Global-scale Anti-spam Testing in Your Own Back Yard

Margaret Nielsen
mfnielse@ucalgary.ca

Dane Bertram
dbertram@ucalgary.ca

Sampson Pun
szypun@ucalgary.ca

John Aycock*
aycock@ucalgary.ca

Nathan Friess
nfriess@ucalgary.ca

Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, Alberta, Canada T2N 1N4

Abstract

A class of anti-spam techniques requires large-scale, sometimes globally distributed networks of computers in order to operate successfully. Testing the effectiveness of these types of anti-spam systems – being able to run large controlled experiments – has not been well-explored in previous work on anti-spam testing.

To address this problem, we describe a system we have constructed to test global anti-spam techniques that is built on top of the Spamulator, a system that simulates relevant parts of the Internet on a single computer. We demonstrate the viability of our testing system by deploying over 10,000 SMTP servers, and using them to perform proof-of-concept experiments on SMTP tarpits and a variation on the Distributed Checksum Clearinghouse.

1 Introduction

> 'The requirement that we be willing to subject our explanations to experimental test is the distinguishing feature of science.'
>
> [8, page 7]

How do we scientifically determine how effective anti-spam techniques are? An approach involving experimental testing is called for, yet anti-spam testing in general poses a number of problems. For example, how is a realistic nonspam (ham) corpus collected while observing privacy concerns? What is really meant by "effective," when some anti-spam users may want to avoid false positives even at the cost of receiving more spam?

We focus on a different testing problem in this work. A number of anti-spam techniques are local, in the sense that they can be tested in isolation. Other anti-spam techniques are global, however, and only work on a very large scale, in a distributed fashion. It would seem that to test global-scale anti-spam techniques, we need our own internet.

In fact, the problem is even worse than that. The real Internet is a complex and constantly-changing environment. Setting up monitors on the Internet to watch spam activity is rife with pragmatic issues, like the technical, political, and legal headaches of deploying monitoring on a very large scale. Where to deploy monitoring for accurate data collection is another question – after all, a thermometer sitting on a heating vent is oblivious to the freezing weather outside.

The controlled experiments we want imply that somehow, we must control all these factors. Furthermore, for good experiments we must control the actions of spammers on the Internet and, if we could do *that*, we would have already solved the spam problem!

In the remainder of this paper, we discuss our preliminary work on a system that addresses these problems. This system, the "Spamulator," was actually developed for teaching university students how to send spam, of all things, but we have realized that it can be successfully applied to global-scale anti-spam testing. Such testing allows controlled experiments and measurements, and also permits new global-scale anti-spam measures to be tried and evaluated in an isolated test environment prior to being deployed in production.

*Corresponding author

Related anti-spam testing work is covered in Section 2; Section 3 describes the Spamulator. Section 4 presents the testing system that we built using the Spamulator, and we demonstrate its effectiveness with two experiments that are detailed in Section 5. Our conclusions and suggestions for future work follow.

2 Related Work

Cormack and Lynam's work in spam filter testing [4] is some of the most comprehensive evaluation work done to date. The experiments they performed on spam filters used their evaluation framework, which uses the messages from a given corpus and sends them through the filter being evaluated. The framework can also provide feedback to filters that can learn from mistakes. However, the testing framework is restricted to filters that operate independently of external systems. Systems that use wall clock time or distributed filtering are not testable in this framework. Our system is designed to handle large-scale distributed filtering as well as being able to support local filtering.

Sarafijanovic et al. [12] designed a framework which facilitates rapid deployment of multiple spam filters and simulated users over a network. This allows multiple instances of a spam filter to be deployed and tested simultaneously. However, they did not perform an evaluation of several filters simultaneously, so it is not clear how realistic (or useful) their simulated environment is. Compared to our system, theirs is also substantially more demanding in terms of resources: they recommend using either PlanetLab [11] or multiple virtual machines. Our Spamulator, on the other hand, runs on a single computer with no virtual machines required. This brings large-scale anti-spam testing into reach for anyone with a laptop.

The closest work to ours is Garcia et al. [7]. They built a simulator to test filters, trying to simulate the behavior of spammers, legitimate users, and legitimate mailing lists. Their system is highly abstracted, and is tightly coupled. In contrast, our system is realistic to the point where real, unmodified bulk-mailing software can run on it to send email.

3 The Spamulator

Details of our Spamulator system and its use in teaching have been covered elsewhere [3], but we give a brief overview here for completeness.

