Revocation of Unread E-mail in an Untrusted Network

Aviel D. Rubin[1] rubin@research.att.com
Dan Boneh[2] dabo@bellcore.com
Kevin Fu[3] fubob@mit.edu

[1] AT&T Research,
[2] Bellcore, 445 South St., Morristown, NJ, 07960
[3] Computer Science Dept, M.I.T, Cambridge MA 02139

Abstract. We present protocols for enhancing e-mail systems to allow for secure revocation of messages. This paper identifies the security requirements for e-mail revocation and then shows how our protocols adhere to these requirements. Three different levels of security and threat models are described. We discuss our implementation of the level 1 protocol, which assumes no security infrastructure. The protocols were designed so that existing mailers can easily be enhanced with these new features.

1 Introduction

In the early days of the Internet, the use of electronic mail (e-mail) was limited to a small subset of highly technical people. It was a luxury that was not enjoyed by the rest of the population, and most people had never even heard of it. In spite of the small number of users of e-mail, the protocols (namely, smtp [11] and X.400 [3]) were highly reliable. Users could assume that if a message was sent, then either it would be received, or they would be notified that the transmission failed. In recent years, the use of e-mail spread to the general population. In spite of the tremendous increase in the number of users, the robustness and reliability of e-mail is still taken for granted. This is a result of standards for how e-mail is sent, received and processed. These standards make e-mail a convenient and reliable means of communication, but they also make it difficult to change e-mail or to add features. In this paper, we propose a feature that can be added to e-mail without requiring changes to the underlying protocols. Any user wishing to enable this feature must enhance his mail software, with minor modifications.

It would be nice if e-mail systems enjoyed the same functionality as the US postal system, but it turns out that they don't quite measure up. We did some digging, and discovered a little-known post office procedure [16]. It turns out that if you mail a letter or a package to someone, and it then becomes important to you that they not receive it, then there is a way to stop it. It can even be done by phone. The local postmaster contacts the postmaster at the destination and alerts him about the recall. The letter is intercepted and returned to you. The procedure will only work if you provided a return address and a proper

description of the envelope or package. It is a federal offense to lie during the mail recall procedure.

If the US post office implements revocation, then it seems that such a service would be desirable in e-mail as well. However, the electronic equivalent of the mail recall form is non-obvious. We use several examples to illustrate why revocation is useful.

Take the following typical scenario (which actually occurred). Alice asks her secretary, Bob, to register her for a conference, where participation is limited. One week before the conference, Alice calls to confirm her registration, because she never received a confirmation. The conference organizer tells Alice that she is not registered. Alice tries to call Bob, but he is away on his lunch break. So, Alice sends a sharp e-mail message to Bob asking why he did not make the reservation. 20 minutes later, the phone rings. It is the conference organizer saying that there had been a mistake and that Alice's registration had been received on time. Alice must send an apology to Bob. Later that day, Bob reads his e-mail. First, he sees the sharp note from Alice, and he sends a long, detailed response about how he took care of everything, and that Alice should not be so quick to blame him when things go wrong. After sending that message, Bob reads the second message. He then sends an apology to Alice. Relations at the office are at an all-time low. The whole problem could have been avoided if Alice had a way of cancelling the first message she sent before Bob returned from lunch. In most current e-mail systems, there is no way to do this without contacting a system administrator or breaking into someone's account.

A second scenario also occurs quite often, and most readers should be familiar with this type of event. Charlie is in charge of a meeting for his entire department of 25 people. He sends out a broadcast message using an e-mail alias announcing that the meeting will be at noon on Friday. An hour later, he gets an e-mail response from a member of the department reminding him that Friday is the company picnic, and that the department is playing another department in the company softball finals. The CEO of the company is the umpire, and so Charlie must reschedule. He immediately sends out another broadcast message announcing that the meeting has moved to Monday. Over the next two days, Charlie receives messages from people about the softball conflict. These messages are usually followed by apologies saying that the person just saw the second broadcast and never mind. If Charlie had a way to revoke the first broadcast message for all users who hadn't seen it yet, his mailbox would have been much emptier the rest of the week.

These two scenarios involve people in the same organization. The problem becomes more interesting for users across the Internet. Again, we use a scenario to illustrate. Donna is negotiating a deal to purchase the ACME toy company. Her main contact there is Ed. At 4:55 PM, Donna sends Ed an e-mail message with a bid for $5,000,000. At 5:30, there is no response, so she goes home. That night, she watches the news and hears that Ed is being sued for unpaid child support and that he is extremely desperate for cash. She logs into work from home and checks to see if Ed has received her e-mail yet. She receives

confirmation that he hasn't. She revokes the e-mail that she sent earlier and sends a new message with a bid for $3,500,000. The next morning, Ed gets to work at 9:00 and sees the bid from Donna. He immediately accepts it.

The latest scenario is not possible in today's Internet for two reasons. There is no way to find out if a message has been received, and there is no mechanism for revoking a message. In this paper we present a way to receive a correct notification on the status of a message (received or not received), and a mechanism for revoking a message without leaving any evidence that it was ever sent. The sender will also be able to revoke a message and receive correct notification that either the message was revoked without leaving any trace, or that it was received before the revocation completed.

While there are mail systems, such as the Michigan Terminal System (MTS) [8] and Novell GroupWise™version 4.1 that enable this functionality within a local site, there is no general-purpose system that allows a user to revoke a message to an Internet user with a different mail system. Furthermore, these systems do not protect users from eavesdroppers on their networks. For example, in MTS, if a user is able to listen in on another user's communication, say, by listening to ethernet traffic, then he can cause a message sent by that user to be revoked. It is obviously desirable that only the user who sends a message should be able to revoke it.

Finally we note that the ideas and protocols developed in this paper also apply to the domain of news groups. It is desirable to enable a user to revoke a message he posted. Currently there is no mechanism which *securely* enables users to revoke their postings. As in the E-mail case it is desirable that only the user who posted the message should be allowed to revoke it. Our techniques can be directly applied to securely revoking news postings.

