visit us at", "Get rich by", etc.

Implementing a more elaborate Parser would not have given us better results at this stage, because the spam in our databases was not tailored to defeat our system–our false positives and misses were not due to unreliable contact info parsing. However, once an alpha version of the system is put in use, spammers will start using techniques to defeat our scheme. Gradually we will be able to collect a database of hard-to-detect spam that we can use to test more elaborate implementations of the Parser. Additionally, once we have a more reliable algorithm for extracting contact information from an e-mail, we plan to test our system with different parameters. For example, we might assign different weights to different types of matches in the database (i.e, a matching phone number could have a higher weight than a matching URL) in order to improve detection and reduce false alarm rates.

7 Conclusions

We presented SpamStrangler, a distributed spam-detection tool that relies on a distributed database of spam content. Our system uses primarily contact information present in the message in order to associate an e-mail with spammers. Our system also takes advantage of the similarity between sentences in different versions of the same spam e-mail. Finally, our system also stores a hash of the visible text content of the e-mail.

Our tests show that checking for identical e-mails, by itself, does not yield a high detection rate. A high detection rate, however, was obtained by using contact information. The reason is that spammers slightly modify their e-mails before sending them to different users, but they cannot alter their contact information. Usage of sentences did not provide significant improvements; however, we expect them to be important in the future, when spammers start writing e-mails to deliberately defeat the contact info technique. Performance tests showed that our system is efficient enough to be used in mail clients; on the other hand, a stripped-down version of the filter that only checks for e-mail bodies might be more appropriate for mail servers.

References

[Con02] World Wide Web Consortium. Libwww–the w3c protocol library. `http://www.w3.org/Library/`, June 2002.

[Gra] Paul Graham. Spam archives. `http://www.paulgraham.com/spamarchives.html`.

[Hai02] Julian Haight. Spamcop. `http://spamcop.net`, 2002.

[Ham02] Catherine A. Hampton. The spambouncer: a procmail-based spam filter. `http://spambouncer.org`, October 2002.

[MAPS01] LLC Mail Abuse Prevention System. Maps rbl end user information. `http://mail-abuse.org/rbl/enduser.html`, July 2001.

[Maz01] D. Mazieres. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Technical Conference*, 2001.

[PL98] Patrick Pantel and Dekang Lin. Spamcop: A spam classification & organization program. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin, 1998. AAAI Technical Report WS-98-05.

[Shi02] Maggie Shiels. Why one spam could cost $50. *BBC News*, April 2002.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[Sof02] Rhyolite Software. Distributed checksum clearinghouses. `http://www.rhyolite.com/anti-spam/dcc/`, October 2002.

[spa02] Spamassassin: Welcome to spamassassin. `http://spamassassin.org`, 2002.