The Spamulator was originally designed and built for use in a course on spam and spyware offered at the University of Calgary [1]. Students do course assignments in a secure lab, isolated from the Internet for safety.

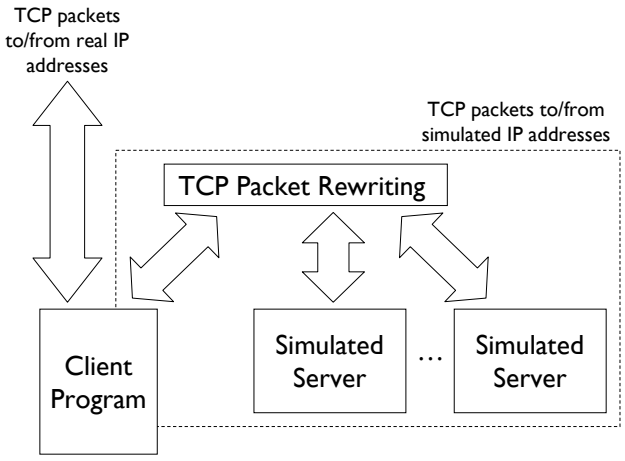

Figure 1: Spamulator architecture

We take a balanced, "hands-on" approach, with pairs of offense and defense assignments; for example, an offense assignment had students writing spam software, a defense assignment anti-spam software. The spam assignment was particularly problematic, though. The reason for this was that there weren't enough people to spam – a small, finite number of SMTP servers in the lab limited what students could do. As a result, the students' experience was not as realistic as it should have been.

This is solved by the Spamulator's simulation of relevant parts of the Internet. Running on a single machine,[1] it simulates up to a million domains' worth of web pages, SMTP servers, and open proxies. This allowed students to harvest email addresses and send spam within the confines of their own private internet. The network code they wrote could be in any programming language, and did not need to link with special libraries or make any concessions to the Spamulator. In fact, regular network programs like web browsers could use the Spamulator unmodified, and we ran unmodified, real bulk mailing programs to stress-test the Spamulator. Note that we only simulate higher-level services; low-level issues like network topology, latency, and transmission errors are outside the Spamulator's scope.

A high-level architectural overview of the Spamulator is shown in Figure 1. Simulated domain names are mapped into IP addresses by a local DNS server (not shown). TCP packets from a client on a machine to simulated IP addresses are rewritten, sending them back to the same machine. A simulated server handles the TCP traffic, pretending to be at the simulated IP

[1] Currently Linux, Mac OS, and FreeBSD versions exist; the anti-spam testing system we describe later used the Linux version.

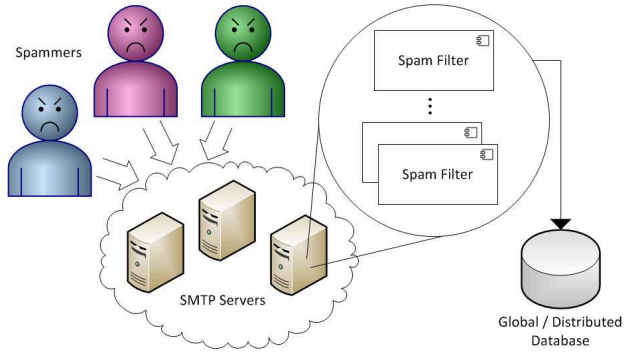

Figure 2: Testing system overview

address and port; one process is used per TCP connection. Return packets from the simulated server are also rewritten to complete the illusion, appearing as if sent from the simulated IP address.

Of particular importance to global anti-spam testing is the fact that the Spamulator has a well-defined mechanism through which it can be extended [2]. For example, a connection to the simulated IP address 10.0.0.42, port 25 causes the Spamulator to look in a directory for an executable named

```
10.0.0.42:25
```

and if found, invokes it with command-line arguments that reflect the simulated IP address and port.[2] This executable, after a brief handshake with the Spamulator, handles the connection as if it were the simulated machine. In this example, it would pretend to be an SMTP server at port 25. Appendix A contains the source code for a simple Spamulator extension.

The standard Spamulator installation we used in our spam lab had SMTP servers that, upon receiving a piece of email, simply discarded it. (Recall that we were using the Spamulator for a spam-sending assignment, so there was no point doing anything with the sent email.) However, we realized that the Spamulator's extension mechanism could be used to build a system for anti-spam testing, and the fact that the Spamulator was simulating the Internet meant that we could experiment with anti-spam on a global scale.