2 E-mail

To familiarize the reader with our terminology we give a high level overview of electronic mail protocols. A user known as the *sender* may at any time send mail to another user known as the *receiver*. When a mail message is sent it is received on the receiver side by a *mail user agent* or MUA for short. At any point in time the receiver may ask his MUA to deliver all of the new incoming messages. Usually the MUA is implemented as a daemon running on some server. When new mail is sent to one of the server's clients the MUA stores the incoming message in a spool file. The receiver can read his incoming messages by instructing the MUA to send him the contents of the spool file.

The MUA is composed of two components. The first handles the reception and storage of incoming mail messages. The second delivers messages to the receiver on demand. There are two categories of mail systems. In the first type of system, users can receive their e-mail only by interacting with a limited interface provided by the MUA. Internet service providers such as America Online,

[3] *GroupWise* is a trademark of Novell.

Prodigy, Compuserve, etc. fall into this category. In the second type of system, users can bypass the MUA and have direct access to their e-mail. This is often the case in Unix, where users can use the `mail` program, or they can access their mail directly by reading a spool file. We will call the first type of system the Limited Interface Model (LIM) and the second type the Direct Access Model (DAM). Figure 1 demonstrates a typical LIM e-mail system.

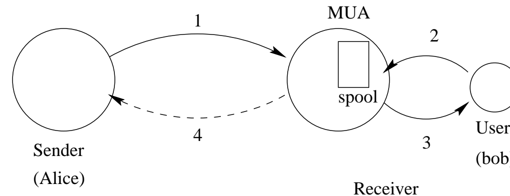

Fig. 1. A typical LIM e-mail system In step 1, Alice sends a message, M, to Bob's Mail User Agent, MUA, where it is stored in a spool file. When Bob is ready to read his mail, he requests his messages from MUA (step 2), and in step 3, MUA sends Bob his message. Most systems do not implement step 4, where MUA notifies Alice that Bob read the message.

For the LIM model, we view the MUA as one unit. It receives and stores messages and then delivers them to the user on demand. In the DAM model, these two processes are quite different. A process with higher privilege than most users, called the *mail daemon*, handles the receipt of incoming messages. These are stored in a spool file according to the permissions of each user so that users can access their own mail, but not the mail of others. At this point, users can access their mail through a mail reading program or directly through the spool file. In Unix, the mail daemon runs with *root* privileges so that it can store messages such that they are owned by different users. Thus, the mail daemon is a "trusted" program. We will make use of this fact in our protocols.

Our enhancements to the mail system require some changes to the MUA. Since the MUA is totally out of the user's control in the LIM model, it is easy to prevent the user from bypassing our enhancements. Furthermore, in this model our enhancements can be made transparent to the user. In the DAM model, only the daemon can be secured. Access to the mail messages must be controlled through encryption once the messages are stored.

3 Goals and Requirements

In our system, we would like to provide some new capabilities to e-mail systems. Besides the ability to *send* and *read* messages, we would like the sender to be able to *check* the status of messages (read or not read), to *revoke* unread messages, and to receive notification (a receipt) when a particular message is read.

The following are requirements for the new e-mail system. In the following list, we assume that Alice has sent message M to Bob, Charlie is a malicious eavesdropper who can forge message from Alice or Bob, and Donna is a sophisticated attacker who can modify messages in transit and has complete control of all traffic. We also assume that there is a delay, α, for a query to travel from Alice to Bob, a delay δ, for a response to travel from Bob to Alice, and a negligible processing time for queries. We say that Bob *read* M if Bob requests M from his MUA. Obviously, there is no way to determine if Bob actually scanned the message with his eyes.

It should be noted that a way for Bob to avoid revocation is for him (or a program written by him) to requests his mail so frequently from his MUA that there is never a chance for Alice to revoke a message. We say that Bob can *avoid* revocation, but that he cannot *defeat* revocation. That is, Bob cannot prevent Alice from revoking a message that he has not read. The second protocol in this paper provides a mechanism for Alice to detect attempts by Bob to *avoid* revocation. She can then take proper action, such as ceasing further communication with Bob.

The following are desirable requirements for an e-mail revocation system:

1. Alice can check to see if Bob read M. If she receives an answer of 'no' at time t, then Alice knows that at time $t - \delta$, Bob had not read M.
2. Alice can revoke M at time t. If at time $t + \alpha$, Bob has not read M, then Bob will not ever read M.
3. If Bob reads M at time t, then if Alice tries to revoke M, after time $t - \alpha$, she will be notified that the revocation failed.
4. If Bob has not read M at time $t + \alpha$, then if Alice checked at time t, the notification says that M has not been read. This is the converse of Goal 1.
5. If Alice successfully revokes M, then Bob does not find out that Alice sent him a message.
6. If Alice successfully revokes M, then Bob does not find out that any message was sent from anyone.
7. Charlie cannot cause M to be revoked.
8. Donna cannot cause M to be revoked.
9. Charlie or Donna can never check to see if Bob has read M, without access to the link between Bob and his MUA.
10. Charlie cannot cause Alice to receive the wrong notification.
11. Donna cannot cause Alice to receive the wrong notification.

Some of these requirements are easier to meet than others. We define three levels of e-mail revocation. In the rest of the paper, we provide solutions to the problem at each of these levels. As expected, solutions at the levels with the higher number or requirements come at a greater cost. In each section, we describe which requirements are met.

4 Infrastructure

The infrastructure assumptions are the most important in designing protocols for e-mail revocation. It is obvious that these protocols would be very easy to design if we had a full-blown public key infrastructure, where every party had valid copies of every other party's public keys. In fact, authentication protocols in the literature could be used to meet all of our goals.

However, it is unrealistic to assume that a public key infrastructure will exist any time soon. Our aim was to explore what could be achieved with little or no infrastructure. Our level 1 protocol assumes that there is absolutely no infrastructure. In level 2, we assume a weak form of infrastructure to achieve better security. Rather than assume that there is a universal certifying authority that everybody trusts to verify user's identities and to issue certificates, we assume that there is a party that is trusted to keep secrets and behave appropriately. This is very different from trusting a CA. In particular, one of the most criticized aspects of public key infrastructure is that there is no way to be sure that CA's are competent in verifying users' identities. Thus, our trusted third party assumption is weaker than the public key infrastructure assumption. We call our third party an *honest* third party.

