APPENDIX D

Two Email Proxies

Software for encrypting email messages has been widely available for more than 15 years, but the email-using public has failed to adopt secure messaging. This failure can be explained through a combination of technical, community, and usability factors.

As part of the work on this thesis, two email proxies were designed based on the design principles and patterns outlined in Chapters 1 and 10 of this thesis. Those proxies, Stream and CoPilot, are based on the same design principles but were created to serve different purposes:

- **Stream:** Written in C++ and deployed on MacOS and FreeBSD, Stream was designed and written to be used in day-to-day operations. It functions as an POP and SMTP proxy, and could also be used as a filter on a mail server.
- **CoPilot:** Written in python and shell script, CoPilot was created specifically for the purpose of conducing the *Johnny 2* user test. As a result, CoPilot actually had to be designed twice. First a *theoretical design* was created to reflect how a system like CoPilot would actually be written and deployed. But because CoPilot was used in a user test, an *implemented design* also had to be created for the actual code that would be used to conduct the study.

  The theoretical CoPilot system, like Stream, was designed to be deployable as a POP and SMTP proxy, as a procmail filter, or as an Outlook Express plug-in. The implemented CoPilot used in the user study was written in a combination of Python and shell scripts, it's user interface was framed HTML messages, and it's sole purpose was to create those messages for the user study.

Table D.1 compared Stream, the theoretical CoPilot design, and the practical CoPilot design.

	Stream (implemented)	CoPilot (theoretical)	CoPilot (implemented)
Implementation Language	C++	C++/C#	Python & shell
Channel to User:	Subject: line rewriting	Subject: line rewriting -or- Outlook Express toolbar	Framed HTML
Cryptographic Engine:	PGP	S/MIME & PGP	S/MIME
Key distribution	Hidden in header	S/MIME attachment	S/MIME attachment

Table D.1: A comparison of Stream vs. CoPilot

D.1 Proxy Philosophy

Through an application of the GOOD SECURITY NOW principle, these proxies all implement a straightforward design philosophy that is designed to provide some security features for some email messages now—and as a result let some email pass without security processing—rather than trying to be an all-comprehensive email system that either secures all email now, or else provides military-grade authentication for some email tomorrow. For the email proxies, this philosophy can be distilled into several key points (patterns introduced in this thesis are noted where appropriate):

- Be unobtrusive—do not require an input from the user under normal circumstances. (Zero-click security.)
- Be informative—tell the user what is going on, and make it possible for the user to learn more. (Visibility of Actions.) EXPLICIT USER AUDIT
- If the user doesn't have a key, create one. CREATE KEYS WHEN NEEDED
- Sign all outgoing messages. SEND S/MIME-SIGNED EMAIL
- Attach the user's key to every outgoing message. LEVERAGE EXISTING IDENTIFICATION EMAIL-BASED IDENTIFICATION AND AUTHENTICATION
- If possible, seal outgoing messages for each recipient. GOOD SECURITY NOW
- Inform the user of extraordinary happenings, and give the user a chance to discover routine events.
- Do not cause significant usability problems for the recipients of the proxy user's messages, or who wish to send email to proxy user. NO EXTERNAL BURDEN

D.1.1 Philosophical justification

With a few notable exceptions, today's email systems force users on every messages they send whether that particular message will be sent with a digital signature and/or sealed for the recipient. One common justification for giving users such low-level control is that cryptographic protection is not always necessary—or even desired. Giving the user control ensures that the user will take the right action.

The problem with this common justification is that it assumes a user who is unrealistically educated, informed, and concerned.

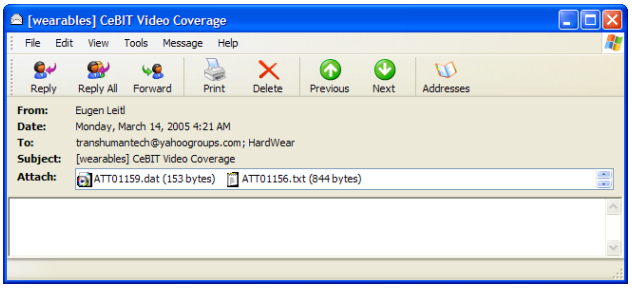

Figure D-1: Outlook Express 6 shows OpenPGP-signed messages as a blank message with two accompanying attachments: one for the original message, and one for the signature.

