Addressing Email Loss with SureMail: Measurement, Design, and Evaluation

Sharad Agarwal — Microsoft Research
Venkata N. Padmanabhan — Microsoft Research
Dilip A. Joseph — U.C. Berkeley

***Abstract*— We consider the problem of *silent* email loss in the Internet, where neither the sender nor the intended recipient is notified of the loss. Our detailed measurement study over several months shows a silent email loss rate of 0.71% to 1.02%. The silent loss of an important email can impose a high cost on users. We further show that spam filtering can be the significant cause of silent email loss, but not the sole cause.**

***SureMail* augments the existing SMTP-based email infrastructure with a notification system to make intended recipients aware of email they are missing. A notification is a short, fixed-format fingerprint of an email, constructed so as to preserve sender and recipient privacy, and prevent spoofing by spammers. SureMail is designed to be usable immediately by users without requiring the cooperation of their email providers, so it leaves the existing email infrastructure (including anti-spam infrastructure) untouched and does not require a PKI for email users. It places minimal demands on users, by automating the tasks of generating, retrieving, and verifying notifications. It alerts users only when there is actual email loss. Our prototype implementation demonstrates the effectiveness of SureMail in notifying recipients upon email loss.**

I. Introduction

The Internet SMTP-based email system does not guarantee the timely or even eventual delivery of messages. Email can sometimes be delayed by hours or days, or even fail to be delivered to the recipient(s). Sometimes, the users are not even notified that their email was lost. Such *silent* email loss (i.e., the message is lost without a trace, not merely bounced back or misrouted to the junk mail folder), even if infrequent, imposes a high cost on users in terms of missed opportunities, lost productivity, or needless misunderstanding. Our SureMail system addresses this problem. Our targeting of silent loss is not fundamental. It is a trivial policy change to consider emails sent to the junk folder as lost email.

Recent measurement studies [15, 26] have reported email loss in the range of 0.5%-5%. We conducted a more thorough measurement study spanning months and find a *silent* loss rate of 0.71%-1.02%. While the lack of direct information from the email infrastructure makes it difficult to pin down the cause of email loss, we present evidence from one popular email service that points to spam filtering being the main cause.

From anecdotal evidence, we believe email loss also occurs elsewhere on the Internet, beyond the 22 domains in our experiment. Some users of EarthLink lost up to 90% of email silently in June 2006 [10]. AOL instructs users on what to do when email goes missing [3]. There are companies [8] that offer email monitoring services for businesses concerned about email loss.

Our measurement findings suggest that the existing SMTP-based email system works over 95% of the time. So our approach is to augment the existing system rather than replace it with a new one of uncertain reliability. SureMail augments the existing SMTP-based email delivery system with a separate *notification* mechanism, to notify intended recipients when they are missing email. By notifying the intended recipient rather than the sender, SureMail preserves the asynchronous operation of email, together with the privacy it provides. By having small, fixed-format notifications that are interpreted by email clients rather than being presented to users, we avoid the notification system from becoming a vehicle for malware such as spam and viruses as the current email system is.

Unlike some prior work, a key goal is for SureMail to be usable immediately by email users, without requiring cooperation from email providers. By not modifying the email infrastructure (including not altering spam filters), SureMail ensures against any disruption to email delivery that its installation might otherwise cause. Further, given its limited, albeit useful, functionality, a notification system would likely need less frequent upgrades (often disruptive) than a featureful email system.

We believe there is significant value in simply informing users that email to them was lost (i.e., is in neither their “inbox” nor “junk” folders). They can then contact the identified sender in a variety of ways to obtain the missing information (e.g., over email, different email accounts, phone, instant messaging).

We present two complementary approaches to delivering notifications: in-band delivery using email headers and out-of-band delivery using a web storage system. We have implemented both approaches and plan to make a SureMail add-in for Microsoft Outlook 2003 available. As in most P2P systems, both senders and receivers need to use it to benefit from SureMail. Our evaluation of the out-of-band approach shows that over 99.9976% of notifications are delivered successfully. We show that the incremental cost of SureMail is orders of magnitude lower than that of email, and thus we believe it is reasonable to deploy it for the added benefit of reliability.

We first proposed SureMail in a HotNets 2005 position paper[17]. Our design has since evolved considerably. The novel contributions presented here include:

- Measurement of email loss designed to avoid the shortcomings of prior studies.
- A redesign of SureMail to support in-band and/or out-of-band notifications, and to allow posting of notifications by legitimate first-time senders.

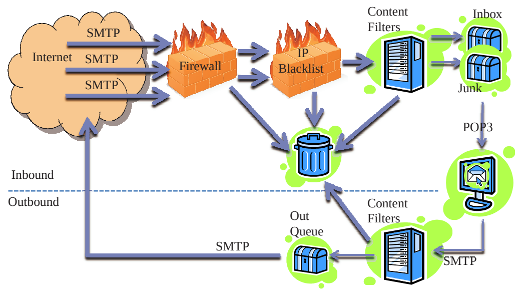

Fig. 1. A Typical Large Email Service Provider : top refers to inbound email, bottom to outbound email

- Implementation and evaluation of SureMail.

A. Outline

§ II describes a typical email service provider, various email filtering components and likely culprits of loss. This is followed by related work on email architectures and spam filter improvements. § III motivates the paper with our email loss measurement study, designed to avoid shortcomings of prior work [15, 26].

Given the significant amount of measured loss, we begin our design by first describing the *ideal* requirements of a solution in § IV. We strive to achieve these requirements in our design in § V. There are several challenges to be addressed within the security assumptions presented early in the section. § V-D presents a critical technique, *reply-based shared secret*, to prevent spammers from annoying users. § V-F explains how we distinguish some legitimate first-time senders from spammers.

SureMail allows notifications to be delivered in-band or out-of-band from email, or both in conjunction (§ VI). However, as with any new communication channel, the out-of-band technique introduces several challenges - our design is both low cost in terms of storage and message overhead, and resistant to security and privacy attacks. Our implementation and experimental evaluation of SureMail appear in § VII. We present a discussion of various issues pertaining to SureMail in § VIII.

II. Problem Background and Related Work

A. Typical Email Components and Email Loss

The typical email user may have a very simple view of their email system as that of their desktop client and the email server. In reality, large email providers tend to have complex architectures. Figure 1 provides a basic view of a typical provider. Each component in the figure may be replicated for load balancing, and some functionality may even be split among multiple devices.

The email system may be protected from malicious entities on the Internet by a firewall. It may block attacks, including excessive SMTP connections, SMTP connections with undesired options, connections with source addresses without reverse DNS entries, etc. The IP "blacklist" further blocks connections from certain IP addresses. This may include IP addresses of known spammers and open mail relays. Alternatively, a provider may use a "whitelist" policy where a source address may be automatically dropped if the volume of email sent by it falls below a threshold [4]. These connections are dropped without *any* inspection of the email content.

Any connections that pass through the firewall and IP blacklist are accepted and their email payloads are then processed by the content filters. These filters scan for viruses, worms and spam. Spam content can identified by a variety of techniques that involve comparing the text in the email to text from typical spam. Email that passes these filters will be stored in user inboxes. Email that is suspicious may be stored in junk mail folders instead. Email that is reliably identified as malware or spam may be thrown away without hitting any storage.

In the figure, only emails going to the "Inbox" or "Junk" folders are actually stored. Everything else is shown as going to the trash, which indicates that those emails are simply dropped. The major reason that not all emails are stored is a matter of volume. A major corporation's IT staff told us that about 90% of incoming email is dropped before it reaches user mail stores. Only the remaining 10% reach the inbox or junk folder.

Note that SMTP is not an end-to-end reliable protocol. Thus any of these components can temporarily fail due to overload, upgrade or maintenance and cause even more emails to be delayed or lost. Ever increasing volumes of spam and virus attacks make the infrastructure more susceptible to overload and failure.

In the outbound direction, email can also be lost. Emails composed by the user are typically sent via SMTP to the Internet. Some large email providers apply content filtering on outbound emails before they go to the Internet to filter out malware and spam that their users may be sending. This is to deter spammers who obtain email accounts on these providers in violation of the user agreement. Also, if too much malware or spam is sent by a particular provider, other providers may add that provider to their IP blacklist (or remove from their whitelist). Sometimes, email sent by travelers can be lost if they are forced to use a hotel's SMTP server that is not on the whitelists of destination SMTP servers.

Given such extensive filtering, it is not surprising that some legitimate email gets discarded entirely, not merely misrouted to the recipient's junk mail folder (we do *not* consider the latter as email loss).