Thus, we present level 1 and level 2 protocols with increased security at the cost of greater assumptions. One advantage of this is that the level 1 protocol could be implemented right away. If, at some future date, there is more infrastructure available, then people could switch to level 2, and maybe even level 3. For the latter, we assume full public key infrastructure where all parties have each others' public keys. In this case, secure revocation of e-mail is a simpler problem.

5 Level 1

The level 1 system offers e-mail revocation under weak assumptions. In particular, although requirement 7 is satisfied even if Alice or Bob cheat, requirements 1-3, 5 and 6 are only satisfied if all parties play by the rules. This type of system is useful for users who trust each other and want to have e-mail revocation as a convenient, useful service. It could also be effective for unsophisticated users who would be unable to mount the attacks necessary to defeat the system. Today, most e-mail users fall into this category. Level 1 is especially suited for people whose mail service is LIM (see Section 2) because in this model, users cannot directly access their mail file, so they cannot defeat revocation by, say, reading their spool file directly.

We note that in the description of the protocol we assume in order delivery of mail messages. For instance, if a user sends an E-mail message and later revokes it by sending a revocation message then we assume the two messages arrive at the appropriate order. In practice, this is not necessarily the case, and future work is needed to account for messages that are received in the wrong order.

5.1 Security model

In level 1, we assume that the principals involved behave according to the prescribed protocol. If a party cheats, it can potentially defeat some of the security requirements. Figure 1 depicts a typical e-mail system. In level 1, we assume that Bob does not

- eavesdrop on message 1
- access the spool file without MUA recording it
- change the behavior of MUA

We present a protocol where requirements 1-7 and 10 hold, under these assumptions. Requirement 9 holds until Bob actually reads M. At that point, Charlie and Donna will detect the notification message that is returned to Alice.

One of the features of our system is that Alice can send a query to MUA to see if Bob read M. Alice must be able to do this, while Charlie and Donna must not be able to generate a valid request to MUA.

Donna can trivially cause M to be revoked (requirement 8) by modifying the original message, M, to contain nothing. Similarly, Donna can prevent Alice from successfully revoking a message by blocking the revocation request, and she can also tamper with notification messages (revocation successful, etc). Therefore, we assume that Donna will not behave this way. It should be noted that with the same behavior, Donna could cripple any existing mail system. Our system is not resistant to such a powerful attacker (nor is any mail system that we know).

5.2 The protocol

In the following protocol, we assume that f and h are cryptographically strong one-way functions. h is also a hash function, as it is applied to variable-length messages. f is applied to small, fixed-length messages consisting of a random string and a short fixed-size message. In practice, MD5 [12] and DES [9] could be used to implement h and f, respectively. We also assume that each mail user agent (MUA) has a data structure called a *revocation table*, where it stores some useful information about messages it has received.

Step 1: send Alice sends a message to Bob's MUA.

- Alice generates a random string, k.
- Alice computes $x = f(k, h(M))$ and sends M, x to MUA. She then stores M and k for future use.
- MUA checks for x in the *revocation table*, to make sure that this message is not a replay. If x is not found, then MUA stores M in a regular spool file. It then computes $h(M)$, creates a revocation table entry, and stores $x, h(M)$ there, with a pointer to M in the spool file. If x is found, then the message is a replay and it is logged and ignored.

The result of Step 1 is illustrated in Figure 2.

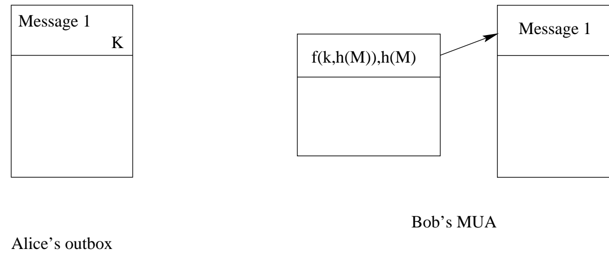

Fig. 2. Step 1 of Level 1 This figure illustrates the information that is stored by Alice and Bob's MUA after step 1 of the level 1 protocol is executed.

Step 2: check Alice checks to see if Bob has read the message, M. This step can be executed any number of times (or not at all) before Step 3.

- Alice generates a new random string, k'.
- Alice recomputes $x = f(k, h(M))$ using the stored k, computes $y = f(k', h(M))$, and sends k, x, y and a keyword *check* to Bob's MUA. She then replaces k by k' in her outbox.
- MUA receives k, x, y, and checks for x in the revocation table. If x is not found, MUA replies that M has been seen.
- If x is found, MUA computes $f(k, h(M))$, using k from the message it just received and $h(M)$ from the revocation table. It then verifies that $f(k, h(M))$ matches the x just received. If it does *not* match (i.e. MUA received an invalid key k), MUA replies that M has been seen (the reason for this reply is explained in the next section).
- If $f(k, h(M))$ matches x (i.e. the key k is valid), then MUA replaces x by y in the revocation table and replies to Alice that the message has not been seen.
- If M has been seen, then Alice erases M and k' from her outbox.

The result of step 2 is shown in Figure 3.

Step 3: revoke Alice revokes the message, M.

- Alice sends her current stored string, k (which may be different than the original k in step 1, if step 2 was executed) and $x = f(k, h(M))$ to MUA along with a keyword *revoke*.
- Bob's MUA looks for an entry in the revocation table containing x. If one is found, MUA computes $f(k, h(M))$ using the key just received and $h(M)$ in the revocation table. If it matches x, then MUA removes M from the spool file. MUA also removes the entry $x, h(M)$ from the revocation table. Then, Alice is notified that M was removed successfully.
- If x is not found in the revocation table, then MUA notifies Alice that the message has already been seen by Bob.
- Alice then removes M and k from her outbox.

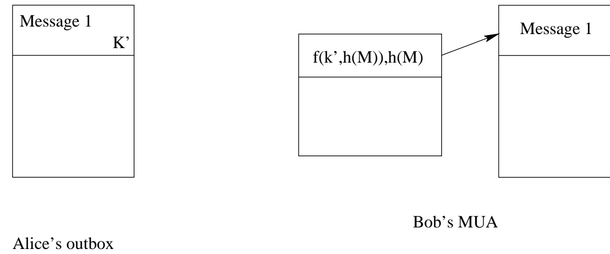

Fig. 3. Step 2 of Level 1 This figure illustrates the information that is stored by Alice and Bob's MUA after step 2 of the level 1 protocol is executed.

