From Proxy Encryption Primitives to a Deployable Secure-Mailing-List Solution

Himanshu Khurana, Jin Heo, and Meenal Pant

National Center for Supercomputing Applications (NCSA),
University of Illinois, Urbana-Champaign
{hkhurana, jinheo, mpant}@ncsa.uiuc.edu

Abstract. Proxy encryption schemes transform cipher-text from one key to another without revealing the plain-text. Agents that execute such transformations are therefore minimally trusted in distributed systems leading to their usefulness in many applications. However, till date no application of proxy encryption has been deployed and used in practice. In this work we describe our efforts in developing a deployable secure mailing list solution based on proxy encryption techniques. Securing emails exchanged on mailing lists requires that confidentiality, integrity, and authentication of the emails be provided. This includes ensuring their confidentiality while in transit at the list server; a functionality that is uniquely supported by proxy encryption. In developing this solution we addressed the challenges of identifying requirements for deployability, defining a component architecture that maximizes the use of COTS components to help in deployment, developing the proxy encryption protocol to satisfy requirements and to fit within the component architecture, implementing and testing the solution, and packaging the release. As evidence of its deployability, the resulting secure mailing list solution is compatible with common email clients including Outlook, Thunderbird, Mac Mail, Emacs, and Mutt.

1 Introduction

Proxy encryption techniques enable the transformation of cipher-text from one public key to another without revealing the plain-text to the transforming agent. These techniques have been developed and studied for almost a decade since they were proposed by Mambo and Okamoto [25] and Blaze *et al.* [4]. Since then researchers have identified useful properties of proxy encryption schemes and developed several protocols that satisfy these properties [2], [18], [19], [33]. An important consequence of this cipher-text transformation capability of proxy encryption is that the transformation agent can participate in distributed protocols with minimal trust as it never gets access to the plain-text while still providing useful processing capabilities. Centered around this consequence of trust minimization several applications have been identified including simplification of key distribution [4], key escrow [18], file sharing [2], security in publish/subscribe systems [22], multicast encryption [10], and secure and certified email mailing lists [20], [21]. Furthermore, [2], [20] demonstrate practical feasibility of proxy encryption via prototype development and testing.

In this work we take on the task of building a deployable application that requires proxy encryption. Doing so for any new cryptographic technique in general, and proxy

P. Ning, S. Qing, and N. Li (Eds.): ICICS 2006, LNCS 4307, pp. 260–281, 2006.
© Springer-Verlag Berlin Heidelberg 2006

encryption in particular, requires that several challenges be addressed. First, there is the need to identify requirements of the security application geared towards deployability. This involves a requirement analysis and an in-depth study of both the application domain (e.g., file systems, mailing lists) and, if available, the lessons learned from successful security applications deployed in the domain. Second, the system design task must be undertaken to satisfy these requirements. This effort should attempt to maximize the use of COTS (Common-Off-The-Shelf) components in order to make deployment easier. Third, the proxy encryption based protocol must be designed and implemented using available cryptographic libraries. The protocol design must address the needs of the security application and it is often the case that protocol and system design steps are inter-linked and iterative. Fourth, the implemented components must be integrated with the application and then tested for performance and presence of vulnerabilities and errors. Finally, the integrated security application must be packaged and released along with an identified maintenance process. The security application that we develop by addressing these challenges is secure email mailing lists.

As more and more user communities are engaging in collaborative tasks, use of Email List Services (or simply *Mailing Lists* - MLs) is becoming common; i.e., emails exchanged with the help of a list server (examples of commonly used list server software include Mailman (`http://list.org`) and Majordomo (`http://www.greatcircle.com/majordomo/`)). Many tasks where MLs are used require exchange of private information. For example, a ML of security administrators that manage critical infrastructure would not want their emails publicly disclosed to prevent hackers from getting that information. Specific instances of this include the LHC Grid (`http://lcg.web.cern.ch/LCG/`) and TeraGrid (`http://security.teragrid.org/`) systems where the Incident Handling and Response policies recommend the use of encrypted and signed mailing lists. In general, use of encrypted and signed lists is recommended for incident response by IETF [7] and CERT [29]. Additional examples include a list of (1) healthcare and pharmaceutical researchers would not the want their emails publicly disclosed to protect patient privacy, and (2) corporate executives would not want their emails disclosed to protect proprietary information. For such lists cryptographic solutions are needed that provide adequate protection (i.e., confidentiality, integrity, and authentication) for the private content from threats at the client side, at the network paths where the emails are in transit, and at the server side where the emails are processed for distribution to the list. That is, there is a need to develop Secure Mailing Lists (SMLs). Threats to the server side are an important concern in practice and lack of good solutions today has forced users to develop their own clunky ones; e.g., distribution of passwords to list members out-of-band and requiring members to use password-based-encryption so that the list server does not have access to email plain-text[1]. It is in addressing this threat that proxy encryption provides a good solution by allowing the list server to transform email cipher-text between list members without gaining access to the plain-text.

By addressing the outlined challenges for mailing lists we developed PSELS – a Practical Secure Email List Service. Looking at the history of secure email development and deployment as well as the needs of mailing lists, we identified requirements in

[1] This particular approach has been adopted by several critical infrastructure security protection groups today.

the categories of security properties (i.e., confidentiality, integrity, and authentication), infrastructure compatibility, key management and performance. A primary requirement here is the minimization of trust in the list server. We then designed the architecture to include several COTS components that minimize development effort and maximize ease of deployment. In particular, we were able to use the OpenPGP message format [8] and standard GnuPG plugins at the client side to eliminate the need for developing email client-specific plugins. We then developed the PSELS protocol to satisfy the identified requirements and to fit with the system architecture and design. In particular, PSELS uses proxy encryption to minimize trust in the list server. This proxy encryption protocol is a modified version of that proposed in [20], however, unlike [20] it focuses on deployment and practical use. We then implemented the protocol and the system using the Mailman list server, GnuPG and BouncyCastle cryptographic libraries, and standard GnuPG plugins and APIs. We then tested our implementation in a test-bed environment for functionality, email client compatibility, and performance. Our results show the viability of PSELS in enterprise settings, compatibility with Microsoft Outlook, Emacs, Mac Mail, Mutt and Thunderbird, and satisfactory performance that scales to support enterprise mail servers that process hundreds of thousands of emails per day.

An initial version of the software has been packaged and released for community evaluation and is available at `http://sels.ncsa.uiuc.edu`. We plan to support the release in terms of software patching and update as well as enhancing the software with additional features.

The rest of this paper is organized as follows. In Section 2 we identify the requirements. In Section 3 we present the PSELS component architecture. In Section 4 we present the PSELS protocol. In Section 5 we discuss the implementation and testing efforts. In Section 6 we analyze the security of our design, protocol, and implementation. In Section 7 we discuss related work and conclude in Section 8.

2 Requirements

In this relatively new area of Secure Mailing Lists (SMLs) there is both a need and an opportunity to define a set of technical requirements such that the resulting tools and solutions that satisfy these requirements have a high likelihood of being deployed and used in practice. Fortunately, this area can benefit from the long history of solutions for secure two-party email exchange (or simply, secure email). Though secure email is not used nearly as commonly as the security research community would like, availability of inexpensive tools and solutions based on the S/MIME [26] and OpenPGP [8] standards bring us closer to this vision of ubiquitous secure email use with every passing year. We identify three important lessons for SMLs from the history of secure email (i.e., history of standards and tools such as S/MIME and OpenPGP). First, a secure email solution must provide the necessary security properties, namely, confidentiality, integrity, and authentication. Second, a secure email solution will be adopted by users only if it comes with support for easily obtaining, trusting, and managing public and private keys. Third, a secure email solution is deployable only if it is compatible with existing email infrastructure and if its hardware, software and administrative costs are reasonable. These lessons form the basis of our design and implementation efforts geared

towards deployability. We now define the various entities in SMLs and the technical requirements for PSELS.