SMTP allows a server to generate non-delivery messages ("bouncebacks") for emails it cannot deliver. This would only occur for emails where the incoming SMTP connection is accepted and parsed. In Figure 1, any drops by the firewall or IP blacklist would not cause any bouncebacks to be generated. For the emails dropped by the content filters, several issues reduce the effectiveness of bouncebacks in making email loss non-silent: (i) Typical content spam filters do not generate bouncebacks. (ii) Bouncebacks may be sent to spoofed source addresses, leading to user apathy toward such messages, or worse, to their classification as spam. (iii) Bounceback generation may be disallowed for privacy (e.g., to avoid leaking information on the (in)validity of email addresses). (iv) Servers sometimes (e.g., after a disk crash) do not have enough information to generate bouncebacks. (v) Bouncebacks cannot warn users about emails lost from a email server to a client.

B. Prior Work on Email Unreliability

We are aware of two recent measurement studies of silent email loss. Afergan et al. [15] measured silent email loss by recording the absence of bouncebacks for emails sent to non-existent addresses. 60 out of the 1468 servers measured exhibited a silent email loss rate of over 5%, with several others with more modest but still non-negligible loss rates of 0.1-5%. However, a shortcoming of their methodology is that bouncebacks may not reflect the true health of the email system for normal emails and many domains do not generate bouncebacks.

Lang [26] used a more direct methodology to measure email delays and losses. 40 email accounts across 16 domains received emails over a 3-month period. Their overall silent email loss rate is 0.69%, with it being over 4% in some cases. While that study does not depend on bouncebacks, it may be biased by the use of a single sender for all emails and the use of very atypical emails (no email body; subject is a message sequence number) that could significantly bias these being filtered as spam. Our study addresses some of these shortcomings.

To put these findings in perspective, even a silent loss rate around 0.5% (1 lost email among 200 sent, on average) would be a serious problem, especially since a user has little control over which emails are lost.

Prior proposals to address the email unreliability problem range from augmentating the current email system to radical redesign. The message disposition notification mechanism [21] (i.e. "read receipts"), enables senders to request that the recipient send an acknowledgment when an email has been downloaded or read. We believe that most users do not enable this feature in their email clients as it exposes too much private information — when the user reads reads email, conflicting with the inherent "asynchronous" use of email. Read receipts also enable spammers to detect active email accounts.

While re-architecting the email delivery system to enhance reliability (e.g., POST [5, 27]) is certainly desirable, that alone will not solve the problem because of loss due to spam filters. Although a public key infrastructure (PKI) for users, as assumed by POST, can help with the spam problem, it can be an impediment for deployment. In contrast, SureMail does not modify the underlying email delivery system and keeps the notification layer separate. This avoids the need to build (or modify) the complex functionality of an email delivery system and ensures that even in the worst case, SureMail does not adversely affect email delivery.

C. Spam Filters and Whitelisting

Improved spam filtering techniques ([9, 11, 13]), reduce false positives while still doing effective filtering. However, it is difficult to entirely eliminate false positives as spam constantly evolves to mimic legitimate traffic. Very high spam volumes often necessitate content-independent filtering (e.g., IP blacklisting) to reduce processing load on email servers.

Whitelisting email senders (using social relationships or otherwise [19, 23]) to bypass spam filters is complementary to SureMail. SureMail tries to notify recipients upon email loss, regardless of the cause, without actually preventing the loss. Whitelisting seeks to prevent email loss specifically due to spam filtering. Thus it needs to operate on email before it hits the spam filtering infrastructure. This requires the cooperation of the email administrators and convincing them that this modification to their servers will not negatively impact email delivery. In contrast, SureMail leaves the email infrastructure untouched, and allows individual users to start using the system without involving their email administrators. Finally, if a trusted sender's computer is compromised, potentially harmful emails may be whitelisted through the filtering infrastructure. In SureMail, this compromise only results in bogus notifications being delivered. We rely on human involvement for conveying the missing information, because email loss is relatively rare.

III. Email Loss Measurement

We begin by quantifying the extent of email loss in the existing email system. Due to privacy issues and the difficulties of monitoring disparate email servers, we resort to a controlled experiment where we send all the email, like [26]. However, we improve on their study by using multiple sending accounts, more realistic email content, and shedding light on the causes of email loss.

A. Experiment Setup

1) Email Accounts: To measure email loss on the Internet, we obtained email accounts on several academic, commercial and corporate domains (see Table IV). The non-academic domains include free email providers, ones that charge us for POP or IMAP access, and a private corporation. The domains are spread across Australia, Canada, New Zealand, UK and USA. In most cases, we obtained two mailboxes to catch cases where accounts on the same domain are configured differently or map

```
1) Seed random number generator
2) Pick a sender email address at random
3) Pick a receiver email address at random
4) Pick an email from corpus at random
5) Parse email and use the subject and body
6) With 30% probability, add an attachment, selected at random
7) If such an email (sender, receiver, subject, body, attachment)
   has not been sent before, send
8) Log sent email and any SMTP error codes
9) Sleep for a random period under a few seconds
10) Go back to step 2
```

Fig. 2. Pseudo-code for Sending Process

to different servers. Most systems allowed us to retrieve emails over POP3, IMAP, Microsoft Exchange, or RPC over HTTP. Many allowed us to programmatically send emails using SMTP. Overall, we have 46 email accounts: 44 allow receiving email, and 38 allow sending.

2) Email Content: We programmatically send and receive emails across these 46 accounts. To mimic content sent by real legitimate users, we use the “Enron corpus”, a large set of emails made public during the legal proceedings against the Enron corporation. We obtained a subset of this corpus [12] containing about 1700 messages manually selected for business-related content, while avoiding spam. Of these, we use a subset of 1266 emails with unique subjects, which facilitates the subsequent matching of sent emails with received emails. We use only the body and subject from the corpus and ignore the rest of the header.

We do not use any attachments from the corpus for fear that sending malware might bias our findings. To understand the impact, if any, of attachments on email loss, we picked 16 files of 7 different formats and various content : marketing, technical, and humorous materials (see Table III). The largest is about 105 KB since we do not want to overburden the hosting email domains. We did not include executables and scripts, since they increase loss due to virus and trojan scanners – at least one of the domains drops all emails with executable attachments. We do not attempt an exhaustive study of email loss due to attachments, but instead estimate if typical attachments influence the observed loss rate.

3) System Setup: We use the sending process in Figure 2. It is codified in a Perl program which uses separate C programs that handle SMTP connections for sending emails. We bias the sleep period in step 9 to not violate the daily volume limits in most account agreements. While most accounts have very similar limits, the msn.com and microsoft.com accounts allow us to send and retrieve almost 10 times more emails. Thus steps 2 and 3 are appropriately biased to more frequently send to and from these 6 accounts.

To retrieve emails, we configured Mozilla Thunderbird 1.5 to download emails from all receiving accounts. We download emails from the inbox of each account, as well as any junk mail or spam folders. Whenever allowed, we configured the accounts to disable junk mail filtering and/or created a whitelist with all 46 account addresses. We use Windows XP SP2 machines located on Microsoft’s network to send and retrieve email.

Once an experiment has completed running, we use a Perl program to parse the sending logs and feed them into a Microsoft SQL Server 2005 database. A second Perl program parses the Thunderbird mail files and feeds the retrieved emails into the same database. The program also attempts to parse the contents of any bouncebacks to determine which original email bounced. However, in some cases, not enough information is present in the bounceback to uniquely identify the lost email or the format of the bounceback is atypical and difficult to parse. We issue SQL queries to match sent emails with received emails, and calculate email loss statistics. The matching is done based on the following fields: sender email address, receiver email address, subject, attachment name. We do not use the body of the email for matching, because some email providers (e.g. Yahoo) insert advertisements. Hence our corpus consists of emails with unique subjects.

B. Email Loss Findings

We conducted three separate email loss experiments, summarized in Table I. The setup for experiments #1 and #2 is slightly different and is described in a technical report [16]. In this paper, we focus on the latest study, #3. We received more emails than we sent, primarily due to spam and domain announcements. Our SQL queries for matching sent emails with received emails ignore these extraneous emails. The overall loss rate is about 1.79%. We received about 1216 bouncebacks, with various status codes and reasons, not all of which we could accurately match to the sent email. 10 pairs of senders and receivers were unable to exchange any email during our experiment. These constitute the 363 “hard failures”. If we remove these unusual hard failures, and count all bouncebacks as successful notifications of delay or loss, we have a conservative *silent* loss rate of 1.02%. It is difficult to pin down the exact cause of each loss due to the opacity of so many email domains. However, we later attempt to identify the possible causes.