4 Anti-spam Testing System

At a high level, the testing system aims to simulate an environment of multiple SMTP servers not dissimilar to that found in the real world. Each of these servers can make use of a variety of spam filters, including those that employ distributed techniques for classifying mail (see Figure 2). Our testing system is described in the remainder of this section.

4.1 SMTP Server

The SMTP protocol [10] is a relatively simple, text-based protocol that uses a limited set of commands and reply codes for communication. We created a program which emulates an SMTP server within the Spamulator; this is distinct from the SMTP server described in the last section, because this server does not discard incoming email. A collection of soft links to this server were added to the Spamulator's extension directory, each link mapping a single IP address and port number to an instance of our SMTP server. We could easily emulate thousands of SMTP servers by simply creating additional soft links. Whenever a TCP connection on port 25 was made to one of our IP addresses, a single instance of the SMTP server was started. Once the SMTP transaction is completed the SMTP server process shuts down. For our experiments, only a subset of the SMTP protocol needed to be implemented (HELO, MAIL, RCPT, DATA, QUIT).

4.2 Mailer Process

For the purposes of our experiments we developed a simple bulk-mailing program to interact with our SMTP servers. This mailer program acts as both a spammer and as a regular sender of legitimate email, switching between the two roles by changing the email content. When acting as a regular sender, the content of the email is randomized to represent the variation in the contents of legitimate email. When acting as a spammer, however, the exact same content is sent out with each email to represent typical bulk-mailing practices. Our mailer process also allows sending email through SOCKS proxies to obscure the originating location of the sender, although we did not exploit this in the tests we describe here.

4.3 Spam Filtering Architecture

There were three major requirements for our system. The first requirement was that the system must be able to simulate filtering on a global scale. This meant that our implementation would need to allow the filters to communicate with other simulated servers (e.g., to coordinate filtering across multiple SMTP servers). The second requirement was that the implementation of each spam filter should not require the use of a specific programming language. It seemed logical to make this flexibility a requirement rather than restricting ourselves to only a single programming language. Finally,

[2] On Unix-based systems, this may seem redundant because the same information is available from `argv[0]`. However, we are only describing part of the extension protocol here; the arguments are needed for other cases.

IP Address	Injection Point	Path to Filter	Argument
10.125.0.1	RCPT	/filters/filter1	10.125.0.1
10.125.0.1	FROM	/filters/filter2	
10.135.0.2	DATA	/filters/filter3	300
10.135.0.2	RCPT	/filters/filter1	10.135.0.2

Figure 3: Sample filter configuration file

we needed a mechanism that would allow spam filters to be called from different places during the SMTP transaction. This requirement allows different types of filters to be called from their respective point(s) of interest.

Keeping these requirements in mind, we developed a plug-in system that executes each spam filter as an independent process. As independent processes, each filter can be written in any language that can be compiled and executed on the host system. Filters are able to act freely as local or global spam filters. The only expectation from the SMTP server is a verdict regarding the current email's classification (spam or ham).

There are three places within our SMTP server where an anti-spam filter can be called: after the MAIL command, after the last RCPT command has been completed, or after the DATA portion of the session has been successfully received. To facilitate spam filtering, three types of data are harvested from each SMTP session: the sender's email address (from the MAIL FROM command), the email recipients' addresses (from the RCPT TO commands), and the actual email contents (following the DATA command). This data is stored in three separate text files in order to make them easily accessible to the spam filter plug-ins.

The plug-in system is controlled by a filter configuration file, that is read by each SMTP server process during startup. This file specifies which filters need to be run at each filter injection point. The server will then run the specified filters and pass them information regarding the current SMTP transaction as command line arguments. These arguments are presented in the form of file paths to the temporary files that were created by the SMTP server. The filter is then free to use the information stored within these files to determine whether the current email should be considered spam.

An example of a filter configuration file is presented in Figure 3. Each line within this file represents a filter that will be run for a specific SMTP server. Each line begins with the IP address of the SMTP server this filter would like to be called from. The next argument specifies which injection point the filter should be called from (RCPT, [MAIL] FROM, or DATA). The complete path to the filter's executable is then provided. Finally, one optional argument may also be specified that will be passed into the filter. In order to allow a single spam filter to be called from more than one injection point, a given filter may be specified more than once within the configuration file.