Step 4: read Bob reads the message, M. This step assumes that there is some message M that has not been revoked and that Bob has not read. If there is no such message, MUA simply replies that there is no mail.

- Bob requests his mail from MUA.
- MUA sends M to Bob, removes $x, h(M)$ from the revocation table, removes M from the spool file, and sends $h(M)$ as a *receipt* to Alice that M was read.
- Alice receives the receipt for M and searches backwards in her outbox comparing the hash of each message to $h(M)$.
- Alice removes M and k from her outbox.

5.3 Security considerations

We now discuss the security concerns that shaped the protocol described above. The main concern is that a malicious eavesdropper, Charlie, should not be able to revoke a message sent by Alice (Goal 7). Furthermore, Charlie should not be able to check whether Alice's mail was read by Bob. We assume that Charlie may not alter messages in transit. However, he may read messages in transit and send new messages if he so desires. We begin by explaining why our protocol satisfies Goal 7. We then move on to explain other aspects of the protocol.

When Alice first sends a message to Bob she picks a random key k. She then applies a one-way function to k and obtains x. We assume that given x it is intractable to determine k. This is a standard cryptographic assumption which is believed to be satisfied by various potential one-way functions [15]. Recall that Alice sends x along with the message M. Bob stores x in his *revocation table.* Since messages cannot be corrupted in transit the link between x and M cannot be broken by Charlie. Alice keeps the value k hidden in her private outbox. Since Alice is the only one who knows the value k she can prove ownership of the message M. On the other hand Charlie cannot determine the value k unless he is able to invert the one-way function.

The discussion above shows that in both the check and revoke steps Alice may prove ownership of the message M to Bob by revealing the secret key k. This prevents Charlie from either revoking or checking on a message belonging

to Alice. Of course once the key k is revealed a new secret key must be generated. For this reason during the check step Alice generates a new secret key, k', and sends its hash to Bob. This is in the same spirit as one time password systems [13] such as S/KEY™[7].

The protocol for checking whether a message has been read contains an interesting subtelty. Recall that after the message is read all information regarding the message is erased. At this point, one can not verify that the party performing the check is indeed the owner of the message. In other words, there is no way to prevent Charlie from learning the fact that a message has been read. Indeed, after the message is read, when Charlie performs a check he receives notification saying "the message has been seen". Before the message is read Charlie can not perform the check protocol since he does not possess the secret key k proving ownership of the message. However, observe that when Charlie runs through the check protocol using an invalid key the notification sent to him is still "the message has been seen". Consequently, the response Charlie receives gives him no information as to whether the message has been read or not. On the other hand, Alice, who knows the secret key k, will receive notification "the message has not been seen" when she performs a check before the message is read. Thus, by setting the reply messages appropriately we avoid the difficulty in verifying ownership of the message after it has been read.

Notice that in Step 1 when Alice applies the one-way function to k she is actually evaluating the function $x = f(k, h(M))$. There are several reasons for including the hash of the message $h(M)$ as input to the one-way function. Conceptually it creates a link between the message M and the secret key k. Furthermore, recall that the sender ID and the transmission time are parts of the message M (as SMTP headers). As a result the value x depends on those parameters as well.

Our protocol would run into unexpected behavior if two different messages hashed to the same value x. More precisely, we assume that two different pairs (k, M) and (k', M') satisfy $f(k, h(M)) = f(k', h(M'))$ with negligible probability. This can be achieved by assuming that $x = f(k, h(M))$ is chosen from a large enough space; we chose 512 bits for our implementation. For this reason if ever $f(k, h(M)) = f(k', h(M'))$ the protocol may safely assume that $k = k'$ and $M = M'$. For instance, in Step 1, Bob's MUA checks if the received value x already appears in the revocation table. If it does, the MUA assumes that the received message is a replay of an existing message. Consequently, the new incoming message is dropped.

This concludes the security considerations in our protocol. We now discuss some weaknesses of this level 1 protocol. The protocol only works when both parties Alice and Bob play by the rules. On a DAM system (See section 2), Bob can easily interfere with his MUA and prevent messages from being revoked. For instance, a program written by Bob may periodically copy incoming messages from his spool file. Doing so will prevent the MUA from revoking messages. (This is not a problem in the LIM model.) In our level 2 protocol, even if Bob

[3] *S/KEY* is a trademark of Bellcore.

interferes with his MUA, he cannot prevent e-mail revocation. (We discuss this in Section 3.)

It should also be pointed that our system does not prevent mail message forgery. For instance, Charlie may prevent Alice from revoking a message by applying the following strategy: when Alice first sends a message, Charlie records it. When Charlie notices that Alice chose to revoke the message he resends the recorded message to Bob pretending that it is actually sent from Alice. This way the original message is stored on Bob's machine as if it was sent from Alice. This replay attack in effect prevents Alice from revoking the message. To prevent this attack one needs the ability to authenticate the identity of the sender. This is addressed in our level 3 protocol which assumes a public key infrastructure.

As a final point we note that throughout the section we assume Alice has a secure pseudo-random number generator. That is, no third party can predict any of the bits generated by Alice's generator. See [5] for a discussion on how to generate secure random bits in software.

5.4 Implementation

We implemented the level 1 protocol. The implementation applies to the DAM model (e.g. UNIX systems), and we assume that all parties use the sendmail daemon.

Environmental Assumptions In our implementation Bob's MUA is sendmail, Mailx is the local mail delivery agent, and that Alice's MUA is a form of MH mail. We assume a standard Unix spool file; Each message begins with "From " and LF is used as a delimiter.

Program Overview The implementation is composed of a series of scripts. Alice's scripts have two functions: attaching revocation headers to outgoing mail and sending check and revoke commands. Bob uses a mail filter script to verify and execute check and revoke commands.

Alice must initially generate a random seed and sequence number. To send a revocable message, Alice composes a message with a special `-revoke` argument which causes the mailer to compute x and attach revocation headers to the message draft. Alice's mailer then sends the message to Bob and saves a copy in her `revoke` folder. Additionally the secret key k is stored as a mail-header as part of the message in the `revoke` folder.