The silent loss rate appears to have increased slightly over time across the three experiments. We speculate that spam filters have had to more aggressively adapt to increasing volumes of spam. In private communication, two of the email providers confirmed that they updated spam filters almost continuously during this period.

Our experiment was biased toward sending and receiving more emails via the msn.com and microsoft.com accounts due to the higher allowed limits. If we remove these 6 accounts completely from our analysis, we have the results shown in Table II. This overall loss rate of

	Exp. 1	Exp. 2	Exp. 3
Sending accounts	36	36	38
Receiving accounts	42	42	44
Emails in corpus	1266	1266	1266
Attachment probability	0.3	0.3	0.3
Start date	11/18/05	01/11/06	09/06/06
End date	01/11/06	02/08/06	10/04/06
Days	54	29	29
Emails sent	138944	19435	203454
Emails received	144949	21015	213043
Emails lost	2530	653	3648
Total loss rate (lost/sent)	1.82%	3.36%	1.79%
Bouncebacks received	982	406	1216
Hard failures	565	70	363
Conservative silent loss rate	0.71%	0.91%	1.02%

TABLE I. Email Loss Statistics

Emails sent	88711
Emails lost	1653
Total loss rate	1653/88711 = 1.86%

TABLE II. Loss Statistics w/o msn.com and microsoft.com

1.86% is very similar to the 1.79% rate from Table I. So we believe our findings are not biased by the higher sending and receiving rates for these 6 accounts.

C. Detailed Findings

We now present detailed loss statistics for experiment 3, broken down by attachment, email body and email account. We only consider overall loss rates since matching bouncebacks to the specific email sent is difficult.

1) Loss by Attachment: Table III presents the loss by attachment. We want to estimate if the type of attachment or its content dramatically influences loss. We had 16 attachments of 7 types. For instance, we had 2 GIFs – home_main.gif and phd050305s.gif – and emails that included them suffered loss rates of 1.68% and 1.99%, respectively. While we do not observe a significant deviation from the overall loss of 1.79%, HTML attachments did suffer higher loss. We speculate that since HTML is becoming a more popular email body format, content-based spam filters are more actively parsing HTML.

2) Loss by Email Subject / Body: Figure 3 plots the loss rates, sorted from high to low, for our 1266 distinct email subjects/bodies. We see that some emails have significantly higher loss rates than others. Most of the email bodies with loss rates above 10% appear to contain business proposals, Enron-related news, and stock details. Even if we ignore these email bodies, most of the rest of the corpus does have a non-negligible loss rate. Thus, our findings are not a result of a few "bad apples" in our corpus, but due to a more general problem.

3) Loss by Account: Table IV shows the total loss rates for each account on each domain. Column 4 lists the aggregate number of emails sent to each account from all of the 38 sending accounts. Column 4 lists the loss rates experienced by these set of emails. Column 5 presents the aggregate number of emails sent from each of these accounts to all of the 44 receiving accounts. The last

Attachment Type	Emails Sent	Loss %
(none)	133198	1.56
2 * JPEG	2322, 4656	1.55, 2.10
2 * GIF	4597, 4532	1.68, 1.99
3 * HTML	4808, 4733, 4768	4.53, 3.53, 4.28
3 * MS DOC	4658, 4582, 4716	2.36, 1.88, 1.68
2 * MS PPT	4596, 2363	1.59, 1.78
2 * PDF	4670, 4808	1.56, 1.71
2 * ZIP	4749, 4698	1.41, 1.43

TABLE III. Email Loss Statistics by Attachment

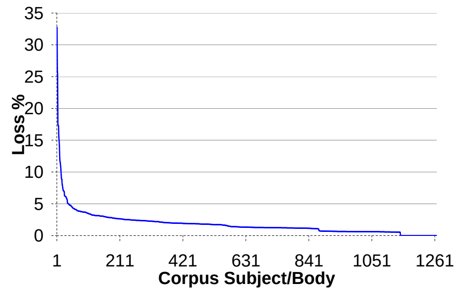

Fig. 3. Loss Statistics by Email Subject and Body

column is the loss experienced by these emails. We make two observations here. First, our overall loss rates are not influenced by a few bad domains or accounts — although a few accounts experienced no email loss, there is loss across most domains and accounts. Second, within each domain, both accounts tend to suffer similar loss rates [1].

4) Cause of Email Loss: In general, it is difficult to exactly determine the cause of each instance of loss because of the complexities of the myriad of email systems and our lack of access to the innards of these systems. We speculate that the likely causes of loss are aggressive spam filters and errors in the forwarding or storage software and hardware.

We focus here on the 4 msn.com accounts, for which we obtained special access. We disabled the content filters (see Figure 1) for $c@msn.com$ and $d@msn.com$. Any incoming emails that the content filters would have thrown away are "tagged", and the number of such emails are shown in column 4 of Table V. The $a@msn.com$ and $b@msn.com$ accounts are regular accounts [2]. The "Loss %" column shows the overall loss rate, while the last column shows what the rate would have been had the "tagged" emails been lost. The regular loss rates for c and d are relatively small, but the "tagged loss" rates for c and d are similar to the regular loss rate for a and b. So we hypothesize that content-based spam filters are the main cause for email loss for msn.com. However, they are not the sole cause, as indicated by the non-zero "Loss %" column for c and d. Due to this and the lack of perfect

[1]The second account at fusemail.com suffered a high receive loss rate. No emails were delivered between 09/30/2006 and 10/02/2006. Technical support was unable to determine the cause of the loss. We suspect server failure or loss between the server and our client.

[2]Note that we still treat any emails sent to the "Junk" folder for any account as though they were delivered to the Inbox and not lost.

Domain	T	Sent to		Recvr loss%		Sent from		Sendr loss %	
		A	B	A	B	A	B	A	B
aim.com	I	2306	2370	3.82	4.35	568	174	0.70	0.00
bluebottle.com	P	2336	2454	0.04	0.08	2446	2452	1.59	1.39
cs.columbia.edu	I	2338	2387	7.83	8.09				
cs.princeton.edu	P	2416	2341	0.29	0.04				
cs.ucla.edu	P	2346	2320	0.94	1.03				
cs.utexas.edu	P	2387	2395	1.01	1.29	3669	3628	2.89	3.03
cs.wisc.edu	I	2386	2350	0.54	0.21	3754	3592	2.08	2.31
cubinlab.ee.mu.oz.au	P	2341	2408	0.04	0.08				
eecs.berkeley.edu	I	1193	2321	4.36	3.62	1893	3688	2.11	2.03
fusemail.com	I	2425	2350	0.00	20.26	3740	3525	2.65	1.53
gawab.com	P					3652	3666	9.94	9.36
gmail.com	P	2313	2369	3.59	3.29	3680	3595	2.20	1.81
microsoft.com	E	19330	18504	1.77	1.62	35079	35157	0.62	0.65
msn.com	H	19465	18932	3.25	3.02	6757	6775	0.92	1.05
msn.com	H	19122	19390	0.37	0.41	6807	6793	1.23	1.06
nerdshack.com	P	2387	2305	0.00	0.17	3651	3588	4.14	4.26
nms.lcs.mit.edu	P	2421	2389	0.00	0.00	3649	3657	1.12	1.12
ulmail.net	P	2378	2360	0.00	0.00	3715	3697	5.49	5.63
usc.edu	P	2314	2371	0.04	0.00	3608	3731	1.36	1.07
yahoo.com	P	2324	2355	2.54	1.10	3835	3630	0.81	0.94
yahoo.co.uk	P	2442	2323	0.00	0.00	3474	3577	1.07	1.06
cs.uwaterloo.ca	P	2464	2323	1.87	1.89	3667	3602	1.77	1.30

TABLE IV. Loss by Email Account; numbers missing where programmatic sending/retrieving not allowed; T=Protocol type: E:Exchange,H:RPC/HTTP,I:IMAP,P:POP3; A=1st account, B=2nd

Receiver	Sent	Matched	Tagged	Loss %	Tagged loss %
a@msn.com	19465	18833		3.25	
b@msn.com	18932	18361		3.02	
c@msn.com	19122	19052	503	0.37	3.00
d@msn.com	19390	19311	531	0.41	3.15

TABLE V. Loss Statistics to msn.com

spam filters, there is value in email loss notification.