2.1 SML Entities

- *List Moderator (LM). LM* is a user (or process) that creates a list to be maintained at the list server, authenticates users, and helps them subscribe to and unsubscribe from the list.
- *List Server (LS). LS* creates lists, maintains membership information (e-mail addresses and key material), adds and removes subscribers based on information received from *LM*, and forwards e-mails sent by a valid list subscriber to all current subscribers of that list.
- *Users/Subscribers.* Users subscribe to lists by sending join requests to *LM*, and send emails to the list with the help of *LS*.

2.2 Technical Requirements

Security Properties. A SML solution must provide *confidentiality*, *integrity*, and *authentication* for all emails exchanged on the list. Confidentiality of emails means that only authorized users (i.e. subscribers of the list) should be able to access the plain-text contents. Note that this definition excludes the list server from being able to read emails as it is not a valid subscriber. Example scenarios where the list server is not trusted to have access to cleartext contents include: (1) when protecting a distributed critical infrastructure the system administrators may not trust the list server as it may be located in a part of the network where most of the administrators have no control but, at the same time, its compromise will affect the security of their own networks, and (2) in military settings, the list server administrator may have a lower security clearance than the list subscribers and, therefore, should preferably not have access to the cleartext contents. In addition, this requirement also protects email content from an adversary that compromises the list server. Arguably, an adversary can more easily compromise a list subscriber to get access to email contents; however, if the email contents are available at the list server then its compromise would allow the adversary access to all messages on all lists managed by the server. Integrity of emails ensures that they cannot be modified in transit without such modifications being detected. Authentication of emails means that recipients can verify the identity of the sender.

Conceptually, by requiring a list server that provides message processing and forwarding functions but does not have access to message contents, we essentially deem it to be a semi-trusted third party. Such an approach minimizes trust liabilities in essential services for multi-party protocols and is often used; e.g., in fair exchange of digital goods [15].

Infrastructure Compatibility. In order to enable the deployment and use of SMLs, the protocols and tools must be compatible with existing email infrastructure. This includes the existing email servers, list servers, and email clients. All of this infrastructure will typically comply with a subset of existing email standards (`http://www.imc.org/rfcs.html`). While it is challenging to support all possible infrastructure systems and configurations, the choice of supported ones greatly influences the spread of SML use.

Key Management. To adopt SML solutions, users need to obtain, trust and manage cryptographic keys. These key management functions should either be built into the SML solutions or must be accessible through easily available, inexpensive means. In contrast, consider the fact that subscribers of a ML can come from a large number of domains. If, in order to use an SML solution, they need to obtain and trust CA certificates of all the domains then the key management becomes too complex.

Performance. Busy mail servers in medium-sized organizations today process more than 50,000 emails a day on average. Since SMLs will depend on existing mail servers for their delivery, it is essential that their performance not overburden them. Requiring additional servers for SMLs would significantly increase their infrastructure deployment costs.

3 Component Architecture

In Figure 1 we illustrate the component architecture of PSELS. We identify components on the server, the list moderator, and subscribers. Where appropriate we identify

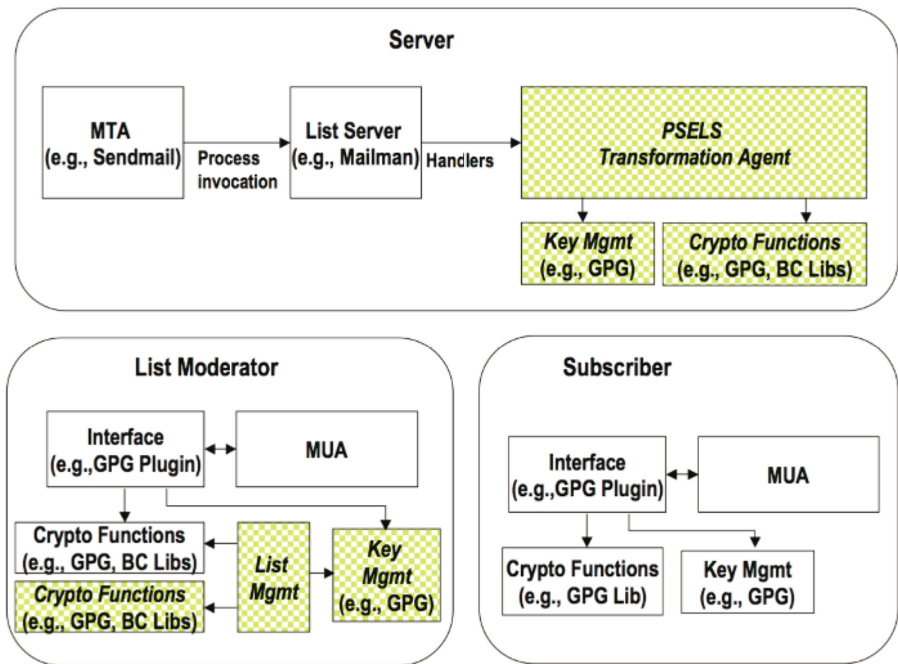

Fig. 1. Component Architecture

examples of COTS tools that can be used for the component functionality as is or with suitable modifications. This architecture is based on the requirements identified earlier, namely, security properties, infrastructure compatibility, key management, and performance.

Server Components

We envisioned the use of public-key based proxy encryption schemes to address the confidentiality requirements and the PSELS Transformation Agent provides that functionality. The Agent needs to include Crypto Functions that execute proxy encryptions as well as Key Management functions that provide generation, storage, and use of cryptographic keying material. Standard libraries such as GPG and BC can be used to develop the necessary crypto functions while for key management the use of COTS tools such as GPG is appropriate. For infrastructure compatibility we envisioned the use of COTS list servers and mail servers with a commonly used example of each being Mailman and Sendmail respectively. It is at the server side that performance is a major concern (as opposed to list moderator or subscriber side) because of a potentially large number of proxy encryption operations that may need to be executed. We study this performance via extensive experimentation but as a design parameter we envisioned the use of appropriate message passing interfaces (e.g., Mailman handlers) to connect the transformation agent with the list server so that, if needed, the agent can run on a separate machine or on multiple machines.

Subscriber Components

We observed that the development of new components on the client-side will greatly impact the infrastructure compatibility requirement because (1) users have preference for email clients (or, MUAs- Mail User Agents) so the new components must be compatible with their existing MUAs and (2) users are reluctant to install new software as well as updates to the software. At the same time, subscribers will need Key management and Crypto Functions to use PSELS; e.g., to store encryption and signature verification keys and to encrypt, decrypt, sign, and verify emails. To address this requirement we envisioned the use of a COTS Interface component that (1) provides the necessary key management and crypto functions for commonly used email clients and (2) complies with standardized messaging formats to ensure interoperability. Two examples of such a component are S/MIME tools for the S/MIME messaging format [26] and GPG tools for the OpenPGP messaging format [8]. These components are available today either as easy-to-install plugins or provided natively for many MUAs. S/MIME standards and tools are RSA based and since our proxy encryption protocol is El Gamal based these standards and tools cannot be used. Therefore, we chose to go with OpenPGP message formats and GPG tools. Furthermore, we argue in later sections that developing RSA based proxy encryption schemes for PSELS is especially challenging because it can result in sharing of the RSA modulus, which is considered insecure. As a result of the design choice of using a COTS Interface component, we require no development on the client side and yet ensure compatibility with a large number of commonly used email clients.

List Moderator Components

LM helps in creating lists and subscribing users with capabilities provided by the List Management and Key Management components. Since user subscription will require generation and distribution of proxy keys, *LM* will need to access crypto functions that are developed using appropriate crypto libraries (such as GPG and BC). Other *LM* components include a MUA and an Interface that provides access to basic crypto and key management functions for which tools such as GPG will suffice.