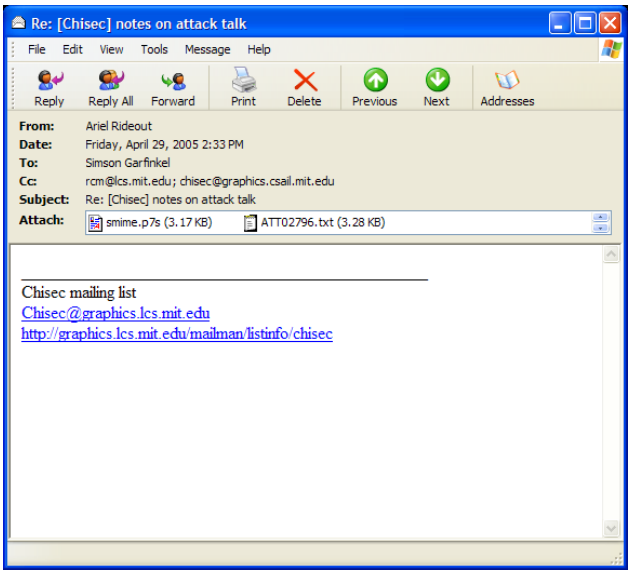

Figure D-2: When an S/MIME-signed message passes through the Mailman mailing list software, Outlook Express 6 displays the result as a brief message containing the mailing list "boilerplate" with two attachments: one for the original message, and one for the S/MIME signature. The reason for this failure is that mailman adds its boilerplate by taking the original S/MIME message, encapsulating it in an S/MIME envelope, and then adding a new `text/plain` part that contains the boilerplate. Outlook Express 6's S/MIME implementation does not understand this second-level encapsulation and act appropriately, even though it is allowable by the standard.

For example, if a message signed with an OpenPGP signature is sent to an Outlook Express user, the message and its signature both appear as attachments on an empty message (Figure D-1). If an S/MIME-signed message is sent to a Mailman mailing list that adds a footer as an attachment, then Outlook Express will display both the original message and the S/MIME signature as attachments on the mailing list footer—*even though Outlook Express allegedly implements the S/MIME standard!* (Figure D-2) This is because the Outlook Express S/MIME implementation is incomplete. Ordinary users are in no position to learn these rules, learn the capabilities of their correspondents, and then consistently apply the rules as needed. More likely, they will stop using the mail security technology entirely.

The proxies developed for this thesis take a different approach. They remove decision making from the user, and instead attempt to "do the right thing" based on the information that they have. The theory is that the proxy is in a better position to notice and track specifics such as email clients used by correspondents, rather than forcing the user to remember such minutia and take it into account when each mail message is sent. When the proxy cannot determine if an email security capability can be used, it should fall and not use it, if there is a possibility that the use of the proxy will cause a burden to the user's correspondents.

D.1.2 Private key migration

When encryption keys for digital signatures and sealing are created dynamically on a client computer, there needs to be some provision for backing up these keys in a secure manner. If keys are to be backed up, then that backup has to happen either manually or automatically.

Programs like PGP provide manual systems for backing up and restoring both public and private keys. Whitten and Tygar tested PGP's facility for backing up keys and found it wanting.[WT98] Manual key escrow further violates one of the design principles of the proxies described in this appendix: if backup is manual, then the proxies cannot be unobtrusive. Outlook Express and Thunderbird provide for manual backing up and migration of private keys in PKCS12 files, but this functionality is difficult to use.

Instead, a variety of techniques were envisioned for automatically backing up the keys created by Stream and CoPilot:

- The most straightforward approach was for the proxy to e-mail to the user's own mail account a copy of the public and private key pairs. If there are multiple instances of the proxy downloading mail from the same mailbox—perhaps by using POP's "leave mail on server" option—then each of those instances would receive access to the same private key material. This approach implicitly trusts the maintainer of the email system with the user's private key. Such trust can be reduced by asking the user for a password when the proxy is installed, and using that password to encrypt the private key material before it is sent.
- If the proxy is downloading email from an IMAP server, then the key can be uploaded to the server as an attachment to a special message stored in the inbox.
- Finally, the key can be stored on an Internet-based synchronization service that is protected by an independent username/password service. This is the approach that Apple MacOS 10.4 takes for synchronizing usernames, passwords, public key certificates, and private keys stored in the Apple keychain through the ".Mac" online subscription service.