To summarize our experiment, we found significant total loss rates of 1.79% to 3.36%, with silent loss rates of 0.71% to 1.02%. That is, if a user sends about 30 emails a day, over the course of a year, over 3 days worth of emails get silently dropped. Based on our detailed results, we believe our findings were not biased significantly by the choice of attachments, message bodies and subjects, email domains, and individual accounts within the domains. Thus we believe that a system for addressing lost email will be of significant benefit to users. Our measured loss may not correspond exactly to the typical user experience, because:

– The 46 accounts on 22 domains in our experiment represent a small fraction of the worldwide email system.

– Our mix of intra- and inter-domain emails, the sending and receiving rates per account, and even the content (despite being derived from a real corpus), may not match user workload.

– In our experiments, all email addresses had previously emailed each other, and thus none was a "first-time sender" to any. First-time senders may experience higher loss, but despite that we found significant loss.

– A user may not be aware that their email was lost. Our experiment avoids this uncertainty by controlling both the senders and the receivers.

IV. Design Requirements

We believe the following properties of a solution to email loss will lead to rapid adoption and deployment.

1) **Cause minimal disruption:** Rather than replace the current system, which works for the vast majority of email, with a new system of uncertain reliability, augment it to improve reliability. It should inter-operate seamlessly with the existing email infrastructure (unmodified servers, mail relays, etc.), with additions restricted to software running outside it (e.g. on end-hosts). Users should benefit from the system without requiring cooperation from their email domain administrators.

2) **Place minimal demands on the user:** Ideally, user interaction should be limited only to actual instances of email loss; otherwise, he/she should not be involved any more than in the current email system.

3) **Preserve asynchronous operation:** Email maintains a loose coupling between senders and recipients, providing a useful "social cushion" between them. The sender does not know whether or when an email is downloaded or read. Recipients do not know whether a sender is "online". Such asynchronous operation should be preserved, unlike in other forms of communication such as telephony, IM, and email "read receipts".

4) **Preserve privacy:** The solution should not reveal any more about a user's email communication behavior than the current system does. For instance, it should not be possible for a user to determine the volume or content of emails sent/received by another user, the recipients/senders of those emails, how often that user checks email, etc. However, as it stands today, email is vulnerable to snooping, whether on the wire or at the servers. We do *not* seek to rectify this issue.

5) **Preserve repudiability:** Repudiability is a key element of email and other forms of casual communication such as IM [14, 18]. In the current email infrastructure, a receiver can identify the sender from the header, but cannot *prove* the authorship of the email to a third-party, unless the sender chose to digitally sign it. Any solution to email loss should not *force* senders to sign emails or facilitate receivers in proving authorship. As an analogy, people are often more comfortable identifying themselves and communicating sensitive information in person than in written communication, since the latter leaves a paper trail with proof of authorship to a third party. Note that PKI or PGP based authentication of email users, is unsuitable from the viewpoint of providing repudiability.

6) **Maintain defenses against spam and viruses:** It should be no easier for spam or viruses to circumvent existing defenses or tell if an email address is valid.

7) **Minimize overhead:** The solution should minimize network and compute overheads from additional messaging (e.g. sending *all* emails twice would significantly overload some email servers and network pipes).

V. SureMail Design

SureMail is designed to satisfy the requirements listed above. We continue to use the current email system for message delivery, but augment it with a separate, low-cost *notification* system. When a client sends an email, it also sends a notification, which is a short, fixed-format fingerprint of the email. We consider various notification delivery vehicles in Section VI, including email headers and a web service. If email loss occurs, the receiving client gets a notification for the email but not the email itself. After waiting long enough for missing email to appear (Section VIII-A), it alerts the recipient user, informing him/her about email loss from the sending user specified in the notification. SureMail does not dictate what action the recipient user should take at this point. The user may ignore it or contact the sender via email, IM, or phone, presenting the fingerprint in the notification to help the sender find the lost email.

The bulk of the work (creating, posting, checking, and retrieving notifications) is done automatically by the SureMail software. The user is involved only when a lost email is detected. SureMail allows a user to determine if he/she is missing any emails sent to him/her. It does *not* notify the sender about the status of email delivery. Thus SureMail assures senders that either email is delivered or the intended recipients will discover that it is missing. While the concept is simple, there are several key challenges that SureMail must address:

1) Prevent notifications themselves from being used as a vehicle for spam or malware: We use very short, 64-byte, fixed-format notifications (Figure 5), which are interpreted by the SureMail client rather than being presented directly to the user. Section V-C has the details.

2) Avoid the need to apply spam filters on notifications: Unlike email spam, notification spam does not directly benefit the spammer because of the restrictive nature of notifications noted above. However, to block bogus notifications, which could annoy users (e.g., by alerting them to the "loss" of non-existent email or spam), we present our reply-based shared secret technique in Section V-D.

3) Prevent or minimize information leakage from notifications: Information such as the fact that a particular email address is active or that user x has emailed user y could be sensitive even if the email content itself is not revealed. Section VI addresses these problems.

A. Security Assumptions

1) We assume that when a recipient chooses to reply to an email, they are implicitly indicating that the sender of the email is legitimate. I.e., users are very unlikely to reply to spam. We consider this unlikely possibility of a user being tricked into replying to spam in Section V-E.

2) We assume that the attacker cannot mount a man-in-the-middle attack (intercept and modify emails exchanged by a user). While we do not rule out the possibility of eavesdropping, we believe that in practice even this would be hard: an attacker would require access to the path of (remote) users' email. Furthermore, if an attacker does gain access to user email, that compromises user privacy more directly than subverting notifications.

3) If a separate infrastructure is used to deliver notifications, it may not be entirely trustworthy. The notification infrastructure may *try* to spy on users' email activity (e.g., who is emailing whom) or generate bogus notifications. Bogus notifications are a more serious problem than dropped notifications (which the notification infrastructure can always cause), since the former imposes a cognitive load on users while the latter leaves users no worse off than with the current email system.

B. Notation

Unless otherwise stated, S and R represent the sender and the recipient clients (and users) of an email. We assume that all nodes agree on: **(i)** a cryptographic hash function $H(X)$ that operates on string X to produce a short, fixed-length digest (e.g., a 20-byte digest with SHA1), and is pre-image and collision resistant; **(ii)** a message authentication code (i.e., a keyed hash function), $MAC_k(X)$, that operates on string X and key k to produce a short, fixed-length digest (e.g., a 20-byte digest with HMAC-SHA1). A MAC can be used with various well-known keys (e.g., k') to generate new hash functions (e.g., $H'(X) = MAC_{k'}(X)$); **(iii)** a symmetric encryption function, $E_k(X)$; **(iv)** a digital signature scheme to produce a signed message $SIG_k[X]$ using private key k.

C. Notification Basics

A notification is a short, 64-byte, fixed-format structure interpreted by software (not read by humans). This is in contrast to rich and extensible email (e.g. attachments, embedded HTML) that is often orders of magnitude larger. Consequently, it is hard for an attacker to send malware or spam in notifications. This makes it easier to reliably deliver notifications compared to email.

To identify an email in its notification, we cannot use the `Message-ID` field contained in some email headers because it may be set or reset beyond the sending client (e.g., by a Microsoft Exchange server), making it inaccessible to the sender. So we embed a new `X-SureMailID` header with a 20-byte SureMail message ID ($smID$), which is unique with high likelihood.

D. Reply-based Shared Secret

A notification also needs to identify the sender S, so that R knows who to contact for the missing email. However, we cannot simply insert S's email address into the notification since that can be easily spoofed. We instead use our reply-based shared secret scheme which blocks spoofed or bogus notifications, protects the identity of S, and needs no user involvement. It automatically establishes a shared secret between two users who have emailed each other before. We consider the first-time sender (FTS) case in Section V-F.

Say S sends an email M_1 to R and receives a reply $M_1^{'}$ from R. The SureMail client software at S and R

uses the corresponding SureMail message IDs, $smID_{M_1}$ and $smID_{M_1'}$, to perform a Diffie-Hellman exchange and set up a shared secret, $smSS_1$, known only to S and R. $smSS_1$ is computed and remembered by R's SureMail client automatically when user R replies to the email (recall assumption 1 from Section V-A) and by S's client when it receives the reply. S can then use $smSS_1$ to authenticate notifications it posts to R and to securely convey its identity to R (and R alone). Note that R would have to establish a separate shared secret with S, based on an email exchange in the opposite direction, for notifications that R posts to S. The notification for a new message, M_{new} from S to R is constructed as follows:

$$N = \{T, H(smID_{M_{new}}), H(smSS_1), \\ MAC_{smSS_1}(T, H(smID_{M_{new}}))\}$$

T is a timestamp. $H(smID_{M_{new}})$ identifies the new message. $H(smSS_1)$ implicitly identifies S to R, and R alone. $MAC_{smSS_1}(T, H(smID_{M_{new}}))$ proves to R that this is a genuine notification from S with an untampered timestamp, since only S and R know $smSS_1$.