4 PSELS Protocol

For developing a protocol for PSELS that satisfies the outlined requirements and the component architecture, we evaluated existing protocols for multi-recipient email encryption and encrypted mailing lists [20], [28], and [32]. Of these SELS [20] offered a good starting point. However, the protocol required several modifications for satisfying the infrastructure compatibility and key management requirements. For example, SELS requires modifications to messaging formats and special processing capabilities on the client-side making it impossible to satisfy our client-side infrastructure compatibility requirements as well as the client-side component architecture. In this section, we present the PSELS protocol, which is a modified version of SELS. After we present the PSELS protocol we discuss the specific differences and improvements over SELS.

4.1 Proxy Encryption Scheme

We present the ElGamal public-key encryption scheme $\mathcal{E}_{eg}$ and the PSELS public-key encryption scheme, $\mathcal{E}$, which is based on the discrete log problem like El Gamal. $\mathcal{E}$ specifies an encryption transformation function that enables *LS* to transform an e-mail message encrypted with the list public-key into messages encrypted with the receivers' public keys.

Let $\mathcal{E}_{eg} = (Gen, Enc, Dec)$ be the notation for standard ElGamal encryption [16]. *Gen* is the key generating function. Hence $Gen(1^k)$ outputs parameters (g, p, q, a, g^a) where g, p and q are group parameters, (p being k bits), a is the private key, and $y = g^a \bmod p$ is the public key. The *Enc* algorithm is the standard El Gamal encryption algorithm and is defined as $e = (mg^{ar} \bmod p,\ g^r \bmod p)$, where r is chosen at random from Z_q. To denote the action of encrypting message m with public key y, we write $Enc_{PK_y}(m)$. *Dec* is the standard El Gamal decryption algorithm and requires dividing mg^{ar} (obtained from e) by $(g^r)^a \bmod p$. We assume all arithmetic to be *modulo p* unless stated otherwise.

We denote the PSELS encryption scheme by $\mathcal{E} = (IGen,\ UGen, AEnc, ADec, \Gamma)$. Here *IGen* is a distributed protocol executed by *LM* and *LS* to generate group parameters g, p and q, private decryption keys K_{LM} and K_{LS} and public encryption keys $PK_{LM} = g^{K_{LM}}$, $PK_{LS} = g^{K_{LS}}$, and $PK_{LK} = g^{K_{LM}}.g^{K_{LS}}$. K_{LM} is simply a random number in Z_q chosen by *LM*, and K_{LS} is a random number chosen by *LS*. *UGen* is a distributed protocol executed by user U_i, *LM*, and *LS* to generate private keys for U_i and *LS*. $UGen(K_{LM}, K_{LS})$ outputs private keys K_{U_i}, K'_{U_i} and the public keys $PK_{U_i} = g^{K_{U_i}}$, $PK'_{U_i} = g^{K'_{U_i}}$. K'_{U_i} is called user U_i's *proxy key* and is held by *LS*. Furthermore, it is guaranteed

that $K_{U_i} + K'_{U_i} = K_{LM} + K_{LS} \bmod q$. This protocol requires LM, and LS to generate random numbers and add/subtract them from K_{LM} and K_{LS}. $AEnc$ and $ADec$ are identical to Enc and Dec defined above for $\mathcal{E}_{eg}$. $\Gamma_{K'_{U_i}}$ is a transformation function that uses user U_i's proxy key to transform messages encrypted with PK_{LK} into messages encrypted with user U_i's public key. It takes as input an encrypted message of the form $(g^{rK_{LK}}M, g^r)$ and outputs $(g^{rK_{U_i}}M, g^r) = ((g^{rK'_{U_i}})^{(-1)} g^{rK_{LK}}M, g^r)$. Once $UGen$ has been executed for users U_i and U_j, then sending a message between the users requires user U_i calling $AEnc_{PK_{LK}}$, LS calling $\Gamma_{K'_{U_j}}$, and user U_j calling $ADec_{K_{U_j}}$. The encryption scheme $\mathcal{E}$ is correct because $ADec_{K_{U_j}}(\Gamma_{K'_{U_j}}(AEnc_{PK_{LK}}(m))) = m$. In practice, hybrid encryption is used for efficiency as illustrated in Figure 2.

The encryption scheme $\mathcal{E}$ is secure if it retains the same level of security as the standard El Gamal scheme against all adversaries $\mathcal{A}$, and if LS cannot distinguish between encryptions of two messages even with access to multiple proxy keys. The formal theorem and proof of $\mathcal{E}$'s security is provided in [21].

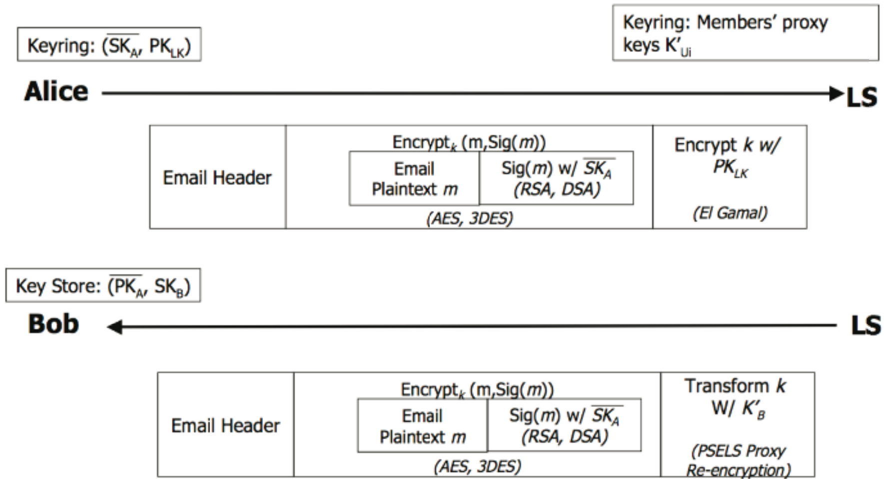

Fig. 2. Sending Emails in PSELS

4.2 Protocol Steps

We now present the protocol steps for creating a list, subscribing users, sending e-mails, and unsubscribing users. The step for sending emails is illustrated in Figure 2, which follows standard secure email messaging formats of S/MIME and OpenPGP. In the protocol description, Enc_{PK_i}(m) denotes the encryption of message m with public-key PK_i, and $Sig_{\overline{K_i}}$(m) denotes the message m along with its signature with private key $\overline{K_i}$. We distinguish between encryption/decryption keys and signature/verification keys by placing a *bar* on top of signature/verification keys; i.e., ($\overline{K_i}$, $\overline{PK_i}$) represents a signature and verification key pair and (K_i, PK_i) represents a decryption and encryption

key pair. As illustrated in the figure, hybrid encryption is used in standard email message formats as using bulk encryption with public key technologies is expensive. However, to simplify the protocol description we do not include details of the hybrid encryption. Therefore, Enc_{PK_i}(m) is actually $\{Enc_k$(m), $Enc_{PK_i}(k)\}$ where k is a symmetric key and Enc_k is a symmetric encryption function such as AES. That is, for simplicity of representation we just use Enc_{PK_i}(m).

Creating a List. To create a new list L, LM and LS execute the following steps:

1. LM begins the execution of $IGen$ and generates parameters $(g, p, q, K_{LM}, g^{K_{LM}})$, and associates the key pair (K_{LM}, PK_{LM}) with the list.
2. LM then sends LS a message with the values g, p, and q, and the new list ID L. Formally, $LM \longrightarrow LS$: $Sig_{\overline{K_{LM}}}$("Create" List L, $g, p, q)$.
3. LS then completes the execution of $IGen$ by choosing a new private key K_{LS} using the group parameters sent by LM, computing public key $PK_{LS} = g^{K_{LS}}$ and associating the key pair with the list. LS then sends the computed public key back to LM. Formally, $LS \longrightarrow LM$: $Sig_{\overline{K_{LS}}}(L, PK_{LS})$.