If a receiver is not configured to accept revocable mail, revocation commands are simply appended to the spool file. Bob's spool file holds the incoming message, but if Alice generates a check or revoke request, Bob's local mail handler processes the request message instead of storing it in the spool file. If the revocable mail message has already been read, Alice is told that the message "has been read". Otherwise, Bob's mail filter will execute a script to perform the check or revoke. Alice is given the results of the request (eg, request succeeded).

Alice uses the traditional MH `scan +revoke` command to view revocable messages. She can scroll through the messages and look at the subject and the text of the messages. To execute steps 2 or 3 of the protocol, she issues a new command, either `revoke_send.pl check` <`msgs`> or `revoke_send.pl revoke` <`msgs`>. For users of xmh, it is trivial to add a button to the graphical user interface to 'revoke' or 'check' messages. For checks and revocations, Alice's mailer will send a message, "(`check/revoke`: k,x,y" to Bob, whose `.forward` and `.maildelivery` files will filter the message via `slocal` and eventually run the script associated with the request.

Design Rationale & Alternatives The protocol was implemented in Perl for two reasons. First, Perl has a powerful regular expression pattern matching mechanism. This is desirable since a significant amount of string comparisons are made. Second, Perl is portable to all modern UNIX operating systems. This allows the implementation to be used on multiple platforms without any tweaking.

The one minor disadvantage of using Perl is that the code must be reinterpretted each time Bob receives a revocation request and each time Alice sends revocable mail.[4] Using a compiled language such as C would win speedwise, but the recompilation and platform dependent issues are not worth it for this simple implementation.

Before choosing, MH, we briefly looked at BSD v.4 `mailx` (Mail) and `pine` v3.95. Both of these mailers proved to be hard to modify and monolithic in design. Instead of dealing with platform compatibility issues and source code patches, we chose MH mail because of it's modularity, popularity, and its many GUI's (mh-e, xmh, exmh)[10]. Our implementation is tailored to MH, but it serves the purpose of demonstrating a working protocol and exposing possible problems. Unlike most mailers (pine, elm, xmail, etc), each of MH's commands is a separate program run from the shell prompt. This modularity and the flexibility of parameter files allow the revocation protocol to be implemented without recompiling any source code and giving the revocation routines a sense of transparency to the user.

The combination of MH's modularity and Perl's portability allow a user to seamlessly use the protocol on most UNIX machines. Furthermore, revocable email can be sent without needing system administrators.

Originally we considered placing a wrapper around sendmail, but this would have required tweaking the sendmail configuration. It was much easier to place the wrapper around MH mail, because unlike sendmail's harry configuration files, MH provides simple and convenient methods to add new functionality. Users can install and use the program more easily.

We shifted to the per-user revocation tables instead of the whole system tables. A large database of all revocable mail would require more infrastructure. The spool files for each user are natural places to store information.

[4] However a Perl compiler is in its alpha stages.

Defining the actual message to hash turned out to be more difficult than expected because of many details specific to SMTP [11]. An encapsulation mechanism similar to PGP ASCII armour [1] was considered, but we wanted the presence of cryptography to be less visible to the user. Instead of mandating special message delimiters, we took the more complicated approach of specifying exactly what constitutes a message body and headers. This does not entirely solve the problem, but for the most part, messages are interpreted the same on both the sending and receiving side.

Sendmail and other MUA's place a ">" in front of each instance of "From", followed by a space, at the start of a line in the message body. This can cause problems as the body to be hashed changes during the transfer. For example, the message body in:

```
To: alice@school.Edu,  bob@company.com,
cc: person@site.net
Subject: Beware of rusty U-Locks

Hi guys,
From now on, beware of evil U-Locks.
-Donna
```

would first be translated to:

```
Hi guys,
>From now on, beware of evil U-Locks.
-Donna
```

To thwart this, the revocation process adds the ">" before sending the message, thereby preventing the MUA from changing anything. Another problem is that addresses may have domain names appended by sendmail. The capitalization and whitespace will vary from sender to receiver. Thus, we had to be very careful to only hash fields that do not change in transit.

Spool files composition is not defined by an Internet standard. Delimiters vary from LF's (mailx) to several C-A's (MDDF). Messages typically begin with "From " and end with a LF. But Bob's revocation process needs to know what delimits the message. We assume that the mailx local delivery program is used. By default, messages are separated by LF's.

Module Descriptions *deslib.perl* contains Dennis Ferguson's implementation of DES routines and `string2key`.

revokelib.pl contains many lower-level cryptographic routines used by other scripts. It is the sole program to decide how to hash a message. The key generation routines are contained here: a new key is generated by taking the output of a DES encryption on the sequence number with the random seed. The sequence number is then incremented and stored with the new key in `.revoke`.

revoke_init.pl creates the initial random seed and sequence number for Alice. The seed is generated by folding an input string and a function of the machine

state into a DES key by the Kerberos v4 string-to-key function. The seed and the sequence number zero are then stored in Alice's /Mail/.revoke file.

Alice uses *revoke_header.pl* to attach special headers to an MH draft file. The required "to" header and optional "subject" header are saved in the draft, but the "date" and the "from" headers are inserted later in the transfer process.

We store the information contained in the revocation table (see Figure 2) as mail-headers embedded in the message. For instance, when Alice sends a message to Bob she embeds the header X-Revoke-Hash: x in the message.

To send a revocable/checkable message, the headers:

```
X-Revoke-Date: <DATE-AND-TIME>
X-Revoke-Hash: <ENCRYPTED-HASH>
```

must be in the mail header. This will instruct the receiving end that this message can be checked for delivery and revoked from the spool file. A copy of the message with the additional X-Revoke-Key: <key> header is placed in Alice's revoke folder.

The string to be hashed is defined as the concatenation of the X-Revoke-From field body, X-Revoke-Date field body, subject field body, and the message body.

revoke_send.pl and mysend.sh are connected with MH to send revocation requests: *check* and *revoke*:

Check request:

```
X-Revoke-Command: check
X-Revoke-Key: <DES-KEY>
X-Revoke-Hash: <ENCRYPTED-HASH>
X-Revoke-New-Hash <NEW-DES-KEY>
```

Revoke request:

```
X-Revoke-Command: revoke
X-Revoke-Key: <DES-KEY>
X-Revoke-Hash: <ENCRYPTED-HASH>
X-Revoke-New-Hash: <NEW-DES-KEY>
```

revoke_receive.pl verifies and executes revocation commands. It is responsible for sending reply messages to Alice and removing messages from Bob's spool.