Upon retrieving a notification, the SureMail client at R checks to see if it is recent (based on T) and genuine (i.e., corresponds to a known shared secret). If it is and the corresponding email is not received soon enough (see Section VIII-A), it alerts user R and presents S's email address. Old and invalid notifications are ignored.

When an email is sent to multiple recipients, a separate notification is constructed and posted for each using the respective reply-based shared secrets.

1) Shared Secret Maintenance: The SureMail clients at S and R perform a Diffie-Hellman exchange to establish a shared secret only if each was the sole addressee in an email exchange. So emails sent to multiple recipients and those send to mailing lists are excluded.

Each node remembers two sets of shared secrets for each correspondent: one for posting notifications and the other for validating received notifications. These are updated with each new email exchange. S remembers the smIDs of all messages sent by it since the most recent one replied to by R. Likewise, R remembers the smIDs of all emails from S that it had replied to (or just the smSSs derived from such emails), since the most recent one that S used as a shared secret in a notification.[3]

This constant renewal allows the shared secret to be reestablished if the user starts afresh, say after a disk crash. It also helps purge a bogus shared secret, such as when a spammer tricks R into responding to a forged email spoofed to be from S (see Section V-E).

2) Reply Detection: To help a client determine that an incoming email is a reply to a prior outgoing email, we include an `X-SureMailInReplyTo` (smIRT) header to echo the smID of the original email. While similar, we cannot use the "In-Reply-To" header included by most email clients because the message ID of the original email may not be available (see Section V-C).

[3]R could instead remember a few older smSSs as well to accommodate the possibility of an old notification, constructed using an older smSS, being reordered and delivered late. Given the (slow) human timescale of the email-reply cycle that generates a new smSS, remembering just a few recent smSSs should be sufficient.

E. Security Properties

The Diffie-Hellman exchange ensures that the shared secret, $smSS$, between S and R is not known to an attacker A that eavesdrops on the email exchange or on the notification. A cannot learn who posted a notification (thereby preserving privacy) or post fake notifications deemed as valid.

However, consider a more difficult attack. A sends R a spoofed email that appears to be from S and is also realistic enough that user R is tricked into replying, thus making R remember a bogus shared secret. If A also eavesdrops on the reply, it can learn this shared secret and then post bogus notifications. However, even if A manages to pull off this attack once, the bogus shared secret gets flushed out with renewals (Section V-D.1).

Unlike PKI/PGP-based systems, our reply-based shared secret scheme does not require any human involvement to set up keys. Our system also preserves the repudiability of email (see Section IV). The shared secret between S and R is not meaningful to a third party. So although R can satisfy itself with the authenticity of N from S, it cannot use N to prove to a third party that M_{new} was sent by S. In contrast, if SureMail required PKI/PGP, M_{new}'s author could be proved to anyone.

F. First-time Sender (FTS)

While the experiment in Section III showed email loss in the non-FTS case, it is desirable to address the FTS case as well. In our reply-based shared secret scheme, a legitimate FTS, who has not exchanged email with the recipient previously, cannot construct an authentic notification. Although it might seem that a legitimate FTS is indistinguishable from a spammer, in practice, there are social or business [4] links that may set apart the legitimate FTS from a spammer. [20] shows that email networks exhibit *small-world* properties and RE: [23] leverages it to create effective whitelists. So, although the FTS F may never have communicated with the intended recipient R, it is likely that F has communicated with an intermediary I who has in turn communicated with R. For example, I may be a colleague at R's institution. I is thus in a position to "introduce" F to R.

We want F to be able post a notification that R would treat as authentic. In leveraging F and R's relationship with I, we seek to preserve privacy by preventing: **(a)** I from learning that F intends to communicate with

[4]Enterprise networks typically have authentication mechanisms such as Kerberos that can validate legitimate employees.

R, **(b)** F from learning that I and R have previously communicated, and **(c)** R from learning that F and I have previously communicated. In the event that (c) cannot be satisfied, we consider a diluted goal, **(c')**, which is to prevent R from learning about any of F's correspondents other than those it shares in common with F.

There has been prior work on similar problems in the context of exchanging email whitelists. In LOAF [19], users exchange address book information with their correspondents using Bloom filters. This scheme satisfies property (a), but not (b), (c) or (c'). RE: [23] uses a novel friend-of-friend (FoF) approach. Using a homomorphic encryption-based private matching protocol [22], RE: ensures properties (a), (b), and (c'), but not (c). (However, RE: may permit a malicious R to violate (c') and learn about friends of F that it doesn't share in common, per Section 6 of [23]). Our introduction mechanism also satisfies (a), (b), and (c'), but not (c).

1) SureMail Introduction Mechanism: In our introduction mechanism, every node I generates a public-private key pair, (S_I, P_I). It shares the public ("shared") key, S_I, with its correspondents to establish a *common* secret known only to its correspondents (the public key is *not* shared with others — there is no PKI). In addition, I generates a public-private key pair, (S_{IF}, P_{IF}), for each new correspondent, say F, which it hands to F along with a signed token containing the F's email address and the newly generated public key, S_{IF}. So when it becomes a correspondent of I, F learns the common secret S_I, the key pair (S_{IF}, P_{IF}) generated for it, and the token $X_{SF} = SIG_{P_I}[F, S_{IF}]$ signed with I's private key, P_I.

F can use S_I and X_{SF} to authenticate notifications it posts for R, which is another correspondent of I. A notification from F for an email M sent to R is :

$$N_{FTS} = \{T, H(S_I), E_{H'(S_I)}(T, X_{SF}, R), \\ SIG_{P_{IF}}[H(smID_M)], SIG_{P_{IF}}[E_{H'(S_I)}(T, X_{SF}, R)]\}$$

$H(S_I)$ allows R to look up S_I from its store of common secrets obtained from its correspondents (and discard the notification if the lookup fails). It can then compute the key $H'(S_I)$ and decrypt $E_{H'(S_I)}(T, X_{SF}, R)$. Note that unlike in the non-FTS construction from Section V-D, F and R need to be identified explicitly since there is no pairwise shared context between them. Also, the signed token X_{SF} prevents F from assuming an arbitrary identity, which helps defend against certain attacks (see the security properties discussion in Section V-F.3).

After verifying the signed token X_{SF}, R uses F's public key to validate $SIG_{P_{IF}}[H(smID_M)]$, preventing an attacker from tampering with the message ID. The encrypted token, $E_{H'(S_I)}(T, X_{SF}, R)$, is also signed, which among other things ties the notification down to recipient R. This prevents R from turning around and reusing it in a fake notification purporting to be from F and destined to another correspondent of I, say Z.

If the introduction is deemed as valid per the above procedure, R honors the notification. Otherwise, R ignores it. Of course, as a side-effect of processing a valid notification, R also learns that both F and itself share I as a correspondent, which violates property (c) noted above but is consistent with (c').

2) Picking an Intermediary: F needs to pick an intermediary I whose shared secret (S_I) it should use for the introduction. We believe that it is appropriate to rely on human input, since the FTS scenario would occur relatively infrequently and the user at F is in the best position to decide which I would likely have communicated with R and whose identity it is allowed to leak to R. So when the SureMail client at F detects that it is an FTS with respect to R, it prompts the user for a recommendation of one or more intermediaries from among F's correspondents. The client aids the process by automatically listing correspondents who are in the same email domain as R and hence are likely to be suitable intermediaries. If the user picks more than one intermediary, a separate notification (constructed as N_{FTS} above) would be posted corresponding to each.

As an alternative, we can avoid user involvement by having F's client automatically post multiple notifications constructed with the shared secrets obtained from each of its correspondents. R could then determine if any match with its list of correspondents, and if so, deem the introduction as valid. Since the shared secrets are opaque tokens, R does not learn anything from the shared secrets that originated from correspondents of F that are not common to R. Thus property (c') is satisfied. However, note that this extreme procedure generates a notification traffic volume proportional to the number of correspondents, as in schemes such as RE:.