Migrating and protecting private keys needs to be an important part of any email security system. But according to the survey of Amazon.com merchants presented in this thesis, of the 414 people responding to the question, only 33% knew that they would be unable to access the content of an email message if they lost the private key needed to unseal it! (When only the 102 *users of cryptography* were considered, the number of those who realized that they needed to retain their key rose to just 56%.) Thus, automatic key migration needs to be part of any system that is intended for significant widespread use.

Although the proxies presented here did not implement key migration, such a system could easily be added.

D.2 Stream: A PGP Proxy

Stream operates as a filter on outgoing email messages through the use of an SMTP proxy, and on incoming email messages through the use of either a POP proxy or (in the case of IMAP), as a filter that can be used by procmail[vdBG05] or placed directly in a ".forward" file.

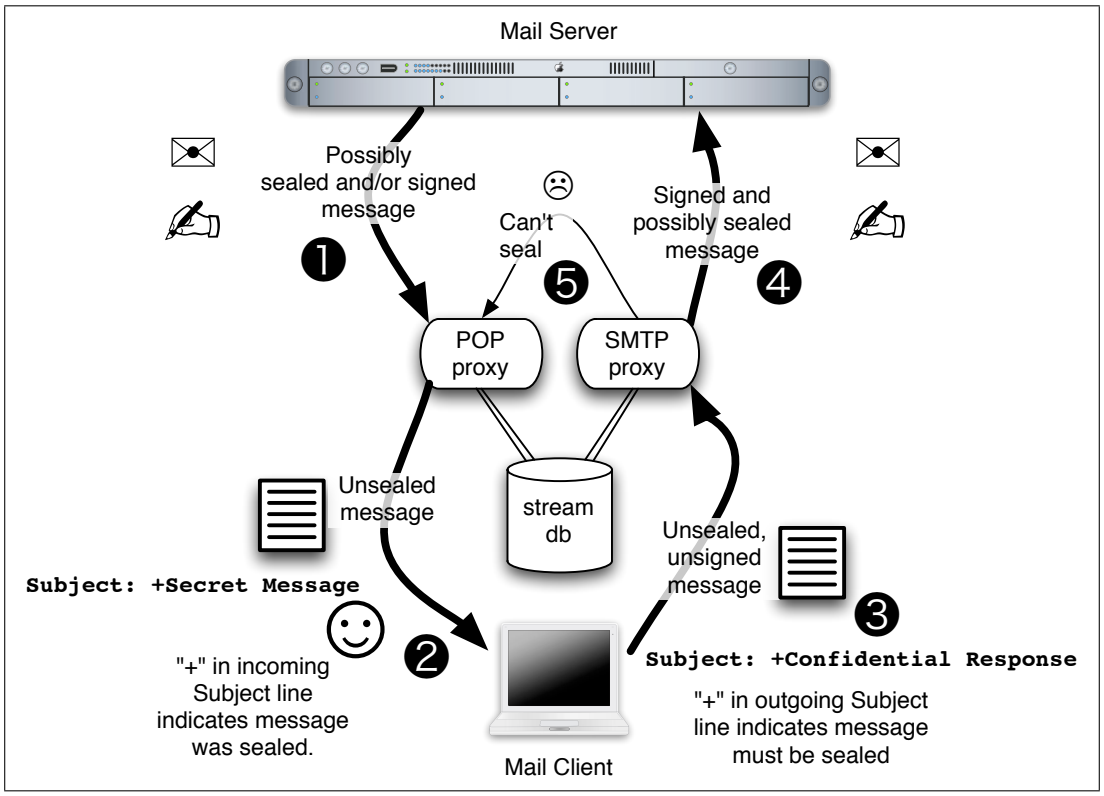

Figure D-3: The stream system design. ❶Messages are downloaded to the stream systems through the POP proxy, which unseals sealed messages and verifies the signatures of signed messages. ❷Unsealed messages are passed to the mail client. ❸Messages that are sent out from the mail client are signed and optionally sealed. ❹The signed and possibly sealed message is to the SMTP server, and from there, to the intended recipient. ❺If a message Subject: line contained the **mandatory encryption character** and Stream was unable to encrypt the message, the message is returned to the sender via the POP proxy.