Both LM and LS implicitly agree that the sum $K_{LK} = K_{LM} + K_{LS}$ (mod q) is the *list key* but neither knows its value since neither knows the other's private key. The list is now ready for subscription.

Subscribing and Unsubscribing Users. To subscribe user U_i to list L, U_i, LM and LS execute the following steps. Here we distinguish between encryption keys generated by PSELS and those that users get from an external PKI (e.g., GPG keys that they generate themselves) with a superscript Ext on the external PKI keys; i.e., (K_i^{Ext}, PK_i^{Ext}) is an external decryption/encryption key pair.

1. U_i sends a signed "join" request to LM. Formally, $U_i \longrightarrow LM$: $Sig_{\overline{K_{U_i}}}$("Join" List L, $PK_{U_i}^{Ext}$), $\overline{PK_{U_i}}$.
2. LM authenticates U_i and generates a random value r and then computes the user's private key $K_{U_i} = K_{LM} + r$ mod q. LM then sends this key to U_i encrypted with his external encryption key $PK_{U_i}^{Ext}$, along with the list encryption key PK_{LK}. Users then decrypt this message and store their decryption/encryption key pair. Formally, $LM \longrightarrow U_i$: $Enc_{PK_{U_i}^{Ext}}$ $(Sig_{\overline{K_{LM}}}(PK_{U_i}, K_{U_i}, PK_{LK}))$.
3. LM sends the value r to LS. Formally, $LM \longrightarrow LS$: $Enc_{PK_{LS}}(Sig_{\overline{K_{LM}}}$("Join" L, U_i, $PK_{U_i}, r))$.
4. LS obtains r from LM, and computes and stores the proxy key $K'_{U_i} = K_{LS}$ - r mod q.

To unsubscribe from list L, user U_i, LM, and LS execute the following steps:

1. U_i sends a signed "leave" request to LM. Formally, $U_i \longrightarrow LM$: $Sig_{\overline{K_{U_i}}}$("Leave" List L).
2. LM authenticates U_i and, deletes U_i's signature verification and external encryption keys from its key ring, and sends a request to LS to delete the user's proxy keys. Formally, $LM \longrightarrow LS$: $Sig_{\overline{K_{LM}}}$("Leave" L, U_i, PK_{U_i}).
3. LS verifies LM's signature on the message and deletes users U_i's proxy key K'_{U_i} from its keyring.

Sending E-mails. To send an e-mail to the list L, sender U_i, LS, and all receivers U_b ($b \neq i$) execute the following steps:

1. U_i first signs the e-mail m with his private key $\overline{K_{U_i}}$ and then encrypts it with the list public key PK_{LK}. U_i then sends to LS the encrypted e-mail message: Formally, $U_i \longrightarrow LS : (X = Enc_{PK_{LK}}\ (Sig_{\overline{K_{U_i}}}(m))$
2. To forward the e-mail to every user U_b who is subscribed to list L, LS computes and sends to U_b a transformation of X with U_b's proxy key K'_{U_b}. Formally, $LS \longrightarrow U_b$: $(Y_b = \Gamma_{K'_{U_b}}\ (X))$.
3. Each recipient decrypts the e-mail message Y_b from LS using his private key K_{U_b} with algorithm $ADec$. The receiver can then verify the sender's signature on the decrypted e-mail.

4.3 Differences with the SELS protocol [20]

The PSELS protocol specified above differs from the one presented in [20] in three ways and all of these changes were needed to enable satisfaction of the infrastructure compatibility and key management requirements. First, the proxy encryption scheme is modified to encrypt outgoing emails with the list public key, PK_{LK}, as opposed to the sender's public-key. This simplifies the proxy re-encryption step and is also more aligned with the manner in which email encryption is used today with standard crypto interface components (like GPG); i.e., associating the list encryption key with the list email address. Second, we simplify user subscription by allowing *LM* to compute and send users' decryption keys. In SELS a distributed protocol is used so that *LM* does not have access to users' decryption keys. We argue that this is not needed in practice because *LM* anyway has access to all emails exchanged on the list and our simplification allows us to satisfy the infrastructure compatibility requirement by not developing any new software on the client side. Third, as opposed to SELS, we do not use keyed MACs on email messages for authentication at *LS*. Such MACing capabilities will require modifications on the client side and, therefore, were excluded from PSELS.

5 Implementation, Testing, and Experiments

5.1 Component Design and Development

Server Components

On the server side we were able to use COTS components for the Mail Server and the List Server and then had to develop components for the PSELS Transformation Agent along with the needed Crypto Functions and Key Management components. For the List Server we chose the open source Mailman tool, which has extensible features and is widely deployed today. Mailman works with most SMTP servers and we chose Send-Mail in keeping with our open source approach. To connect the Transformation Agent with Mailman we used Mailman handlers to allow for easy installation of developed server components on new and existing Mailman setups. The handlers also allow for the Transformation Agent to run on a different machine if needed; e.g., for reasons of performance or security. The Transformation Agent was developed in C, Python and

Java leveraging the GPG and BC crypto libraries for proxy key generation and encryption functions, the GPG key management component for key storage and access, and the Python GnuPGInterface (`http://py-gnupg.sourceforge.net/`) for the interface between Mailman and GPG functions. Since the PSELS proxy encryption scheme is based on El Gamal we decided to go with GPG tools and the standardized OpenPGP message format. Both GPG and BC crypto libraries are open source and provide suitable capabilities to implement the proxy key generation and encryption functions while the GPG key management functions provide suitable capabilities for storing and accessing proxy keys.

For each of the four protocol steps, namely, list creation, list subscription, email sending, and list unsubscription, we define a unique Mailman handler that leads to an execution of that step at *LS*. Most of the operations in executing these steps involve the use of standard GPG functions with the following two exceptions: creating a proxy key on user join and proxy transformation on email forwarding. To generate a proxy key on user join the Agent uses a specialized crypto function developed using BC to extract the random value r from the *LM*'s message and the private key, K_{LS}, and compute the user's proxy key as specified in the protocol.

To send an email to list L, a user first signs the email and then encrypts it with the list public key all using any email client that works with a GPG plugin. The resulting email message is a standard OpenPGP message, which consists of one or more packets. Each encrypted message has a Public-Key Encrypted Session Key Packet followed by a Symmetrically Encrypted Data Packet. The Public-Key Encrypted Session Key packet contains the randomly generated session key (symmetric key) used to encrypt a message and key IDs of public keys used to encrypt the session key. The Symmetrically Encrypted Data Packet contains email contents encrypted with the session key. The data can further be compressed or signed. A handler at *LS*, extracts the Public-Key Encrypted Session Key Packet and uses specialized crypto functions developed using GPG libraries to parse the incoming GPG messages into packets. After correctly locating Public-Key Encrypted Session Key Packet, the functions apply proxy transformation to this packet. This process is repeated for every recipient and the resulting messages are passed to MTA for delivery to list members.

Subscriber Components

On the Subscriber side we were able to use COTS tools for the MUA, Key Management, Crypto Functions, and Interface components. In keeping with the approach discussed above, we use GPG tools and ensure compatibility with all MUA's for which GPG plugins are available. Among others this includes popular MUAs like Outlook, Thunderbird, Eudora, Emacs, Mac Mail, and Mutt. GPG plugins provide all the necessary key management and crypto functions needed by subscribers to use PSELS.

For subscribing to lists, users send signed GPG messages to *LM* and receive back a set of necessary GPG keys and certificates. That is, users receive the list encryption key, their individual decryption keys (encrypted with their external GPG public keys), *LM*'s signature verification key, and their own signature verification key signed by *LM*. Depending on the features of their GPG plugin they either automatically or manually add all these keys and certificates to their GPG keyring. To send an email to the list,

subscribers sign the email with their GPG signature keys and encrypt it with the list public key (which is again a standard GPG key in their keyring). Since the list encryption key is associated it with the list email address, the GPG plugin automatically finds and encrypts the email with this key. On receiving an email on the list, subscribers simply use their GPG plugins to automatically find the appropriate keys and decrypt and verify the message.