The procedure followed by our SMTP server is depicted in Figure 4. First, a spammer connects to the SMTP server and begins the process of sending an email (1). During this process the pertinent information related to the email being sent is recorded to a number of files acting as storage buffers (2). At each of the filter injection points, filters that had been specified in the configuration file are executed with the command line argument pointing to the appropriate storage buffer (3). The spam filter then determines whether the current message should be classified as spam and returns its verdict to the SMTP server (4). If the email is classified as spam the connection with the sender is terminated, otherwise it is completed and treated as a successful delivery of that message. For later data analysis, we log errors, how many emails were attempted to be sent, the reject/accept verdict of each spam filter called, and the name of the spam filters executed.

4.4 Spam Filters

In order to show that the spam filtering architecture is capable of achieving our stated goals, we have implemented two distributed spam filtering techniques. Individual SMTP servers are converted into "tarpits," and a simplified Distributed Checksum Clearinghouse (DCC) filter is deployed within the Spamulator. As we discuss in the next section, these are meant *only* as proofs of concept to show that our system can be used for large-scale experiments.

4.4.1 Tarpits

An SMTP tarpit is a modified SMTP server that delays the communication of an incoming request for an extended period of time [6]. The assumption behind tarpits is that spammers send high volumes of email and that these tarpits can be used to slow them down and thereby diminish their profits. A discussion of whether or not this is a sound assumption is beyond the scope of this paper, but we will take it at face value for the purposes of our experiment.

Tarpits are specifically targeted at spammers since legitimate mailers should only be making use of proper SMTP servers. Should a legitimate mailer accidentally attempt to send mail through a tarpit they would also be delayed for a short period of time, but this should be tolerable due to their relatively low volume of emails. In principle, given enough tarpits (i.e., considering this

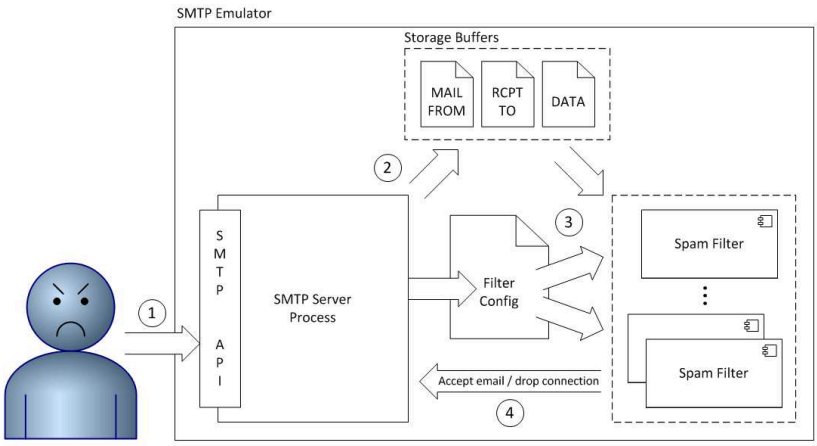

Figure 4: Spam filtering process

as a global-scale anti-spam technique) spammers could be significantly delayed.

Our proof-of-concept tarpit implementation causes the SMTP process to sleep temporarily after the receipt of each SMTP command. During the SMTP connection our tarpit will sleep for an aggregated time of 10 seconds. These tarpits will also reduce the TCP/IP window for data transfer down to a single byte to further delay spammers in sending out mass batches of email.

4.4.2 Distributed Checksum Clearinghouse

Another spam filtering technique requiring global scale is the Distributed Checksum Clearinghouse (DCC) [5]. DCC aids in identifying bulk email (not necessarily spam) by tracking how many times a given email message has been received on a global scale. This method attempts to exploit the fact that much of the spam being sent around the world is not being sent to only a single recipient. Much of the spam is being sent in bulk and DCC attempts to leverage this trait by maintaining a distributed hash table that maps the checksum of a given email message to the number of times that checksum has been reported. Mail handlers (SMTP servers, mail clients, etc.) wishing to make use of a DCC first calculate a checksum for each incoming message. Next, a DCC server is queried to find out how many other people have also received that message. If the count returned is above a predefined threshold the mail can be discarded or filtered as spam.

Our simplified DCC filter works by analyzing the data portion of each email it is asked to inspect. We implemented two different checksum algorithms, the first of which simply looks at the overall size of the message, and the second calculates the ELF-hash [13] of the message. From here the DCC client establishes a TCP connection with a running DCC server – another Spamulator extension – and sends the generated checksum as a request. An optional identifier can be included which allows multiple checksum algorithms to be tested without requiring a separate DCC server for each checksum scheme. The return value from the DCC server is the number of occurrences of that checksum the server has seen. A decision of how to classify the email is then based on this filter's second command-line argument which represents the spam threshold value. This value is defined in the filter configuration file as the optional argument for this filter, as described earlier.