D.2.1 Sending mail

As an outgoing filter, Stream automatically performs these actions for each outgoing message M:

1. Determines the sender's email address E.
2. Creates a public/private key pair for address E (K_E) if one does not exist.
3. Places a copy of K_E in M's message header using the approach described in Section 5.4.
4. Evaluates the recipients ($R_{1...n}$) of M:
   (a) If there are other recipients on the original message for which Stream has the keys on file:
      i. Those keys are extracted from the sender's PGP keychain and signed.
      ii. These signed keys are then embedded in the message's MIME headers.
   (b) If a public key (K_R) for R is known, Stream:
      i. Encapsulates M's original mail header within message M.
      ii. Adds to this encapsulated header the key fingerprint for each recipient's encryption key.

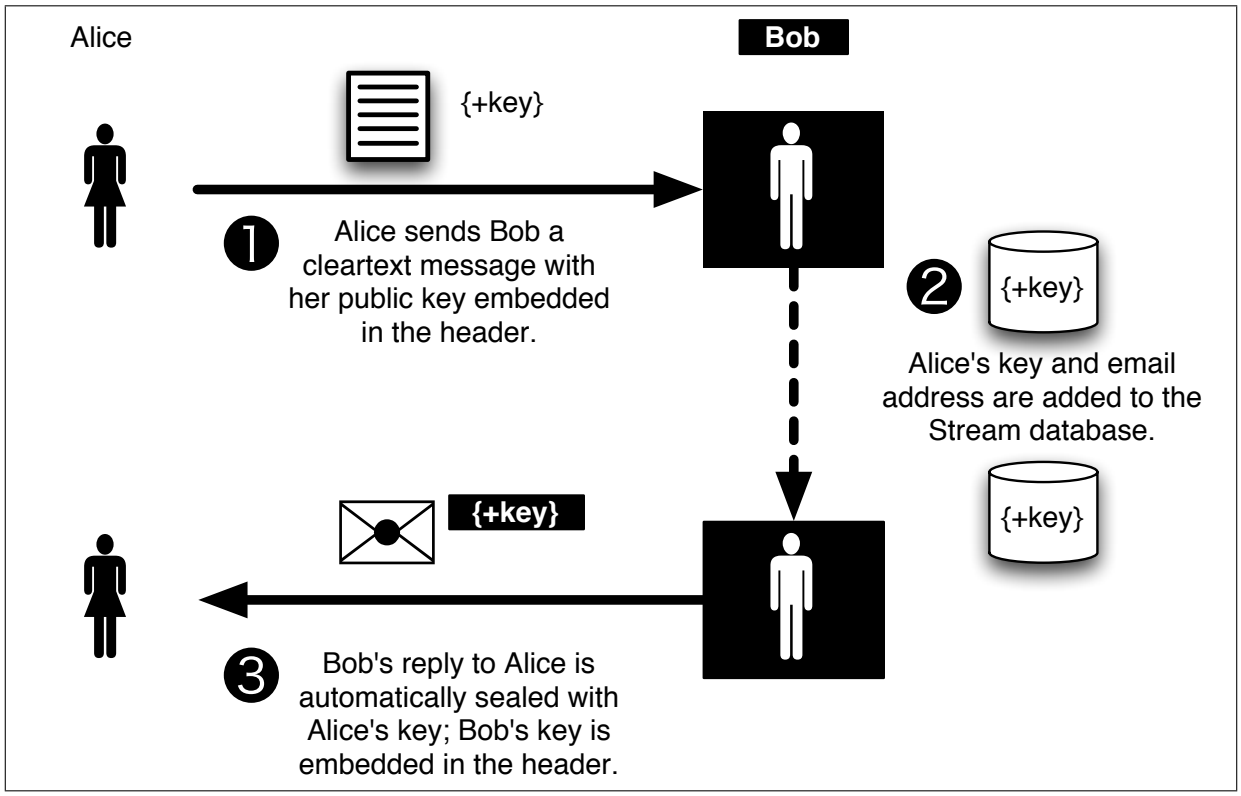

Figure D-4: Stream's key distribution system ensures that the first reply from a stream user to a second Stream users's message will be cryptographically sealed. ❶All stream messages include a copy of the sender's public key hidden in the MIME headers. ❷When a message containing a hidden key is received, Stream automatically incorporates the key into the program's key database. ❸When the recipient of the message (Bob) replies to the sender, the message is automatically encrypted by a copy of Stream that proxies the sender's outgoing messages.

iii. Creates a new sanitized mail header for message M containing a single To: address and a nondescript Subject: line.

iv. Encrypts M for the recipient and sends the message out through SMTP server SS.