3) Security Properties: The notification construction prevents an attacker (other than a correspondent of I), who sees N_{FTS}, from learning the secret S_I, or identifying F or R. Furthermore, even a correspondent of I is prevented from constructing a fake notifications purporting to be from F. Also, note that repudiability is also preserved since F only signs the message ID using P_{IF}, not the message content itself.

In terms of privacy, property (a) is satisfied since I is oblivious to the communication between F and R. Property (c') is also satisfied as noted above. While property (b) is also satisfied by the protocol as described, there is the possibility of a subtle social engineering attack: F could post a notification for R (with I as the intermediary) but *not* send the corresponding email. If (an anxious) R inquires with F about the "missing" email, F can conclude that I and R must be correspondents (for otherwise R would have just ignored the notification).

However, we believe that this social engineering attack

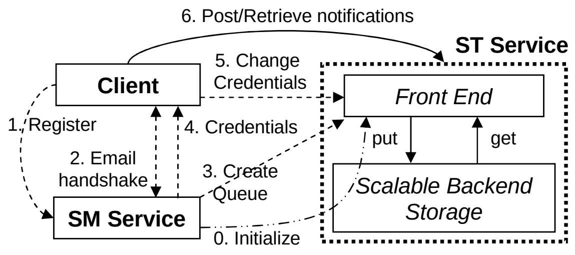

Fig. 4. Interaction between a client, SM, and ST. The various operations are numbered in the order in which they are invoked. The dashed-dotted line operation (#0) is invoked at system initialization time. The dashed line operations (#1-#5) are invoked at the time of client registration. The solid line operation (#6) is invoked during normal operation, i.e., posting and retrieval of notifications.

would be hard to employ in practice. Note that for the attack to succeed, F would have to be in a position to receive R's inquiry about the missing email. However, F cannot use a fake email identity, say F', since it wouldn't have the corresponding $X_{IF'}$ signed by I. So F would likely have to identify itself and run the risk of being caught if it repeats the trick over time.

VI. Delivering SureMail Notifications

We now present two complementary approaches to delivering notifications: in-band and out-of-band.

A. In-band Notifications

In-band notifications are embedded within emails themselves — the notification for an email is embedded later emails between the same sender-recipient pair. Suppose S sends R three emails: M_1, M_2, M_3. M_2 will contain the notification for M_1. M_3 will contain the notifications for both M_1 and M_2. If R did not receive M_1, it will find out when it receives either M_2 or M_3. This is akin to how TCP detects a missing packet via a sequence "hole". Since these notifications are confined to the email system, privacy concerns are avoided. We use a simpler notification construction than in Section V-D.

We include an `X-SureMailRecentIDs` header containing the smIDs of a small number (say 3) of emails sent recently, allowing the R to determine if it is missing any of those emails. Each recent smID is repeated in more than one subsequent email, which provides some tolerance to the loss of consecutive emails.

S also includes an `X-SureMailSharedID` header containing a reply-based shared secret (the smID of an earlier email it had sent and that R had replied to). Upon receiving a new email, R uses the `X-SureMailSharedID` field to check its validity and then `X-SureMailRecentIDs` to check if it is missing any email. This prevents a spammer from subverting loss detection. For an FTS, `X-SureMailSharedID` instead contains $H(S_I)$ (Section V-F.1).

Despite its ease of deployment and simplicity, in-band notifications have a disadvantage. Loss detection can only be as fast as the frequency of email exchange between S and R. Furthermore, consecutive email loss can further delay loss notification.

B. Out-of-band Notifications

In contrast, *out-of-band* (OOB) notifications are delivered via a channel that is separate from the SMTP-based email system. This decouples notification delivery from the vagaries of the email system. The detection of email loss does not have to wait for the receipt of a later email.

The OOB channel can be viewed as a storage system to which notifications are posted and from which they are retrieved using a (synchronous) put/get interface. Such a channel could be realized using, for example:

- A distributed hash table (DHT) overlaid on clients.
- A storage service such as OpenDHT [28] or commercial services such as Amazon's SQS [1].
- Dedicated notification servers for each domain to receive notifications intended for the domain's users.

We choose a design that builds on cheap, large-scale Internet-accessible storage services since these are increasingly becoming available (e.g., Amazon's SQS [1]). The simplicity of notifications and the synchronous posting of notifications (both in contrast to email) mean that the sender has a very high degree of assurance (equal to the reliability of the storage system itself) that its notification will be delivered to the recipient.

We decompose the OOB notification system into two components: a SureMail-specific service (*SM*) for the control path (e.g., handling the registration of new SureMail users) and a relatively generic storage service (*ST*) for the data path (e.g., handling the posting and retrieval of notifications). Figure 4 shows how they interact. We assume that SM and ST are operated by separate, non-colluding entities. ST can directly leverage a generic storage service. However, for full functionality, we identify a few additional capabilities and APIs we need beyond the standard put/get interface of existing storage systems. We believe they are useful for other applications as well.

The separation of SM and ST has several benefits. All of the relatively heavyweight data path workload is confined to ST, which leverages existing scalable storage services built for Internet-scale applications. The SureMail-specific SM handles the relatively lightweight control path workload, which makes it easier to build and operate. Furthermore, SM holds minimal state and is thus easy to scale with increasing load. Administrative separation between SM and ST means that neither component is in a position to learn much about the email behavior of users, even if individually they are not entirely trustworthy (Section V-A). SM knows user identities but does not see their notification workload. The converse is true for ST.

1) SureMail Service (SM): SM handles the registration of new SureMail users. This is needed to **(a)** prevent users from accessing notifications posted for others (e.g., to monitor their volume), and **(b)** detect and block DoS attacks. Except where noted, operations on a user's behalf are performed *automatically* by their SureMail client.

As the first step, a new SureMail user's client contacts SM, giving the email address of the registering user (U). SM performs an email-based handshake (i.e., sends back an email to this address) to verify that the client "owns" the email address it is trying to register. (This is a commonly-used procedure for authenticating users, e.g., when they sign up for online services [24].) Note that this relatively heavy-weight email-based handshake is invoked just once, at the time of initial registration. If the challenge-email itself is lost, the client can retry.

To block registration attempts by automated bots, SM also returns a human-interactive proof (HIP) [6] to the new user and verifies the answer before proceeding. In this process, SM and the client establish a key, k_U, that is unique to the email address being registered and serves as the user's credentials. SM then contacts ST to create a queue to hold notifications sent to user U.

For reasons given in Section VI-B.2, k_U is constructed to be self-certifying — it is of the form $(x, MAC_z(x))$, where the key z used in the MAC is known only to ST, but not to SM or the clients. ST supplies new, valid credentials to SM, as needed, to pass on to new clients.

As explained further below, the notification queue for user U is set up to allow other users to post notifications for U but requires a key for reading these notifications. SM sets this key to k_U at the time it creates the queue for U. Possession of this key enables user U's client to directly talk to ST to change the key associated with its queue to $k_U^{'}$. Doing so prevents SM from snooping on its notifications (described in Section VI-D).

Clients renew their registrations periodically, say once a month. Renewal requires solving a new HIP. The email-based handshake and queue creation are not repeated. Renewal serves two purposes. First, it allows the system to "forget" users who have stopped using it, thereby reclaiming the corresponding resources (their storage queue). Second, it is harder for an attacker to steadily build up a large set of bogus registrations; the attacker would have to do work (solve a HIP) each time a registration expires. We believe that solving a HIP, say once a month, would not be a burden for legitimate users.

2) Storage Service (ST): ST manages the notification queues for users. It is oblivious of notifications themselves and treats them as opaque objects. It allows users to post and retrieve notifications using an interface similar to standard put/get interfaces, but with a few extensions.

First, it supports an administrative interface, which (in SureMail) only allows SM to create queues for new users. It also allows the admin to specify workload limits to block DoS attacks, as we explain below. Second, the queues provided by ST support authenticated read but allow unauthenticated writes. This is in contrast to the usual practice of authenticating writes alone or both reads and writes. ST also allows the read-key associated with a queue to be changed by presenting the current key.

We now turn to the construction and operation of ST. This service has a front end (FE) that communicates with clients and a back end (BE) that provides storage service. When it receives a client request, the FE typically invokes one or more operations on the BE. The FE only holds soft state, so it easy to scale out to keep up with load.

At user registration, SM calls on ST to create a notification queue for the user. The read-key for this queue is initially set to k_U. The user's client then contacts ST to change the key to $k_U^{'}$. This completes registration and the user is then in a position to receive notifications from others and post notifications for others. When a client posts or retrieves notifications for/to U, the FE invokes a $put()$ or $get()$ operation on the corresponding queue in the BE. The client identifies the queue using $H(U)$, so ST does not directly learn U's identity.