Known Problems In our current implementation, messages with multiple recipients cannot be revoked. Some modifications to check/revoke script are needed. We plan to implement this soon.

This implementation does not work with the MH drafts folder. The program deals only with single draft files (eg, Mail/draft). Finally, we do not use file locking. A robust implementation would require this for multiple revocations to take place concurrently.

Notification programs such as biff and xbiff display information to the user that he has a message, sometimes including the first few lines of the message. Implementations of revocation should take these into account. In our current implementation, we do nothing to prevent notification programs from displaying this information.

6 Level 2

This section presents a more secure protocol for revoking electronic mail.

6.1 Security model

In level 2, we assume the existence of an honest third party, T, and that senders and receivers do not necessarily trust each other. Our security requirement is that as long as T follows the protocol, then requirements 1-5,7,9-10 of Section 3 are met. We do not protect against Donna, an active attacker who can modify messages in transit. However, requirement 8 holds as long as Donna does not modify the message at the time it is sent. Also, in the DAM model, requirement 6 cannot be met.

We do not assume that there is an existing relationship between the sender, the receiver or T. Therefore, we cannot defend against somebody impersonating T by, say, spoofing DNS. The level 3 protocol defends against these attacks (see Section 7). The main difference between level 2 and level 1 is that even if Bob tries to cheat, he will not be able to defeat the security requirements. Level 2 is especially well suited for the DAM model.

6.2 The honest third party

We specify an *honest* third party, T, as opposed to a *trusted* third party because this party is only trusted to behave honestly, follow a protocol and not reveal its secrets. The term *trusted third party* is generally used to refer to a party that is trusted to verify identities and issue statements binding entities to keys. Our assumptions about T are weaker.

T consists of two services that could be offered by separate entities (if necessary). The first service is a *mix* that is used to protect messages from traffic analysis. This concept was first proposed by Chaum [4], and an example of such a system is BABEL [6], which also allows users to send anonymous mail with anonymous return addresses. The purpose of the mix is to prevent Bob from using traffic analysis to subvert requirement 5.

The second role of T is to provide a server, S, that can be trusted to store secret keys that are only released under the right conditions. For greater reliability and availability, this service could be distributed to several machines using secret sharing and threshold schemes [14]. Both the mix and the server may consist of any number of geographically dispersed machines.

6.3 Infrastructure assumption

In level 2, we assume that there is no existing relationship between Alice and Bob (i.e. no shared secrets or public keys). In addition, whenever Alice sends a message to Bob, Charlie sees everything that was sent. Therefore, there is nothing that can differentiate Charlie from Bob from the point of view of the secure server. Our protocol requires Bob to request a decryption key from the honest server in order to read messages. Since there is nothing that Bob can do that Charlie cannot do, Charlie can make the same request to the honest server. Thus, Charlie can cause the honest server to mistakenly believe that Bob has read a message.

The only way to prevent Charlie from impersonating Bob to the honest server is to assume that Bob can somehow authenticate himself to the honest server. To do this, we assume a small amount of infrastructure. In particular, we require that each site register a secret key, w, with the honest server S (the second service provided by T). This is not such a great burden considering that an administrator will have to upgrade the mail software anyway to use our protocol. Users need not be aware that there is a site-specific secret key shared with S.

As mentioned in Section 2, the MUA in the LIM model and the mail daemon in the DAM model are trusted. Therefore, we assume that the mail daemon in either case can be trusted with secrets, such as w, that it can keep from the users.

6.4 The protocol

In the following protocol, f, h, and MUA are defined as in section 5.2; Mix and S are the two components of the honest third party as defined in Section 6.2. And w is the secret key shared by the trusted component of Bob's MUA and S. k, k', and k'' are random strings generated by Alice, and $\{M\}_k$ represents the message, M, encrypted under key k with a symmetric cipher. When M is encrypted, the entire message, including the headers that direct the message to Bob, is encrypted. Then, new headers are added that direct the message to a special address at Bob's MUA, where it is processed appropriately. This ensures that each message will be unique because it will have information such as the date, time, sender and receiver.

The following protocol is explained in detail in the subsequent section:

Step 1: send Alice sends a message to Bob's MUA.

- Alice generates k, k', and k'' and stores them, along with M, in her outbox.
- Alice encrypts M under k and computes $x = f(k', h(\{M\}_k))$ and $x' = f(k'', h(\{M\}_k))$.
- Alice sends x,$\{M\}_k$ to Mix, which forwards it to Bob's MUA.
- Alice then sends $k, h(\{M\}_k), x'$ to S.

– MUA checks for x in the *revocation table*, to make sure that this message is not a replay. If x is not found, then MUA stores $\{M\}_k$ in a special spool file. Next, MUA computes $y = f(w, h(\{M\}_k))$. It then creates a revocation table entry and stores $x, h(\{M\}_k), y$ there, with a pointer to $\{M\}_k$ in the spool file.
– S stores the triple $k, h(\{M\}_k), x'$ in a revocation table.

The result of Step 1 is illustrated in Figure 4.