(c) If the public key (K_R) is not known:

i. If the message Subject: line contains the **mandatory encryption character**, the message is sent back to the sender with a brief message added to the top indicating that the message could not be encrypted for recipient R.

ii. Otherwise, the message is sent to recipient R without first being sealed.

Stream provides opportunistic encryption: if the email message can be encrypted, it is. If it cannot be encrypted, it is sent without encryption. This behavior mimics the behavior of many encryption users: they use it if they can, but if they can't, they send their message anyway. However, Stream gives users a simple mechanism to override this behavior: a special character (currently the plus sign) is added to the beginning of the Subject: line.

D.2.2 Receiving mail

As an incoming filter, Stream automatically performs these actions on each incoming message M:

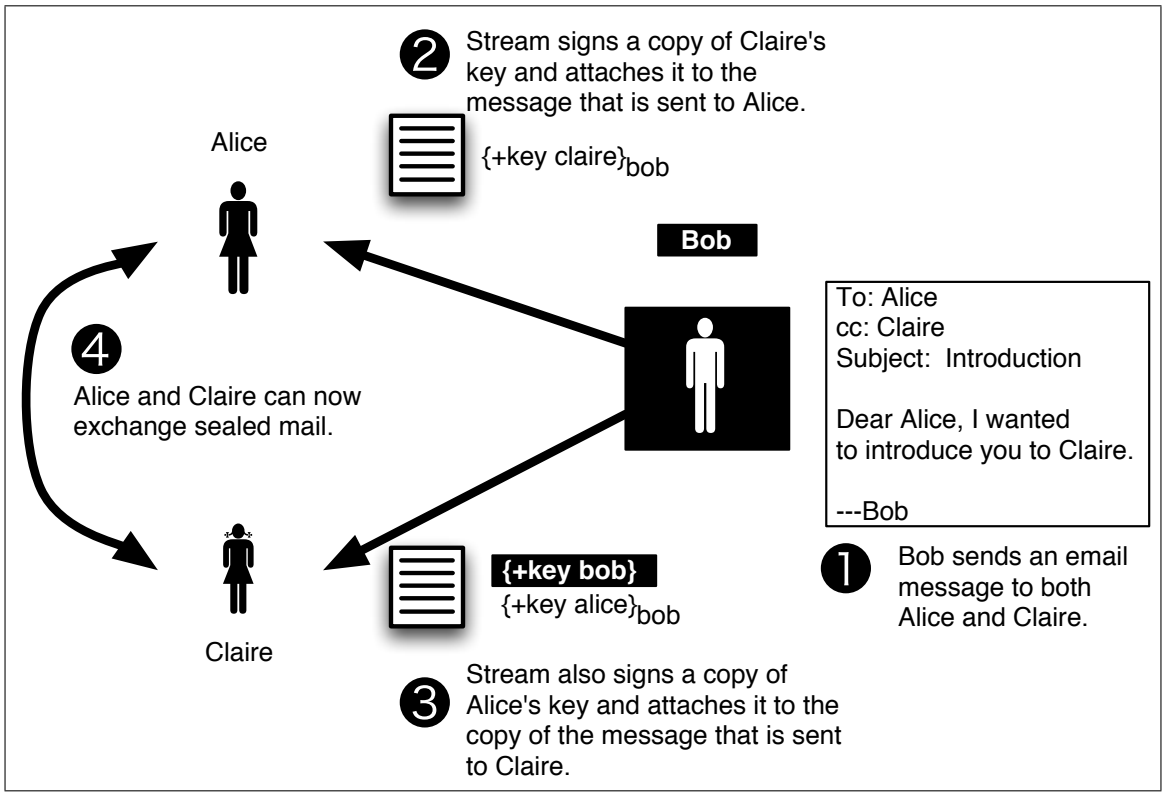

Figure D-5: Stream provides for the opportunistic distribution of keys and peer-to-peer cross-certification, building upon the PGP "web of trust."[Gar94] In this example, ❶Bob sends an email message to both Alice and Claire. Stream splits this message into two messages, one scheduled for delivery to each recipient. ❷Stream signs a copy of Claire's key and attaches that key to the message that Stream sends to Alice. ❸Stream also signs a copy of Alice's key and attaches it to the copy of the message that is sent to Claire. ❹Now Alice and Claire can immediately exchange secure mail, as they have been given copies of each other's keys, and those keys have been certified by Bob.