Despite the small size and lightweight processing of notifications, a DoS attack aimed at overwhelming the computational or storage resources of ST remains a possibility. To defend against this, SM, at system initialization time, specifies a limit on the rate of notification postings by a sender (e.g., the maximum number that can be posted in a day) and the maximum storage allowed for notifications posted by the sender (e.g., the sum of the time-to-live (TTL) values of posted notifications). It also specifies a limit on the frequency of notification retrievals by a recipient, which is enforced using a procedure analogous to the case of notification posting discussed here. ST does not enforce these limits under normal (i.e., non-attack) conditions. However, if it suspects an attack (e.g., its load is high or its storage resources are being depleted rapidly), ST enforces the per-sender limit by dropping any excess notifications. We discuss the implications of such dropping in Section VI-B.3.

To enforce per-sender limits, ST needs to identify senders in a way that is resilient to cheating. We require any client, U, that is posting or retrieving notifications to include, with its request, the original key (k_U) that it had obtained from SM at the time of registration. The HIP mechanism presents a barrier to bogus registrations on a large scale. When a client request arrives, the FE of ST first performs a quick stateless check to verify that the key is well-formed (recall from Section VI-B.1 that k_U is self-certifying), discarding the request if it isn't. Otherwise, the FE queries and updates workload/resource usage information for U in the BE and checks if any limits have been exceeded. If any have been, U's identity is pushed out to all FE nodes (as soft state, say for the rest of the day), to block further requests from U without the need for the more expensive BE lookups.

3) DoS Defense and System Scalability: First, we consider the load placed on ST as notifications are posted and retrieved, assuming the quota checks are being performed. The posting or retrieval of a notification involves:

a) Verifying that the presented key is well-formed and, if appropriate, filtering notification postings from senders who have exceeded workload limits.

b) Retrieving the workload information for the requester, checking whether any limits have been exceeded, and storing back the updated workload information. If any limits have been exceeded, the sender is blocked and its identity is pushed out to all FE servers.

c) Storing the new notification in the case of a posting, or fetching notifications in the case of a retrieval.

Under normal operation, only step #c is performed, which involves a single $put()$ for a posting or a single $get()$ for a retrieval. When the system is under heavy load or attack, steps #a and #b are also invoked. However, we expect much of the attack traffic to be filtered in #a, either because the keys are not well-formed or because the sender has already been blocked. Note that #a is performed at the FE and does not require accessing the storage layer itself. While #b does require an additional $get()$ and $put()$, we could reduce the load significantly by performing it infrequently, say once every 10 posting/retrievals picked at random, and scaling the workload limits accordingly. Thus even when the system is under attack, the load on ST is not much more than one $put()$ or $get()$ per posting/retrieval.

A second issue is what the notification workload limits should be set to. If we set a (generous) limit of 1000 notifications per sender per day and an average TTL of 10 days, then at any point in time, ST would have up to 10,000 notifications posted by a sender. Each notification is 64 bytes in size (Section VII). This is a maximum storage of 625 KB per sender. So an ST with 1 TB of storage would support 1.72 million users. Using Amazon's S3 pricing [1] (storage rate is US $0.15 per GB per month and transfer rate of $0.20 per GB), it would cost a modest $1383 per month ($154 for storage, $1229 for transfer), or about US $0.0008 per user per month. In practice, most users may generate far fewer than 1000 notifications per day, driving costs down even further.

These calculations also suggest that it would be challenging for an attacker to consume a significant fraction of the storage resources. For instance, to consume 50% of the storage resources, the attacker would have to masquerade as over 0.8 million different senders, each of which would have had to register with the SM service and get past the HIPs.

Since the workload limits are only enforced when the system is under attack, the system is flexible in terms of the volume and the TTL of notifications under normal operation. For instance, the relatively small number of legitimate high-volume senders (such as credit card companies emailing monthly bills) can each post well over 1000 notifications per day without impacting the overall system load. Likewise, a client could choose to set the TTL for a notification to much longer than 10 days.

However, when the system is under attack and starts enforcing the limits, high-volume senders will have their attempts to post notifications temporarily blocked. To get around this, such senders would have to set up multiple registrations with SM. In any case, notifications from typical (i.e., low-volume) senders, who presumably constitute the overwhelming majority, would be unaffected even when workload limits are being enforced. Bogus notifications that are part of the attack are still not presented to the user due to our shared secret scheme.

C. Combining In-band and Out-of-Band Notifications

Since both in-band and out-of-band notifications have their advantages, our design incorporates both. In-band notifications are cheap and do not expose any information beyond the email system. Thus, we use these as the first line of defense. If an email is replied to (i.e., implicitly ACKed), there would be no reason to post an OOB notification for it. Likewise, if a NACK is received for an email (e.g., a bounceback is received or the recipient has already complained about the email being missing, say based on an in-band notification), again there isn't the need to post an OOB notification for the original email. If neither has occurred after some duration, we need to post an OOB notification, which is likely to be more reliable than an in-band notification. Analysis of the email behavior of 15 users at Microsoft suggests holding off for about 10 hours (to accommodate 75% of email replies (i.e., ACKs) for this set of users).

D. Privacy Implications of SureMail Notifications

In-band notifications do not impact privacy by design, so we focus here on OOB notifications. Our design prevents users from accessing the notification queues of other users because they do not have access to the read key for the corresponding queue (Section VI-B.2).

SM is similarly also blocked because it does not possess the read key after the user has updated it (i.e., changed it from k_U to $k_U^{'}$ per Section VI-B.2). Recall also from Section VI-B that SM and ST are assumed to be non-colluding. SM could try to cheat by changing k_U to a $k_U^{'}$ of its own choosing at the time of initial registration (or by doing so for arbitrary users who may have not even tried to register). However, doing so would prevent the legitimate U from successfully completing its registration and hence it would opt out of SureMail, leaving SM with access to an unused notification queue.

ST has access to the notifications. However, since it does not possess the reply-based shared secret between a pair of users, it is unable to interpret them. Furthermore, notifications for U are posted to $H(U)$, so ST does not have direct access to U's identity. While, ST could try to reverse the one-way hash for users in its dictionary, this would only allow it to learn the volume of notifications posted to those users, not who posted them or what the corresponding email content is. We believe that volume information alone is not very sensitive because of notification spam or the user themselves posting fake notifications to "pad" their queue, without ST being any the wiser. Also, ST can be prevented from gleaning information from client IP addresses using anonymous communication (e.g., Crowds), if a client so desires.

VII. Implementation and Evaluation

We discuss ST & SM services to support OOB notifications and a client that also supports in-band notifications.

A. Storage Service (ST)

ST comprises a front-end that implements the extensions from Section VI-B.2, and Amazon's S3 & SQS web services as the backend. SQS [1] allows multiple values to be posted for a key and so is used for the notification queues. The simpler hash table like interface of S3 [2] is used for other persistent information (e.g., usage stats.).

1) Front-end Shim (FE): Our FE is a multi-threaded C++ program that processes requests from SM (to create queues for new users) as well as clients (to update their credentials and to post/retrieve notifications). Upon receiving a request, FE validates the credentials presented, updates usage statistics in S3 and checks against workload limits, and posts/retrieves information to/from S3 or SQS, as needed. Communication with the storage backend is secured with an access key known only to FE.

FE processing involves low overhead operations to receive/send client messages, unpack/pack the messages, and perform keyed SHA1 hashes (to check that credentials are well-formed). Ignoring the high network latency to the BE (since the FE and BE were not co-located in our setup), our prototype performed over 9000 operations per sec. of notification posts/retrievals, while saturating one core on a dual-core 3.0 GHz Pentium D with 2 GB of RAM. The FE does not hold any hard state and can easily be replicated to support higher request rates, so the scalability and reliability of the storage backend is not hampered by the additional functionality of the FE.

2) Storage Backend (BE): Based on a conversation with the designers of Amazon's S3 and SQS, we learned that both systems replicate data across storage nodes within the same data center as well as across different data centers. Upon failure of any node or data center, the data is re-replicated. The system is designed to meet 99.99% availability. However, the designers did indicate the reliability target other than to "never lose data".