5 Experiments

We conducted two experiments as proofs of concept to demonstrate our global-scale anti-spam testing system in operation. Note that these experiments are *not* indicative of the only experiments our system is capable of, nor are they intended to yield surprising results at this stage. We are only using them to demonstrate our system's ability to perform large-scale anti-spam experiments.

5.1 Tarpits

For the purposes of testing tarpits within our environment, we ran an experiment where a simulated spammer attempted to connect to an SMTP server selected at random from the list of our configured SMTP servers. This simulates a spammer sending mail to harvested email addresses in an undirected way, i.e., a typical case. We studied the effects of converting increasing percentages of SMTP servers into tarpits, with the goal being to identify the increased cost (cost=time in this scenario) to the spammer caused by the use of tarpits. While Hunter et al. have already conducted tarpit experiments [9], the purpose of our testing was

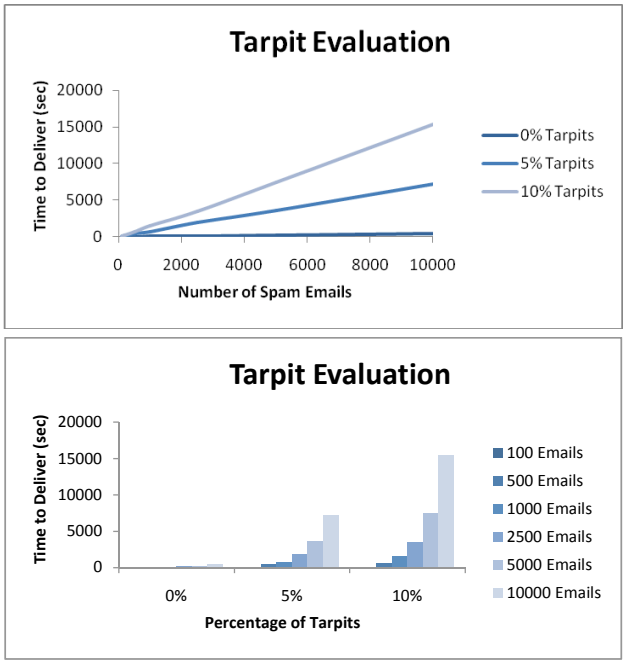

Figure 5: Tarpit test results

to demonstrate our system's ability to produce results for a very large-scale experiment that would be beyond the capability of most organizations to perform physically.

Our tests were conducted using 10240 SMTP servers within the Spamulator. We ran three versions of tests with 0%, 5%, and 10% of the SMTP servers converted into tarpits, respectively. Our mailing process then simulated a spammer by sending 100, 500, 2500, 5000, and 10000 emails over the network and recording the time it took to complete each batch of emails. These tests were repeated a number of times in order to allow the resulting data to normalize.

As seen in Figure 5, the amount of time it takes to send a given number of emails increase along a linear scale as the percentage of tarpits increases. It is interesting to note that with only 5% of our SMTP servers acting as tarpits we increased the cost of sending emails (in terms of time) by over 1300%. With 10% tarpits, the cost increased by over 2800%. However, even if the assumption underlying tarpits regarding spam-sending behavior were to hold, and it doesn't, having 10% of global SMTP servers being tarpits would be practically impossible. It is safe to conclude that tarpits are not an effective global solution.

5.2 Distributed Checksum Clearinghouse

Real DCC systems use "fuzzy" checksums so that, at least in theory, minor alterations to a spammer's message still yield the same checksum value. We conducted tests using two simple, non-fuzzy checksum algorithms: total message size and the ELF-hash of the first line of the email message. Note that the purpose of our testing was not to evaluate the effectiveness of these particular checksum algorithms, but rather to demonstrate the *ability* to test these algorithms easily within a controlled environment.

Table 1: DCC test results

Algorithm	Collisions
ELF-hash	2 (0.02%)
Message size	90 (0.9%)

In order to test these algorithms we configured 10240 SMTP servers within the Spamulator to use the DCC filter. We then modified our mailer process to send out 10000 "legitimate" emails. Although our mailer uses a randomly generated string of text for its messages, we could have just as easily sent actual emails read from a corpus of archived ham messages. Next, our DCC system was seeded with the checksum of a predefined spam message. Finally, we looked at the number of collisions encountered when sending our "legitimate" emails using each of our simplistic checksum algorithms.