1. Remove any mail headers from the message that have `X-Stream` prefixes.
2. Remove the **mandatory encryption character** from the beginning of the Subject: line, if it is present.
3. Determine if an encryption key is present in the mail header.
   (a) If so, the key is added to the user's key database.
   (b) If this is a new key K_E for an existing email address E in the database, the user is notified of this fact. (Stream's method of communicating with the user is to send the user additional email messages.)
4. If the message is sealed with encryption:
   (a) Stream unseals the message.
   (b) Unencapsulate the encapsulated mail headers.
   (c) If key fingerprints were present, Stream verifies that the fingerprints on the encapsulated message headers match those for the copies of the keys in the key database.
   (d) If the fingerprints do not match, a warning is sent to the recipient.

(e) Insert the **mandatory encryption character** at the beginning of the Subject: line.

5. If the message is digitally signed:
   (a) If a key is on-file, verify the signature.
   (b) If the signature verifies, insert an X-STREAM mail header indicating this fact. (This allows sophisticated users to look at headers to verify signatures, should they choose to do so.)
   (c) If the signature doesn't verify, modify the Subject: line to indicate this fact. (For example, by adding the words "(BROKEN SIGNATURE") to the end of the Subject: line.)

In the above description, the phrase "beginning of the Subject: line" means the text that follows the colon and following space, ignoring any number of repeating "Re:" sequences.

Stream was developed in the fall of 2002 and used successfully by the author on MacOS, FreeBSD, and Windows. A paper on the technology was presented at the 2003 National Conference on Digital Government Research.[Gar03b]

D.2.3 Stream evaluation

Stream was intended to be a workbench for refining the technique of placing hidden signatures and keys in e-mail messages; to demonstrate the viability of a transparent encryption property; and to evangelize the philosophy that cryptography with weak email-based authentication was better than email without cryptographic protections at all.

Although Stream was reasonably successful in each of these goals, the software failed to achieve any adoption. The reason was not that nobody wished to download and install the program. Instead, the reason was that people didn't wish to go through the trouble of downloading software that implemented a cryptographic email protocol that wasn't compatible with any other system that was currently deployed. This proved to be a fairly dramatic lesson and it guided the design of the CoPilot system.

D.3 CoPilot: A Proxy or Plug-In that Implements KCM

CoPilot is the second proxy designed to implement the philosophy presented earlier in this appendix. CoPilot builds on the experience of the Stream project with the primary realization that it is better to leverage the existing email security technology that has been deployed over the past decade, rather than try to deploy a technology that implements a fundamental new standard.

Although there are many acknowledged problems with the S/MIME mail security standard—and even more with today's S/MIME implementations—it is the standard that has been deployed. The philosophy of CoPilot is that it is better to use the standards that are deployed, rather than waiting for better ones to come along.

The primary problem in automating S/MIME is that today's S/MIME agents generate annoying warning messages when they encounter email messages that are signed using Digital IDs that were not issued by trusted CAs. CoPilot proposes two approaches to this solution:

- CoPilot could be distributed with an agent that can perform the necessary challenge-response process with the Thawte web site and obtain a Thawte FreeMail certificate. At the present time, the only information that Thawte appears to validate for obtaining these certificates is the user's email address. Since CoPilot would have access to that email address, the entire acquisition process could be automated.
- Alternatively, CoPilot could be distributed with both the private key and the public key of "the permissive CA"—that is, the CA that is willing to sign any digital certificate. (What makes the CA permissive is the fact that its private key has been compromised by being distributed with CoPilot.) Because CoPilot implements the TRACK RECIPIENTS pattern, it is able to differentiate between the S/MIME users who have installed the permissive CA and those who have not.

CoPilot was created to demonstrate the viability of Key Continuity Management. But this thesis does not argue that Apple, Microsoft, and their customers should abandon today's Certificate Authority-based solutions. Instead, the argument is that today's products need to more sensibly handle the case when signed email is received for which the Digital ID was not created by a recognized signing authority. By addressing this particular case in a manner that is consistent with the principles outlined in this thesis, the use of S/MIME can be dramatically increased.