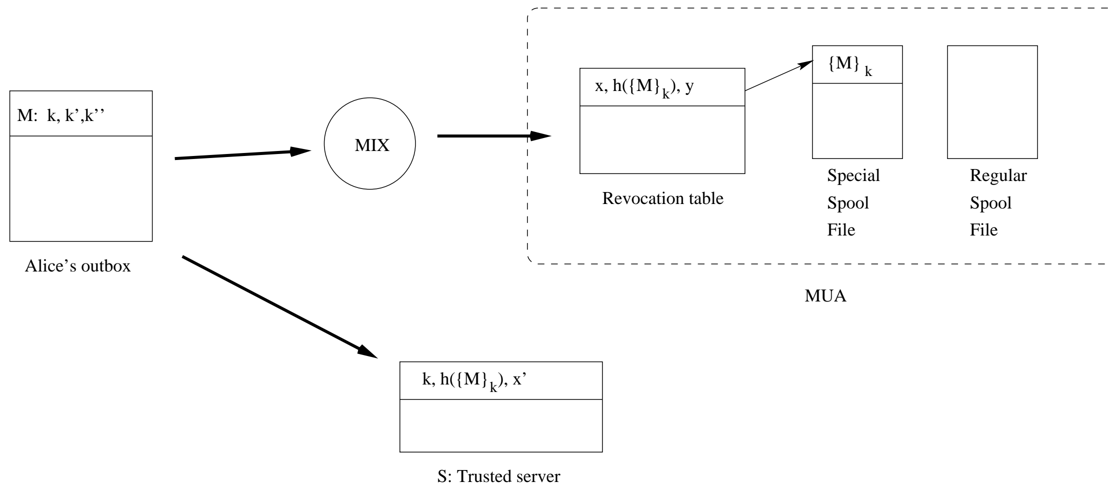

Fig. 4. Step 1 of Level 1 Step 1 of Level 2 This figure illustrates the information that is stored in Alice's outbox, at Bob's MUA, and at the honest server, S after Step 1 is completed. The thick lines represent message flows. The MUA has a regular spool file which it uses to process e-mail that is not sent using our system and a special spool file that is used for e-mail that can potentially be revoked.

Step 2: check Alice checks to see if Bob has read the message, M.

– Alice generates a new random string, l.
– Alice replaces her stored value of k'' with l and sends $k'', f(l, h(\{M\}_k)), h(\{M\}_k)$ and the keyword *check* to S.
– S computes $z = f(k'', h(\{M\}_k))$ and searches for a triple $(*, h(\{M\}_k), z)$ in its revocation table (here $*$ means "don't care"). If no such triple exists in S's revocation table, S replies "message has been seen."
– If the triple is found, S replaces z with the value $f(l, h(\{M\}_k))$ and replies "message has not been seen."
– If the answer is "message has been seen," then Alice removes M and its keys from her outbox.

Step 3: revoke Alice revokes the message, M.

- Alice sends k' and $x = f(k', h(\{M\}_k)$ to MUA (via mix), along with the keyword "revoke". Alice also sends $k'', h(\{M\}_k)$ to S, along with the keyword "revoke".
- MUA looks for an entry in the revocation table containing x. If one is found, MUA computes $f(k', h(\{M\}_k)$ using the key just received and $h(\{M\}_k$ in the revocation table. If it matches x, then MUA removes $\{M\}_k$ from the spool file. MUA also removes the entry, $x, h(\{M\}_k$ from the revocation table.
- S computes $z = f(k'', h(\{M\}_k))$ and searches for a triple $(*, h(\{M\}_k), z)$ in its revocation table (here $*$ means "don't care"). If a match is found, then the entire entry, $k, h(\{M\}_k), z$ is removed. S then notifies Alice that k has been successfully revoked. If no matching entry is found then Alice is notified that the revocation attempt was unsuccessful (presumable, the message has been either read or already revoked).
- Alice then removes M and the associated keys from her outbox.

Step 4: read Bob reads the message, M.

- Bob requests his mail from MUA.
- MUA sends $y, h(\{M\}_k)$ to S.
- S looks up $h(\{M\}_k)$ in the revocation table. It then computes $f(w, h(\{M\}_k)$ and compares it to y. If they are equal, S sends k to MUA. S then sends a notification to Alice that the message encrypted under k has been read and removes the entry from the revocation table. If the check for y does not compute correctly, it is logged and ignored.
- MUA decrypts $\{M\}_k$ using k and sends M to Bob. MUA then removes the entries from its revocation and spool files.
- Alice searches her outbox for an entry containing the k received from S and removes the message and its associated keys from her outbox.

6.5 Security considerations

The advantage of our level 2 protocol over level 1 is that Goals 1-5 are satisfied even when the receiving party, Bob, does not play by the rules. More precisely, we show that even if Bob has full control over his MUA he cannot defeat goals 1-5. Note that in the DAM model, Bob has full control over his MUA since he can browse through his mailbox using any mail reader of his choice. For instance, Bob may use a mail reader which does not inform Alice that the mail was read. As a result Alice may successfully revoke a message even though the message has already been read by Bob. This is a security concern which was purposely ignored in our level 1 protocol and is addressed in level 2. Notice that these problems do not exist in the LIM model, since Bob's MUA is part of the LIM system and hence Bob has no control over it. Consequently, in the LIM model level 1 may be sufficient.

Our level 2 protocol involves several authentication keys. We first explain the purpose of each of these keys and then describe the considerations used in designing the protocol.

- The key k is used to encrypt the message M. Bob cannot read the message until he receives the decryption key.
- The key k' is used by Alice to prove ownership of the message to Bob's MUA. This key is used to notify Bob that the message has been revoked and that he may delete all data associated with it.
- The key k'' is used by Alice to prove ownership of the message to the honest server S. Alice uses this key when she either requests the server S to revoke the message or when she checks if the message has already been read.
- The key w is a secret key shared by Bob's *site* and S. It is used to enable Bob to prove to S that he is the recipient of the message. We emphasize the fact that we do not require a shared secret per user, but rather per site. We refer to this as a *weak infrastructure assumption*.

Throughout the section we consider the message as read by Bob as soon as S sends the decryption key, k, to Bob. This is a reasonable assumption since if all parties follow the protocol then Bob requests the key k when he is ready to read the message.

When Alice sends a message to Bob she first encrypts the message using a random key k and sends the encrypted message to Bob using a MIX. The purpose of the MIX is to mask the identity of Alice. As a result Bob can infer that some message was received; however Bob cannot infer the identity of the sender as required by Goal 5. Clearly at this point Bob cannot understand the message since he does not know the encryption key k.

When Alice wishes to check if the message has been read by Bob she queries the server S to see if Bob has requested the decryption key k. The protocol used at this point is identical to the one used in the corresponding step of the level 1 protocol and the same design considerations of Section 5.3 apply to it. If Bob read the message then the server S will correctly inform Alice of this fact, as required by Goals 1 and 3. The converse, Goal 4, requires a more involved argument. The difficulty is that an adversary, Charlie, may ask S to send him the key k. In doing so Charlie fools S into thinking the message has been read (by Bob). To prevent this we must provide Bob with means to authenticate himself to S.