List Moderator Components

For the List Moderator we were able to use COTS tools for the MUA, Key Management, Crypto Function and Interface components, and had to develop the List Management component. Similar to the reasons discussed above, we chose to use GPG tools for the Interface component along with the provided crypto and key management functions. The List Management component was developed in Python and Java using the GPG and BC libraries. For storing and accessing keys, the standard GPG key management functions were used.

To create lists, the List Management component uses standard GPG functions (via the command-line interface) to generate an ElGamal key pair and then sends the public key to *LS* via email in a GPG signed message. When *LM* receives *LS*'s public key back via email, a special crypto function developed using BC computes the list public key, PK_{LK}, by multiplying *LM*'s and *LS*'s public keys.

To subscribe users, the List Management component executes the following steps after receiving the user's subscription request: (1) verify the user's signature on the request, (2) use a special crypto function developed using BC to generate a private key for the user as specified in the protocol, and (3) send signed and encrypted emails to the user and to *LS* with the appropriate keying material. Once the emails have been sent, *LM* deletes the user's decryption keys for security reasons.

Additional Functions: Trust Management and Key Update

In addition to implementing the protocol steps as described above, PSELS components also implement two additional features: Trust Management and Key Update. Trust management involves distribution of signature verification keys to allow subscribers to verify signatures on emails sent to the list. Key Update involves the distribution of an updated set of decryption and proxy keys.

Subscribers in a mailing list may belong to different organizations, which make it difficult for them to distribute and trust their signature verification keys. In PSELS we address this problem by using *LM* as the trust anchor for the lists. Since *LM* is trusted to distribute decryption keys and to help in generation of proxy keys at *LS*, it is an appropriate entity to enable the establishment of trust in subscribers' signature verification keys. To do so, *LM* signs every subscriber's signature verification key in list subscription step and stores this signed key in the list key ring (as noted above, *LM* also sends the signed key back to the subscriber). Since subscribers already have *LM*'s signature key in their key rings and trust this key, they can place transitive trust in other subscribers' signature verification keys. Furthermore, *LM* can also distribute the signed verification keys to subscribers on request by extracting them from the list key ring and sending it to the subscribers as an email attachment.

The PSELS solution works on the assumption that an adversary cannot get access to the decryption key K_{LK} by simultaneously compromising either *LM* and *LS* (as $K_{LK} = K_{LM} + K_{LS} \ mod \ q$) or any user U_i and *LS* (as $K_{LK} = K_{U_i} + K'_{U_i} \ mod \ q$). Though unlikely, such compromise is possible. To address this concern, the protocol includes a key update step that allows *LM* to easily initiate and complete the process of changing all list encryption/decryption keys. This key update step would be executed either on a periodic basis for proactive security or when a compromise is detected. Key update can also be used to change the *LM* for a given list. To initiate a key update, *LM* sends a "Update *L*" message to *LS*, which includes a new key PK_{LM}. On receiving this message, *LS* (via a handler 'UpdateL') computes a new key pair (K_{LS}, PK_{LS}) and a new list key PK_{LK}. *LS* also deletes all proxy keys for this list. *LM* then generates a new encryption/decryption key pair for each subscriber and sends it encrypted with the subscriber's GPG encryption key stored in *LM*'s key ring (this is the external encryption key referred to as $PK_{U_i}^{Ext}$ in the protocol described in Section 4.2.2). On receiving the message from *LM*, a subscriber simply adds the new certificates to his key ring and associates them with the list. *LM* also sends a "U_i JOIN LIST *L*" message to *LS* for each subscriber, which is processed as usual. The list has now been re-keyed.

5.2 Testing

We have tested the PSELS implementation for correct functionality and for compatibility with multiple platforms and MUAs. To do so, we have set up a test-bed that includes a linux Debian server (which includes Sendmail and Mailman) and a set of client machines each of which have a different platform (including Windows, Mac, and different flavors of *-nix). Since Mailman only works on *-nix platforms, which are similar in nature, we felt that for initial testing on the server side using any one *-nix platform is sufficient. (In the future, we will test other *-nix server platforms as well.) For the List Moderator and Subscriber side, however, we needed to test compatibility with a variety of platforms.

Functional Testing. We wrote scripts that automate all of the protocol steps, namely, list creation, list subscription, email sending, and list unsubscription. We ran these scripts on the test-bed to verify that the implementation works correctly. The scripts use both correct and incorrect inputs and check whether the results are correspondingly correct or incorrect. This process helped us identify several useful checks that were then included in the implementation. For example, the scripts used incorrect list public keys for encrypting emails. Initially, this resulted in undecipherable messages being delivered to the subscribers. To address this we included a check at *LS* to ensure that only emails encrypted with the correct public key are delivered to subscribers.

Compatibility Testing. We've tested the combination of the COTS and developed List Moderator components on three platforms successfully: *-nix (in particular, Debian, Fedora, and Red Hat Linux), Mac, and Windows (XP). In each case a configuration file is generated to allow the developed components access to installed GPG tools.

For the client side, we've successfully tested PSELS with five commonly used email clients each of which has its own GPG plugin (`http://www.gnupg.org/(en)/related_software/frontends.html`): (1) Thunderbird with Enigmail, (2) Microsoft

Outloook with gpg4win, (3) Emacs with Mailcrypt, (4) Mutt with built-in GPG support, and (5) Mac Mail with MacGPG. Testing efforts resulted in a few changes at *LS* to accommodate slight differences in email encryption between the various GPG plugins (e.g., the gpg4win plugin adds an additional attachment to encrypted html messages). We've documented the steps necessary to ensure correct configuration and setup with each email client.

5.3 Experiments

In this section we evaluate the performance of the PSELS implementation. In most organizations the ML software is co-located with the MTA in the mail server. The main goal of the experiments is to observe how the addition of our security solution affects the overall performance of the mail server. To evaluate the performance of these solutions we use an insecure ML setup as a common baseline.

Experimental Setup

For all experiments shown in this paper, we run both Mailman and Sendmail on the same machine. The mail server machine we use for our experiments is equipped with two 3GHZ Dual Core Intel Xeon processors and 3GB RAM. The machine runs Debian Linux with kernel version 2.6.8 (compiled with SMP option turned on). We use version 2.1.5 of Mailman, which was the most recent version when we started this research. For the MTA, we use the Debian linux distribution of Sendmail version 8.13.4. For PSELS, we have developed Mailman handlers and crypto functions using GPG 1.4.2 to perform proxy transformations.

To gauge the overhead of PSELS we use *throughput* as our performance metric; i.e., the maximum number of emails per unit time that the mail server can process and deliver. We focus exclusively on MLs so we assume that the mail server does not process any two-party email exchange. Since we are only interested in the throughput of the mail server, we ignore networking delays by placing list subscribers on the same server machine. In our setup the emails for the subscribers are delivered to `/var/spool/mail/userid`. To estimate the throughput we measure the average delay for processing one email message sent from a list subscriber and delivered to all list subscribers. We then compute the normalized $\textit{throughput} = \frac{1}{\textit{delay}} * \textit{list size}$ where *list size* represents the number of subscribers in the list. Here *list size* is the normalization factor and is important for our experiments because PSELS executes cryptographic operations per email per recipient. For a more detailed analysis of the results we measure the delay in two cases: (1) for Mailman alone, and (2) for Mailman and Sendmail combined; i.e., from receiving the message at Sendmail, its processing at Mailman via handlers, to completing delivery for all subscribers. In the first case we use the Mailman log entries to measure the delay. In the second case we use the wall clock time when the email is sent by our test client as the start time and the log entry of Sendmail when it finishes delivering the email to all subscriber inboxes as the end time.