Queues	44
Notifications posted	207752
Notifications retrieved	207747
Reliability	99.9976%

TABLE VI. Notification Storage Experiment Results

During email loss experiment 3 in Table I, we evaluated the suitability of SQS as the notification storage backend. For each email sent, we pushed a corresponding notification onto the SQS queue for the recipient. We did not route requests via the FE prototype during this experiment. We periodically retrieved the notifications from each queue, clearing out the queue in the process. Our findings are summarized in Table VI. The 4-nines reliability achieved is 500x better than that of email itself in the same experiment (Table I). So we conclude that the SQS storage backend available today is quite reliable for our purposes. (The 5 notifications lost had been posted to 4 different queues within a 30-minute window.)

B. SureMail Service (SM)

SM is a multi-threaded C++ program that processes SureMail client registrations. On receiving a registration request, SM returns a HIP to the client over the same TCP connection. It also emails a randomly generated password string to the email address the client is attempting to register. At a later time, when the client returns this password and the HIP answer in a registration confirmation message over a new TCP connection, SM creates a queue for the user by contacting ST and returns the well-formed user credentials (obtained from ST) to the client.

C. SureMail Client

We have implemented a standalone C++ client that interacts with both SM and ST as noted above. We have used this for testing and for benchmarking. Figure 5 shows the binary format of a post notification message. The notification itself (which is what is stored by ST) is 64 bytes long but is embedded in a 124-byte message.

We have also implemented a C#-based add-in for Microsoft Outlook 2003 to enable real use of SureMail. Our add-in uses the Outlook Object Model (OOM) [7] interface to intercept various events (e.g., email send, receipt, reply) and the Messaging API (MAPI) [25] to add x-headers. For compactness, we coalesce the various x-headers from Section VI-A into a single `X-SureMail`header that includes the message ID, recent IDs, in-reply-to ID, and shared secret. As of publication, the add-in implements the reply-based shared secret scheme and supports in-band notifications. Support for OOB notifications in this add-in is in progress.

VIII. Discussion

A. When Should the Recipient be Alerted to Loss?

A SureMail notification could arrive a few moments before the respective email. During that time, the email

Byte 0	Byte 1	Byte 2	Byte 3
Version	PktType	PktSize	TTL
CredX (4 rows = 16 bytes)			
CredY (5 rows = 20 bytes)			
H(RecptEmailAddr) (5rows)			
Timestamp (T)			
$H(smID_{M_{new}})$ (5 rows)			
$H(smSS_1)$ (5 rows)			
$MAC_{smSS_1}(T, H(smID_{M_{new}}))$ (5 rows)			

Fig. 5. Post Notification Message Format: The top half contains the credentials and the key under which the notification (bottom half) is to be stored (see Sections V-D and VII-A).

is merely delayed and not lost, and thus the receiver should not be falsely alerted to email loss. For each of the 138,944 emails sent in Experiment 1 (Section III), we calculate the end-to-end delay from when it was sent by our program to the timestamp inserted by the receiving account's email server. Almost a third of the non-lost emails have slightly negative delays, due to the lack of clock synchronization between our sender and recipient MTAs. Of the rest, the median delay is 26 seconds, mean is 276 seconds, standard deviation is 55 minutes, and maximum is 36.6 hours. Thus we believe 2 hours is an appropriate duration to wait before alerting the user.

B. Should Emails also be Delivered via SureMail?

Emails themselves should *not* be delivered via the SureMail OOB channel because doing so would fundamentally alter the nature of the channel. The rich and extensible nature of email would necessitate filtering to block malware, which is not needed with our tiny, fixed-format notifications. Emails also tend to be much larger and so would impose a far greater overhead on the OOB channel. The median email size (including attachments) across 15 user mailboxes we analyzed at Microsoft was 4 KB and the 95^{th} percentile was 44 KB. In contrast, our notification payload is 64 bytes (Section VII).

C. Supporting Email Lists, One-Way Communication

There are cases such as mailing lists and one-way communication (e.g., email bank statements) where the normal reply-based handshake may not work. Instead, we could have a shared secret be set up, say via the usual email address validation handshake at sign-up. For a mailing list, this shared secret could be shared across all members and a common list-wide queue used for posting and retrieving notifications. In the bank case, the shared secret would be unique for each user and notifications would be posted to the individual user queues.

IX. Conclusion

Our measurement study shows that on average, 0.71% to 1.02% of email is lost silently. We have designed and prototyped SureMail. It complements the current email infrastructure and increases reliability by notifying recipients about email loss. This notification provides significant user value because the informed user can contact the identified sender for the missing information, say over phone or IM. SureMail supports in-band and out-of-band notifications with no changes to the existing email infrastructure and without PKI/PGP. It places minimal cognitive load on users. It includes mechanisms to defend against notification spam and breaches to user privacy. In our evaluation, SureMail ensured that silent email loss was detected with 99.9976% reliability.

Acknowledgements

We thank H. Balakrishnan (MIT), P. Barford (Wisconsin), C. Dovrolis (GaTech), R. Govindan (USC), S. Keshav (Waterloo), L. Qiu (UT Austin), J. Rexford (Princeton), H. Schulzrinne (Columbia), D. Veitch (Melbourne), and L. Zhang (UCLA) for helping us obtain email accounts for our experiments.

References

[1] Amazon Simple Queue Service. *http://aws.amazon.com/sqs*.
[2] Amazon Simple Storage Service. *http://aws.amazon.com/s3*.
[3] AOL Missing Email Self Help. *http://postmaster.info.aol.com/selfhelp/mbrmissing.html*.
[4] AOL Whitelist Information. *http://postmaster.info.aol.com/whitelist*.
[5] ePOST Serverless Email System. *http://www.epostmail.org/*.
[6] Human Interactive Proofs. *http://www.aladdin.cs.cmu.edu/hips/*.
[7] Outlook Object Model. *http://msdn2.microsoft.com/en-us/library/ms268893.aspx*.
[8] Pivotal Veracity. *http://www.pivotalveracity.com/*.
[9] Sender policy framework (SPF). *http://www.openspf.org/*.
[10] Silent Email Loss by EarthLink. *http://www.pbsf/cringely/pulpit/2006/pulpit_20061201_001274.html*.
[11] Spamassassin. *http://spamassassin.apache.org/*.
[12] U.C. Berkeley Enron Email Analysis Project. *http://bailando.sims.berkeley.edu/enron_email.html*.
[13] Vipul's razor. *http://razor.sourceforge.net/*.
[14] B. Adida, S. Hohenberger, and R. Rivest. Fighting phishing attacks: A lightweight trust architecture for detecting spoofed emails. In *DIMACS Wkshp on Theft in E-Commerce, April 2005*.
[15] M. Afergan and R. Beverly. The State of the Email Address. *ACM CCR*, Jan 2005.
[16] S. Agarwal, D. A. Joseph, and V. N. Padmanabhan. Addressing email loss with SureMail: Measurement, design and evaluation. In *Microsoft Research Tech. Report MSR-TR-2006-67*, May 2006.
[17] S. Agarwal, V. N. Padmanabhan, and D. A. Joseph. SureMail: Notification overlay for email reliability. In *ACM HotNets-IV Workshop*, Nov 2005.
[18] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *ACM WPES*, 2004.
[19] M. Ceglowski and J. Schachter. Loaf. *http://loaf.cantbedone.org/*.
[20] H. Ebel, L.-I. Mielsch, and S. Bornholdt. Scale-free topology of e-mail networks. *Physical Review E66*, Feb 2002.
[21] R. Fajman. An Extensible Message Format for Message Disposition Notifications. *RFC 2298, IETF*, Mar 1998.
[22] M. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. *EUROCRYPT*, 2004.
[23] M. J. Freedman, S. Garriss, M. Kaminsky, B. Karp, D. Mazieres, and H. Yu. Re: Reliable email. *USENIX/ACM NSDI, May 2006*.
[24] S. L. Garfinkel. Email-based identification and authentication: An alternative to PKI? *j-IEEE-SEC-PRIV*, Nov/Dec 2003.
[25] I. D. la Cruz and L. Thaler. *Inside MAPI*. Microsoft Press, 1996.
[26] A. Lang. Email Dependability. Bachelor of Engineering Thesis, The University of New South Wales, Australia, Nov 2004. *http://uluru.ee.unsw.edu.au/~tim/dependable_email/thesis.pdf*.
[27] A. Mislove, A. Post, C. Reis, P. Willmann, P. Druschel, D. Wallach, X. Bonnaire, P. Sens, and J. Busca. POST: A Secure, Resilient, Cooperative Messaging System. *HotOS*, May 2003.
[28] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses. *SIGCOMM*, Aug 2005.