The results (as shown in Table 1) indicate that these algorithms are not sufficient for use as DCC checksums, as we expected. Our test set of 10000 emails is not representative of the variations in emails sent throughout the world and yet we still have a substantial number of collisions. However, our goal was to demonstrate the ease with which such checksum algorithms could be tested; to test these different algorithms for DCC servers on a massive scale, only our filter configuration file needed to be modified.

6 Conclusion and Future Work

Our system provides researchers with a low cost, low risk way of testing anti-spam techniques. These techniques can range from local spam filters to complex global-scale distributed systems or any combination in between. We have demonstrated the flexibility and efficacy of our system by testing several sample configurations of tarpits and DCC filtering techniques.

There are a number of avenues for future work. While we have not addressed the problem of finding good (training and) test corpora that is common to anti-spam testing, an obvious improvement to our system would be to supplement it with good ham/spam corpora, or look at realistic synthetic traffic generation. In addition, a number of additional anti-spam tech-

niques, both local and global, can be implemented within our system.

Acknowledgments

Some of the authors have their research supported in part by grants and scholarships from the Natural Sciences and Engineering Research Council of Canada, the Informatics Circle of Research Excellence, and Alberta Ingenuity.

References

[1] J. Aycock. Teaching spam and spyware at the University of C@1g4ry. In *Third Conference on Email and Anti-Spam*, pages 137–141, 2006. Short paper.

[2] J. Aycock. Writing Spamulator extensions, 2007. Unpublished documentation.

[3] J. Aycock, H. Crawford, and R. deGraaf. Spamulator: The Internet on a laptop. In *13th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 142–147, 2008.

[4] G. V. Cormack and T. R. Lynam. Online supervised spam filter evaluation. *ACM Transactions on Information Systems*, 25(3), July 2007. Article #11.

[5] Distributed Checksum Clearinghouse, last accessed 29 January 2008. http://www.rhyolite.com/anti-spam/dcc/.

[6] T. Eggendorfer and J. Keller. Dynamically blocking access to web pages for spammers' harvesters. In *CNIS 2006*, pages 205–210, 2006.

[7] F. D. Garcia, J. Hoepman, and J. van Nieuwenhuizen. Spam filter analysis. In *18th IFIP World Computer Congress/TC11 19th International Information Security Conference*, pages 395–410. Kluwer, 2004.

[8] M. Goldstein and I. F. Goldstein. *How We Know: An Exploration of the Scientific Process*. Plenum Press, 1978.

[9] T. Hunter, P. Terry, and A. Judge. Distributed tarpitting: Impeding spam across multiple servers. In *Proceedings of the 17th Large Installation Systems Administration Conference*, pages 223–236, 2003.

[10] J. Klensin, ed. Simple mail transfer protocol, 2001. RFC 2821.

[11] PlanetLab, last accessed 28 January 2008. http://www.planet-lab.org/.

[12] S. Sarafijanovic, L. Hernandez, R. Naefen, and J. Le Boudec. Antispamlab – a tool for realistic evaluation of email spam filters. In *Fourth Conference on Email and Anti-Spam*, 2007.

[13] TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification, version 1.2, 1995.

A Sample Spamulator Extension

The C code for a sample Spamulator extension is below. It simply prints "Hello, world!" across the TCP connection to the client and exits. For brevity, all error checking code has been omitted.

The Spamulator holds a new TCP connection in limbo until it can start up a simulated server to handle the connection. Once started, the simulated server establishes a socket that it can listen for the new connection at, and passes that port number back to the Spamulator by writing it to the standard output. Then the Spamulator can let the new connection proceed, rewriting packets appropriately. The simulated server accepts the new inbound connection and begins its normal operation.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet.in.h>
#include <unistd.h>

int main()
{
    // Set up socket to listen at
    int sock = socket(PF_INET,
                      SOCK_STREAM,
                      IPPROTO_IP);
    listen(sock, 1);
    // Send port number to Spamulator
    struct sockaddr_in sin;
    socklen_t len = sizeof(sin);
    getsockname(sock, &sin, &len);
    unsigned short port = sin.sin_port;
    write(STDOUT_FILENO, &port, sizeof(port));
    // Finally, accept the connection
    int conn = accept(sock, NULL, NULL);
    // Now talking to client...
    write(conn, "Hello, world!\n", 14);
    return 0;
}
```