Measurements

In order to get an idea of how PSELS affects throughput we vary both the list size (number of subscribers) as well as the size of the email message. We use 10, 25, 50, 100, and 200 as the different list sizes and 1KB, 10KB, and 100KB as the different email sizes.

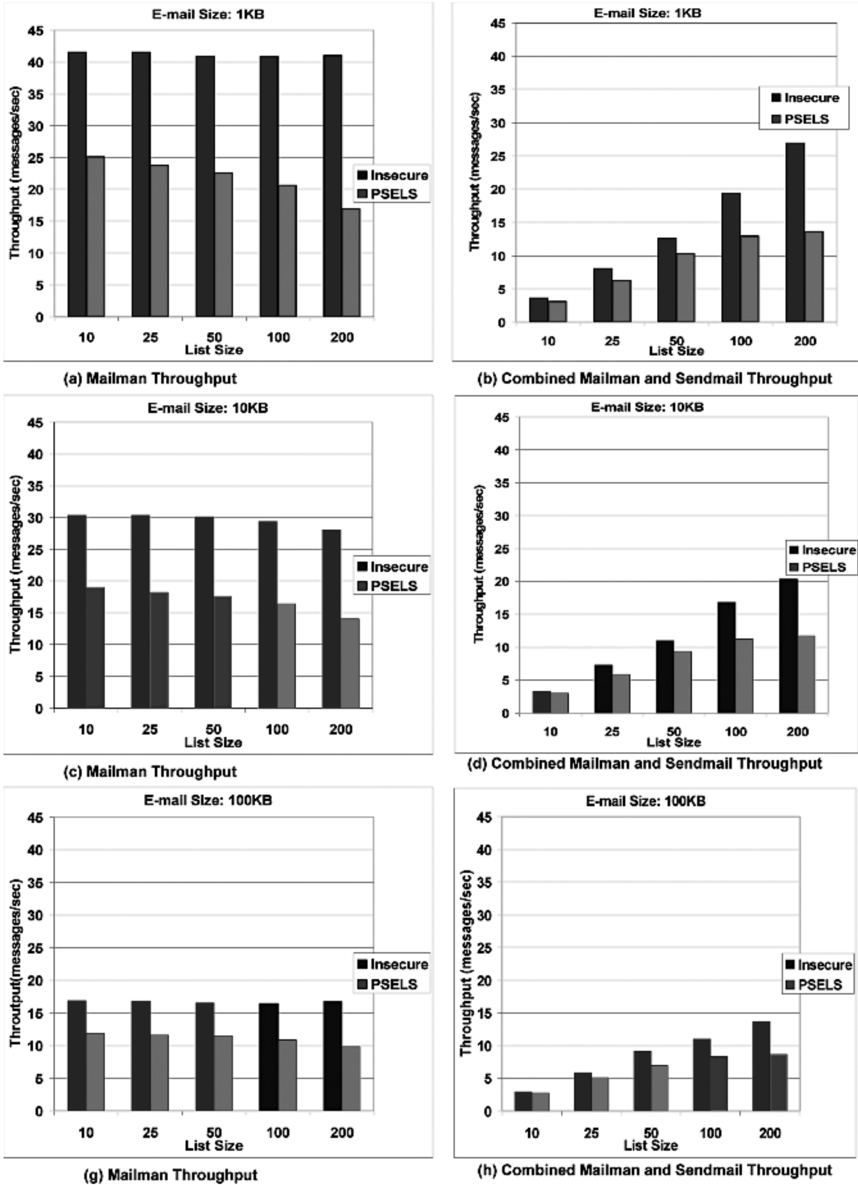

Fig. 3. Results

We limit the list size to 200 because we argue that any sensitive message that needs to be encrypted is unlikely to be sent to a large number of recipients. (If the list has only signed but otherwise cleartext contents then the list server need not do any additional work.) Since in MLs subscribers usually do not send large attachments and most of the

posted messages are text (including HTML, RTF formats), we believe that 100KB is a reasonable maximum size.

To run the experiments we first populate the lists by generating user keys (stored in GPG key rings) and subscribing users. The subscription process in PSELS also results in generation of proxy keys for *LS*, which are stored using a GPG key ring. We also pre-generate the signed and encrypted messages of the different sizes to be sent on the lists. We then run the experiments as follows. For each of the two setups (Insecure baseline and PSELS) we first fix the list size as well as the email size. We then execute a script at the sender to send 20 emails (one at a time waiting for complete processing and delivery) and measure the delays for Mailman as well as for Mailman and Sendmail combined for email processing and delivery. We average the result over these 20 runs. We then vary the email size and the list size and measure the delays similarly.

Figure 3 (a), (c), and (e) shows the measured throughput using Mailman, and Figure 3 (b), (d), and (f) shows the measured throughput using Mailman and Sendmail combined. In all of the figures, the x axis represents list size and the y axis shows the throughput in terms of messages per second (processed and delivered). Figures 3 (a), (c), and (e) show the expected result of the baseline case having the better performance with the throughput reaching up to 40 messages/sec regardless of list size. PSELS has lower performance that degrades as the list size increases. Figures 3 (b), (d), and (f) are similar in that the baseline case has better performance with one big difference being that here the throughput of the two setups increases with list size and varies greatly.

Analysis

In all experiments the baseline insecure setup show the better performance, as expected, since there are no cryptographic operations involved. Also, the performance of the base case is not affected by list size because besides delivery to individual subscriber inboxes there is no additional processing of an email message per recipient.

Mailman Throughput. We first discuss the throughput for Mailman, which is shown in figures 3 (a), (c), and (e). In PSELS, across all measurements, throughput decreases as the list size increases. This is because increase in list size leads to large key ring files and our measurements indicate that the overhead of searching and reading through the key ring files begins to dominate. A general observation for these set of experiments is that as the email message size increases, the throughput decreases. The difference is significant and owes to the overhead of managing larger sized buffers for the email messages.

Combined Mailman and Sendmail throughput. Figures 3 (b), (d), and (f) indicate that the throughput increases with list size and varies greatly. This is because Sendmail has a constant overhead in processing an incoming mail message – about 3 sec in our setup. The effect of this overhead reduces as the list size increases and, therefore, the throughput increases with list size. This is true for both setups and the performance of PSELS lags behind the Insecure case. Similar to the case above, as we increase the email message size the throughput decreases.

Average throughput. In the case of Mailman, PSELS shows a 42.2% average throughput degradation against the baseline. For combined Mailman and Sendmail throughput, PSELS shows a 28.8% average throughput degradation. Overall, we see that even the

worst throughput of 2.5 messages/sec for PSELS with list size 10 and message size 100KB corresponds to a throughput of more than 200,000 messages per day. Since most mail servers in small and medium-sized organizations do not typically process more than 100,000 messages per day (of which only a subset are ML messages) we conclude that adding security to MLs will not impose an undue overhead on the mail servers.

6 Security Analysis

After the functional, compatibility, and performance testing of the PSELS implementation, we analyzed the design and implementation and identified the following security concerns.

List Key Compromise

If an adversary compromises either LM and LS or any user U_i and LS then he can compute the list decryption key K_{LK}. This is because $K_{LK} = K_{LM} + K_{LS} \; mod \; q = K_{U_i} + K'_{U_i} \; mod \; q$. This would allow the adversary to read all emails encrypted with PK_{LK}. Furthermore, it would also allow him to compute every list subscribers decryption keys as $\forall_j \; K_{U_j} = K_{LK} - K'_{U_j} \; mod \; q$. This latter capability is a known property of proxy encryption schemes sometimes referred to as the "collusion" property [2]. The consequence of this attack is that recovery requires re-keying of the entire list rather than revoking one member. However, note that this would not allow the adversary to arbitrarily impersonate a list subscribers because all emails are signed with the subscribers' signature keys that are not compromised in this attack. In [2] they develop signcryption schemes that provide combined signing and proxy encryption capability and provide similar protection against impersonation by ensuring that compromise of decryption key does not imply compromise of signing capability.