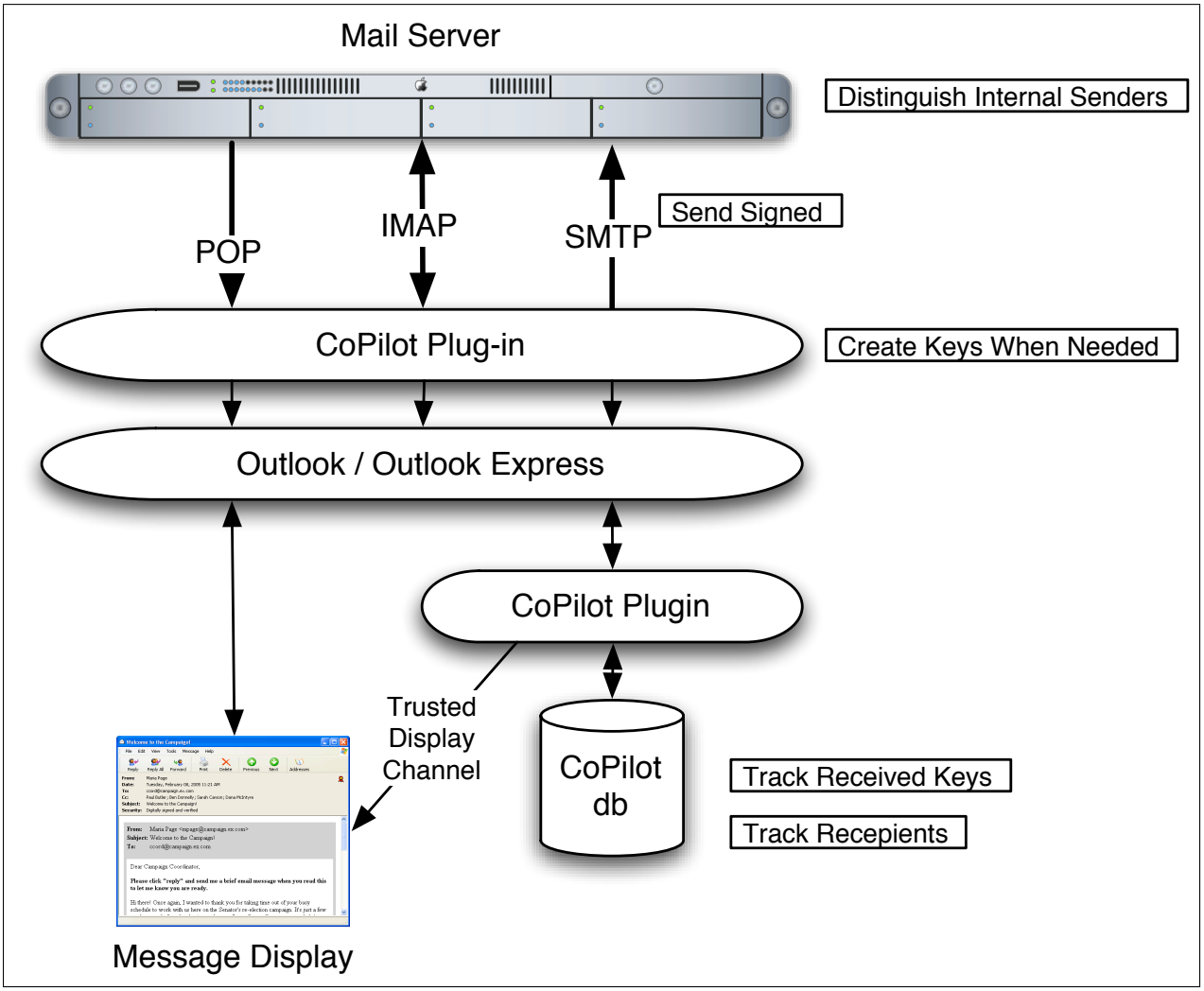

Figure D-6: The design of the CoPilot Plug-in for Outlook Express specifies that CoPilot monitors incoming mail (from POP or IMAP) and outgoing SMTP. As with Stream, CoPilot is able to unseal messages as they are downloaded and automatically sign and/or seal messages as they are sent. Unlike Stream, CoPilot uses a piece of reserved real estate in the Outlook Express window to convey information to the user. CoPilot also maintains a database of keys that have been received and the inferred capabilities of the key holders.