Providing Bob with means of authenticating himself to S seems impossible at first without S and Bob sharing a common secret; if all information is public, Charlie can impersonate Bob by simulating Bob's actions. However, we wish to avoid giving secret keys to every user. Fortunately the authentication problem can be solved using one shared secret key, w, per site. The key w is known only to Bob's MUA (e.g. the mail daemon running on Bob's machine) and is inaccessible to anyone else. Recall that in step 1 of the protocol, when Bob's MUA receives a message $\{M\}_k$ it first computes $y = f(w, h(\{M\}_k))$ and stores it in Bob's revocation table along with $\{M\}_k$. Notice that no one other than Bob or his

MUA has access to this revocation table. Clearly the server S is also capable of computing y. When Bob requests the decryption key k from S he authenticates himself by sending S the value y.

We claim that Charlie cannot determine the value y and consequently he cannot fool S into sending him the key k. This will prove the validity of S's reply in Step 2 of the protocol and the correctness of the receipt sent back to Alice in Step 4. Observe that during previous invocations of the protocol Charlie observes the values $y_1 = f(w, h(\{M_1\}_k)), \ldots, y_n = f(w, h(\{M_n\}_k))$ for messages $M_1, \ldots, M_n$. Since h is collision secure we may assume that $h(\{M_1\}_k), \ldots, h(\{M_n\}_k), h(\{M\}_k)$ are distinct. Charlie then has to compute the value $y = f(w, h(\{M\}_k))$ where M is the current message. Since f is a symmetric block cipher this is equivalent to decrypting the cipher-text $h(\{M\}_k)$ given the set of plain-text/cipher-text pairs $(y_1, h(\{M_n\}_k)), \ldots$. By definition, this cannot be done if f is a secure block cipher. Consequently, Charlie cannot fool S into thinking he is Bob. This proves that Goals 4 and 10 are fully satisfied.

When Alice chooses to revoke a message, she sends a notification to the server S asking it to erase the encryption key. By doing so the encrypted message sent earlier to Bob is now useless. In other words, Bob will never be able to read Alice's message. It follows that if Alice revokes her message, Bob will not be able to read the message. Hence, Goal 2 is satisfied

Our level 2 protocol requires the use of an honest server S to store encryption keys k. The storage requirements of the honest server are quite low. One may avoid using an honest server by making Alice the honest server. That is, when Bob wishes to read his mail he contacts Alice and asks for the decryption key. We avoid this design for several reasons, primarily for availability. It is quite likely that Bob may wish to read his mail when Alice's server is down. Without a trusted server, Bob would have to wait for Alice to come back on-line.

7 Level 3

In level 3, we assume that there is a public key infrastructure. Minimally, that means that the public key of some trusted authority is shared by all, and that the trusted authority has issued public key certificates for all parties. In practice, the only important thing is that each party has a valid copy of every other party's public key. To date, systems requiring a public key infrastructure do not scale well. The problem is that neither the PEM model [2] nor the PGP model [17] are acceptable as global solutions. In the PEM model, there is a strict certification hierarchy. There are two problems with this model. One is that there are many people who are unwilling to trust a single *root* of the certification tree. The other problem is that some certification authorities may be more competent than others at certifying their users.

In the PGP model, people are responsible for maintaining their own set of others that they trust. Trust can propagate based on transitive trust. The PGP model does not scale because different people have different ideas of when the transitive relationship should hold.

The level 3 system has advantages over levels 1 and 2 because all messages can be signed as authentic and encrypted for confidentiality. Revocation is easier because every message can be authenticated. However, level 3 solutions can only be adopted on an *Intranet* or Communities of Interest (COI) scale. The protocols for this level could be easily built using existing technology.

8 Conclusions

This paper describes the security requirements for adding several new features to electronic mail. These features allow users to revoke message that they have previously sent, receive notification when messages are read, and check to see if messages have been seen yet. Careful analysis shows why the security properties of our protocols hold.

In the level 1 system, we assume that all parties comply with the rules of the protocol. This is natural and enforceable in the LIM model. Level 2 protocols are designed to withstand users who try to cheat, at the cost of requiring an honest third party and a weak infrastructure assumption. This protocol is designed for DAM systems. The level 3 protocol is described at a very high level as it is not applicable in the Internet until the public key infrastructure problem is solved.

We implemented the level 1 protocol.

References

1. D. Atkins, W. Stallings, and P. Zimmermann. Pgp message exchange formats. *RFC 1991*, August 1996.
2. D. Balenson. Privacy enhancement for Internet electronic mail: part iii—algorithms, modes, and identifiers. *RFC 1423*, February 1993.
3. CCITT. Message handling systems: System model–service elements. *CCITT Recommendations X.400*, 1984.
4. David Chaum. Untraceable electronic mail, return adresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, February 1981.
5. D. Eastlake, S. Crocker, and J. Schiller. Randomness recommendations for security. *RFC 1750*, December 1994.
6. Ceki Gulcu and Gene Tsudik. Mixing E-mail with BABEL. *Symposium on Network and Distributed System Security*, pages 2–16, February 1996.
7. Neil Haller. The s/key(tm) one-time password system. *Symposium on Network and Distributed System Security*, pages 151–157, February 1994.
8. MTS volume 23: Messaging and conferencing in MTS, February 1991.
9. National Bureau of Standards. Data encryption standard. *Federal Information Processing Standards Publication*, 1(46), 1977.
10. J. Peek. MH and xmh: Email for users and programmers, 1996.
11. Jonathan B. Postel. Simple mail transfer protocol. *RFC 821*, August 1982.
12. R. Rivest. The md5 message digest algorithm. *RFC 1321*, April 1992.
13. Aviel D. Rubin. Independent one-time passwords. *USENIX Journal of Computing Systems*, 9(1), 1996.
14. A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, November 1979.

15. Douglas Stinson. *Cryptography: Theory and Practice*. CRC Press, Inc, 1995.
16. Domestic mail manual, September 1995.
17. P. Zimmerman. Pgp user's guide. December 4, 1992.

This article was processed using the LaTeX macro package with LLNCS style