To resolve this problem one can consider both a theoretical and a practical approach. Theoretically, the design of collusion-resistant proxy encryption schemes is an open problem. Designing such schemes to work with COTS components in a deployable architecture compounds this problem further. In practice, one can argue that simultaneous compromise of LS and LM (or U_i) is very unlikely. Furthermore, the provided key update functionality provides a mechanism to (1) prevent K_{LK} compromise by executing it as soon any one entity is compromised and (2) limit the adversary's access to email contents in case of successful K_{LK} compromise by executing it periodically to change list encryption keys. However, there are cases where this risk may be unacceptable. In such cases, additional security can be provided by splitting the list key, K_{LK}, three or more ways with the additional splits being hosted in different servers or by using threshold cryptographic approaches such as the one proposed by Jakobsson [19]. Now, the adversary would have to compromise multiple servers in order to get access to K_{LK}. However, this security comes at significant infrastructure costs of managing multiple servers that execute appropriate distributed protocols for proxy transformation. We argue that these costs would be unacceptable in most enterprises today, however, in the future world of virtual machines it may be possible to split the key across multiple virtual machines on the same physical server with lower costs.

Denial of Service Against *LS*

A potential attack against our design that was also observed in initial testing efforts is denial-of-service against *LS*. This attack would involve an adversary composing a large number of encrypted messages and sending them to valid list aliases from valid subscriber email addresses (which can be spoofed). In the PSELS protocol *LS* does not cryptographically verify authenticity of emails sent by subscribers because the senders' signatures are enveloped by the encryption. Consequently, *LS* may end up executing a large number of cryptographic proxy transformation operations leading to a potential denial-of-service attack.

In our current implementation we mitigate this attack by requiring *LS* to check whether incoming emails are encrypted with the list encryption key, PK_{LK}, using the encrypted session key packet of OpenPGP. Though the adversary can spoof this packet, it imposes an additional hurdle in the adversary's path. SELS [20] addresses this problem by using HMAC based authentication at *LS*, which unfortunately requires fundamental changes to email message formats and is therefore not a deployable solution. Fortunately, the S/MIME Extended Security Services [17] (ESS) include an additional signature wrapping around encrypted envelopes that would enable *LS* to verify sender signatures prior to the transformation. Since RSA signature verification is much cheaper that proxy transformation, the denial-of-service threat would get significantly mitigated. As these ESS services are deployed we will look at integrating them with PSELS.

***LM* Generated Decryption Keys**

In the PSELS protocol and implementation *LM* generates every user's decryption key, K_{U_i}, and sends it to the user. As part of the protocol, *LM* then deletes the key. This can be viewed as a weak security design because ideally only the true owner of the decryption key (in this case user U_i) should generate and have access to the key. In fact, in the original SELS protocol, users added a random number in the proxy key generation process to ensure a strong security design.

In PSELS *LM* generates the decryption keys so as to avoid the need for developing new client-side software and achieve our deployability goals. We see three potential consequences of this design choice: (1) a corrupt *LM* can choose to retain K_{U_i} and decrypt messages intended for the user, (2) a corrupt *LM* can share K_{U_i} with users outside of the list, and (3) a corrupt *LM* can retain K_{U_i} and attempt to avoid revocation at a later point in time. For the first two consequences we argue that *LM* already has his own decryption key, K_{LM}, that allows him to decrypt all emails sent on the list and one that he can share with adversaries if he chooses to do so. For the third consequence, the key update protocol can be used whenever the list moderator changes to ensure that the previous list moderator cannot continue to be a part of the list.

7 Related Work

Proxy Encryption. Previous proxy encryption schemes enable unidirectional and bidirectional proxy transformations by first setting up a transformation agent that is given the proxy key and then sending messages to the agent for transformation [4], [18] and [25]. Unidirectional schemes only allow transformations from some entity *A* to another entity *B* with a given proxy key while bidirectional schemes additionally allow

transformations from B to A with the same proxy key. For PSELS we need a proxy encryption scheme that allows for the transformation from one entity, LS, to many subscriber entities (i.e., to all list subscribers). The El Gamal based unidirectional proxy encryption scheme of Ivan and Dodis [18] is closest in nature to PSELS with the additional relationship between the proxy keys (i.e., $\forall_i\ K'_{U_i} + K_{U_i} = K_{LK}$) imposed to allow for a single list encryption key, PK_{LK}, to suffice. Extending the RSA based unidirectional scheme of [18] in a similar manner will not work because it would require the sharing of the modulus across all list subscribers. Jakobsson [19] and Zhou *et al.* [33] allow for proxy transformation without the need for distributing proxy keys but use costly threshold crypto-systems to ensure the necessary security. Ateniese *et al.* [2] extend proxy encryption schemes with useful properties such as non-interactiveness, which for PSELS might allow for generation of proxy keys without involving both LM's and LS's decryption keys; however, their scheme uses proprietary message formats and bilinear maps that are not easily available in standard cryptographic libraries or tools with interfaces to email systems making deployability very challenging.

Multi-recipient Email Encryption. The problem of sending confidential messages to multiple recipients has been addressed in past via multi-recipient email encryption [28], multi-party certified email [32], secure group communication and broadcast encryption. A major difference between these approaches and ours is that by using a mailing list we remove the user's burden of managing recipient addresses and public keys while still satisfying the confidentiality requirement. In these approaches the sender must manage the sender list and address all of the intended recipient's directly. In multi-recipient email encryption, Wei *et al.* [28] combine techniques from identity-based mediated RSA and re-encryption mixnets to enable a sender to encrypt messages to multiple recipients with only two encryptions (as opposed to one encryption for each recipient in the trivial case). To do so, they use a partially trusted demultiplexer that is akin to LS in terms of its security properties but also use an additional fully trusted CA. Their scheme is not intended for mailing lists and, furthermore, requires development of client-specific plugins. In PSELS the sender needs to execute only one encryption allowing compatibility with existing messaging formats and tools thereby avoiding the need to develop client-specific plugins. In multi-party certified email [32], the sender must maintain each recipient's public key and encrypt the message individually to each recipient. This overhead is avoided in PSELS via the use of mailing lists while still providing confidentiality.

In secure group communication either a trusted group controller (e.g., LKH [30]) distributes session keys to group members or the group members generate session keys in a distributed manner (e.g., TGDH [23]). In either case, list subscribers would have to maintain state on current session keys and update them on every membership change (in PSELS existing subscribers are not affected by the joins and leaves of other members). This makes the use of secure group communication techniques impractical for secure mailing lists as it goes against the nature of the largely offline email use. So-called "stateless" broadcast encryption schemes (e.g., [14], [6]) allow for encryption of messages to a dynamic set of group members without the members requiring to maintain state and executing key updates on membership changes. However, they vary the encryption key and cipher-text sizes depending on the group membership. This

variation cannot be supported by today's mailing lists making such solutions difficult to implement. PSELS, on the other hand, addresses the confidentiality and deployability requirements of secure mailing lists in a practical way.

Secure Mailing Lists. Simple approaches that extend security solutions for two-party email to mailing lists have already been developed; e.g., (`http://non-gnu.uvt.nl/mailman-ssls`). In these solutions, subscribers send emails to the list server encrypted with the list server's public key. The list server decrypts the emails and then re-encrypts them for every subscriber using their registered public keys. Clearly, these solutions do not satisfy the confidentiality requirement as they allow the list server access to decrypted emails. Previously we have developed a Secure Email List Service solution that satisfies the confidentiality problem in mailing lists by using proxy encryption [20]. However, as we discussed in Section 4 this work is not practical for deployment in today's email systems. We have also developed a Certified Mailing List protocol that uses proxy encryption techniques to provide certified delivery in mailing lists [21]. This protocol provides confidentiality using proxy encryption similar to that in PSELS. However, since the primary motivation is a protocol for certified delivery, the protocol results in modifications of messaging formats and special processing at client-side making it impractical for deployment in today's email systems.