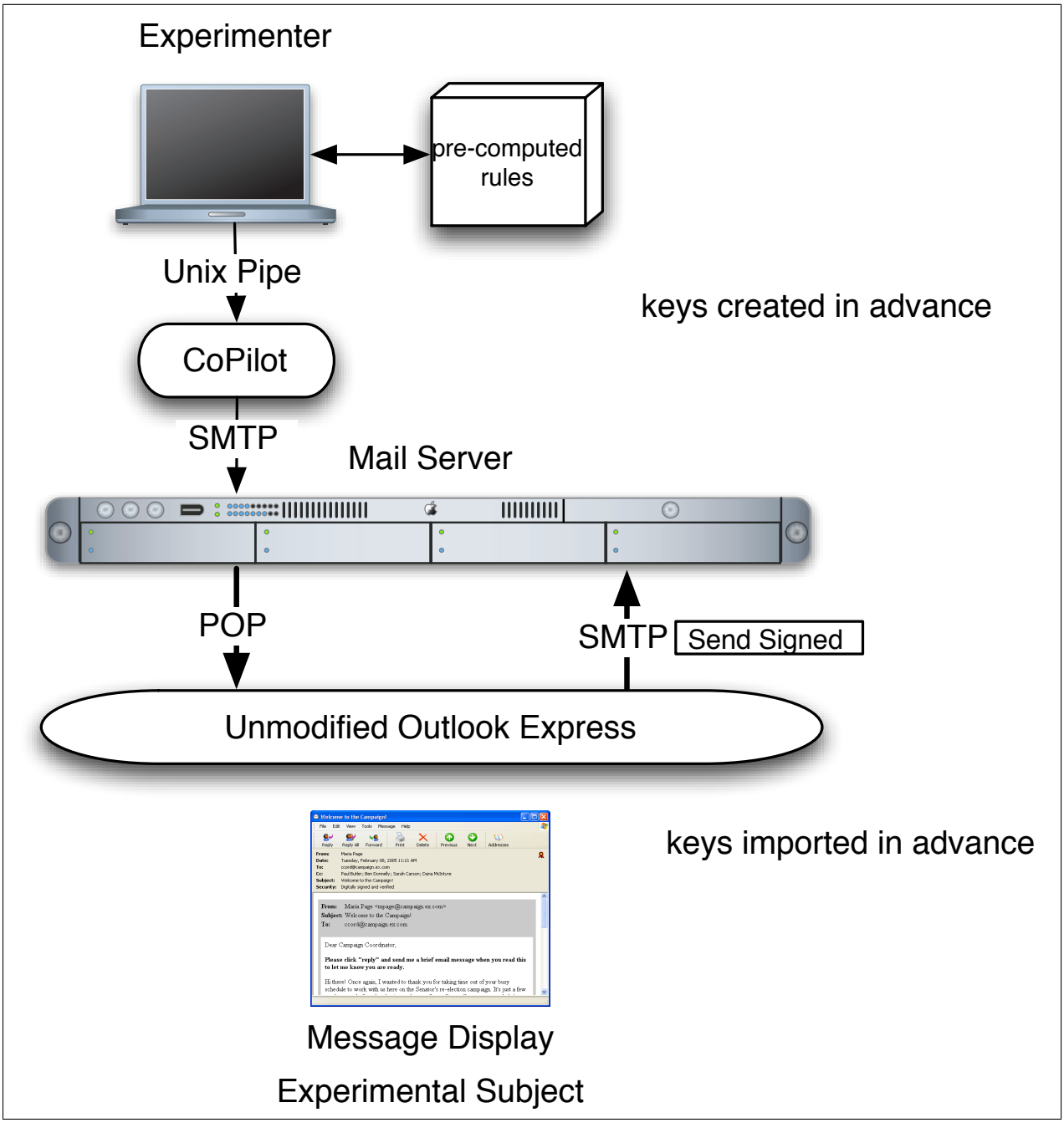

Figure D-7: The actual CoPilot system that was developed for the *Johnny 2* experiment. Keys were created in advance using OpenSSL and imported directly into the copy of Outlook Express running on the subject's workstation. The rules for each message were pre-computed in advance to infer whether messages should be displayed with the yellow, green, red, or gray borders. Messages were transmitted to CoPilot through a Unix Pipe. The CoPilot then opened an SMTP connection to the Unix server and sent the messages. These were then displayed using an unmodified copy of Outlook Express.