8 Conclusions and Future Work

In this work we have described the process of going from the new cryptographic primitive of proxy encryption to a deployable application that secures mailing lists. We chose mailing lists because there is a need to secure sensitive messages in multi-party settings for which email is a convenient, default method. In designing secure mailing lists we identify the need to minimize trust liabilities in the list server for which proxy encryption provides the necessary capabilities. We then defined a component architecture and a protocol geared towards deployability taking into account available COTS tools and configurations of deployed email infrastructures. The resulting PSELS implementation was then tested for functionality, compatibility, and performance.

The PSELS software is now available for community evaluation. We look forward to supporting the software in terms of software patching and update as well enhancing it with new features that the community desires. In addition, we will undertake usability studies to understand the effectiveness of the solution and report the results back to the community.

Acknowledgements

We would like to thank Jim Basney and Von Welch for their comments and suggestions, which greatly improved the design and development of PSELS. We would like to thank Rakesh Bobba for help in development and testing of the software. This material is based upon work supported by the Office of Naval Research (ONR) grant N0001404-1-0562. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the view of ONR.

References

1. B. Adida, S. Hohenberger, and R. L. Rivest, "Lightweight Encryption for Email", Proceedings of Usenix's Symposium on Reducing Unwanted Traffic on the Internet (SRUTI 2005), July 2005.
2. G. Ateniese, K. Fu, M. Green and S. Hohenberger, "Improved proxy re-encryption schemes with applications to secure distributed storage," in Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, February 3-4, 2005.
3. D. Bentley, G. G. Rose and T. Whalen, "ssmail: Opportunistic Encryption in sendmail", in Proceedings of the 13th Usenix Systems Administration Conference (LISA), 1999.
4. M. Blaze, G. Bleumer, and M. Strauss, "Divertible protocols and atomic proxy cryptography", in Eurocrypt'98, LNCS 1403, Springer-Verlag, 1998.
5. D. Boneh and M. Franklin, "Identity based encryption from the Weil pairing", *SIAM Journal of Computing*, Vol. 32, No. 3, pp. 586-615, 2003.
6. D. Boneh, C. Gentry, and Brent Waters, "Collusion Resistant Broadcast Encryption with Short Ciphertexts and Private Keys", Annual International Cryptology Conference (CRYPTO), Santa Barbara, California, USA, August 14-18, 2005, Lecture Notes in Computer Science 3621 Springer pp 258-275.
7. N. Brownlee and E. Guttman, "Expectations for Computer Security Incident Response", IETF Network Working Group, RFC 2350, June 1998.
8. J. Callas, L. Donnerhacke, H. Finney, R. Thayer, "OpenPGP Message Format", IETF Network Working Group, Request for Comments, RFC 2440, November 1998.
9. J. Callas, "Identity-Based Encryption with Conventional Public-Key Infrastructure", in Proceedings of the 4th Annual PKI R&D Workshop, 2005.
10. Y-P. Chiu, C-L. Lei, and C-Y. Huang, "Secure Multicast Using Proxy Encryption", Proceedings of the 7th International Conference on Information and Communications Security (ICICS '05), LNCS 3783, December 2005.
11. S. Crocker, N. Freed, J. Galvin, and S. Murphy, "MIME Object Security Services", IETF Network Working Group, Request for Comments, 1848, October 1995.
12. M. Delaney, Editor, "Domain-based Email Authentication Using Public-Keys Domain-based Email Authentication Using Public-Keys", IETF Internet Draft, September 2005.
13. X. Ding and G. Tsudik, "Simple Identity-Based Cryptography with Mediated RSA", in Proceedings of the RSA Conference, Cryptographer's Track, 2003.
14. Y. Dodis and N. Fazio, "Public Key Broadcast Encryption for Stateless Receivers", ACM Workshop on Digital Rights Management (DRM), November 2002.
15. M. Franklin and G. Tsudik, "Secure group barter: multi-party fair exchange with semi-trusted neutral parties", in Financial Cryptography, 1998.
16. T. E. Gamal, "A Public Key Cryptosystem and a Signature Scheme Based on the Discrete Logarithm", *IEEE Transactions of Information Theory*, pages 31(4): 469-472, 1985.
17. P. Hoffman (Editor), "Enhanced Security Services for S/MIME", IETF Network Working Group, RFC 2634, June 1999.
18. A. Ivan and Y. Dodis, "Proxy Cryptography Revisited", in Proceedings of the Network and Distributed System Security Symposium (NDSS), February 2003.
19. M. Jakobsson, "On quorum controlled asymmetric proxy re-encryption", in Proceedings of the Second International Workshop on Practice and Theory in Public Key Cryptography (PKC'99), volume 1560 of Lecture Notes in Computer Science, pages 112–121, Berlin, Germany, 1999.
20. H. Khurana, A. Slagell, and R. Bonilla, "SELS: A Secure E-mail List Service", in the Security Track of the ACM Symposium on Applied Computing (SAC), March 2005.

21. H. Khurana and H-S. Hahm, “Certified Mailing Lists”, in n Proceedings of the ACM Symposium on Communication, Information, Computer and Communication Security (ASIACCS’06), Taipei, Taiwan, March 2006.
22. H. Khurana and R. Koleva, “Scalable Security and Accounting Services for Content-Based Publish Subscribe Systems”, in the *International Journal of E-Business Research*, Vol. 2, Number 3, 2006.
23. Y. Kim, A. Perrig and G. Tsudik, “Simple and Fault-Tolerant Key Agreement for Dynamic Collaborative Groups”, in Proceedings of 7th ACM Conference on Computer and Communication Security (CCS), 2000.
24. J. Linn, “Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures”, IETF PEM WG RFC 21, 1993.
25. M. Mambo and E. Okamoto, “Proxy Cryptosystems: Delegation of the Power to Decrypt Ciphertexts”, *IEICE Transactions on Fundamentals*, vol. E80-A, No. 1, 1997.
26. B. Ramsdell, Editor, “Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification”, IETF Network Working Group, Request for Comments, RFC 3851, July 2004.
27. D. K. Smetters and G. Durfee, “Domain-based authentication of identity-based cryptosystems for secure email and IPsec,” in Proceedings of the 12th Usenix Security Symposium, August 4-8, 2003, Washington, D. C..
28. W. Wei, X. Ding, and K. Chen,“ Multiplex Encryption: A Practical Approach to Encrypting Multi-Recipient Emails”, International Conference on Information and Communications Security (ICICS), 2005.
29. M. J. West-Brown, D. Stikvoort, K-P. Kossakowski, G. Killcrece, R. Ruefle, and M. Zajicek, “Handbook for Computer Security Incident Response Teams (CSIRTs)”, CERT Handbook, CMU/SEI-2003-HB-002, April 2003. Available at `http://www.cert.org/archive/pdf/csirt-handbook.pdf`.
30. C. K. Wong, M. G. Gouda, S. S. Lam, “Secure group communications using key graphs”, *IEEE/ACM Transactions on Networking* 8(1): 16-30, 2000.
31. P. Zimmerman, *The Official PGP User’s Guide*, MIT Press, ISBN: 0-262-74017-6, 1995.
32. J. Zhou, “On the Security of a Multi-Party Certified Email Protocol”, in proceedings of the International Conference on Information and Communications Security, Malaga, Spain, October 2004.
33. L. Zhou, M. A. Marsh, F. B. Schneider, and A. Redz, “Distributed Blinding for Distributed ElGamal Re-Encryption”, International Conference on Distributed Computing Systems (ICDCS), 2005, pp. 815-824.