SSARES: Secure Searchable Automated Remote Email Storage*†

Adam J. Aviv
Department of Computer and Information Science
University of Pennsylvania
aviv@seas.upenn.edu

Michael E. Locasto
Department of Computer Science
Columbia University
locasto@cs.columbia.edu

Shaya Potter
Department of Computer Science
Columbia University
spotter@cs.columbia.edu

Angelos D. Keromytis
Department of Computer Science
Columbia University
angelos@cs.columbia.edu

Abstract

The increasing centralization of networked services places user data at considerable risk. For example, many users store email on remote servers rather than on their local disk. Doing so allows users to gain the benefit of regular backups and remote access, but it also places a great deal of unwarranted trust in the server. Since most email is stored in plaintext, a compromise of the server implies the loss of confidentiality and integrity of the email stored therein. Although users could employ an end–to–end encryption scheme (e.g., PGP), such measures are not widely adopted, require action on the sender *side, only provide partial protection (the email headers remain in the clear), and prevent the users from performing some common operations, such as server–side search.*

To address this problem, we present Secure Searchable Automated Remote Email Storage (SSARES), a novel system that offers a practical approach to both securing remotely stored email and allowing privacy–preserving search of that email collection. Our solution encrypts email (the headers, body, and attachments) as it arrives on the server using public–key encryption. SSARES uses a combination of Identity Based Encryption and Bloom Filters to create a searchable index. This index reveals little information about search keywords and queries, even against adversaries that compromise the server. SSARES remains largely transparent to both the sender and recipient.

*This work was partially supported by the National Science Foundation through Grant ITR CNS-04-26623.

†all work done at the Network Security Lab at Columbia University Department of Computer Science

1 Introduction

Most email is both sent and stored in a plaintext format. During transmission, encryption standards, such as SSL, can protect a message from eavesdroppers. However, email "at rest" (stored on the server) remains at risk. Servers that store email and provide remote access and easy backups of a user's mailbox are also trusted to protect the email's contents; a compromised server implies the compromise of the users' email, and a user cannot easily prevent such a situation. Even though email content can be secured using public-key encryption (*e.g.,* PGP), this solution alone is not viable for two reasons. First, PGP preserves the headers of the email so that the message can be properly delivered. Consequently, an attacker can still partially compromise the users' privacy by determining who the user is communicating with. More importantly, PGP-style protection relies on the correspondents actively employing the tool. Unfortunately, the use of public-key encryption is not widespread among the general public.

The first step toward a solution to the email "at rest" problem involves the construction of an email system that provides confidentiality protection without the direct interaction of the user. Having a transparent procedure would allow for the average user to not change his/her normal email practices while still being assured of the protection provided. Incoming email can be completely encrypted on the email server as it arrives. More precisely, the email body, headers, and attachments are entirely encrypted using a RSA-style public key so that once encrypted, the email can not be read except by the appropriate recipient. Doing so assures that, regardless of who the sender is, the content will be protected once it arrives. The users' email practices need not change, and they do not need to convince their correspondents to alter theirs.

1063-9527/07 $25.00 © 2007 IEEE
DOI 10.1109/ACSAC.2007.30

Since the server encrypts the entire message with a public key upon arrival, it cannot access any content in the email once the message is encrypted, limiting the amount of data exposure to an attacker that compromises the system. However, the server also cannot provide search capabilities like those supported by email protocols such as POP or IMAP. Although we could move the search process to the client, doing so requires extra processing and bandwidth, since every message must be transferred, decrypted, and then searched. If the client is working from a mobile device or has a large amount of email, this choice involves serious delay. It seems that we have arrived at an impasse: we gain confidentiality protection but must relinquish the ability to search the archive of encrypted emails.

A first approach to remote searches of the encrypted email archive uses a hash table to reference keywords within a message. A user sends a hash of the keyword, and the server uses the hash table to determine which messages matches the request without decrypting any messages or needing knowledge of the keyword being searched. If an attacker compromises the server, however, the attacker would have access to the hash table and could perform a dictionary attack using keywords that are relevant to the victim. The attacker could also watch the user perform searches and initiate a dictionary attack against the hash requests. Both the emails and the search mechanism require protection.

To solve the problem of simultaneously protecting email "at rest" and allowing for keyword-based searching, we present Secure Searchable Automated Remote Email Storage (SSARES). Our system completely encrypts incoming messages and also allows a user to search their email without revealing identifiable information about either the keywords of the messages or the keywords in the search queries. The system is built using a combination of PEKS (Public Key Encryption with Keyword Search) [5], a form of Identity Based Encryption (IBE) that works with a public/private key architecture, and Bloom Filters [4].

Our threat model focuses on two types of attackers. The first can break into the server and download the contents of the mailbox for offline analysis. The second observes the system in action and watches how messages are matched to try and determine the contents. Of course, once the server becomes compromised, all subsequent arriving unencrypted messages are trivially exposed.

2 Related Work

Song *et al.* studied the problem of searching encrypted data by using asymmetric key techniques [15]. Their system requires the user to perform keyword encryption locally so that no information is leaked to the untrusted server. For newly arriving email, the keywords must be stored in an unsearchable format until the user can connect and encrypt the keywords using the searchable format.

Goh presented an encrypted Bloom Filter, Z-IDX, that is secure against adaptive chosen keyword attack [10]. The document is encrypted with a public key on the server, but a Z-IDX is attached to the message only after the client connects, downloads new messages, decrypts them, creates the Z-IDX, and returns it to the server. Once the Z-IDX is created by the user, the user can produce trapdoors, an encryption of the keyword. The server can use the trapdoor to test whether that keyword is contained in the filter. Bellovin *et al.* presented a similar technique, but their scheme requires an independent third party [3]. The system is designed such that two or more parties may share data when they do not fully trust each other. It is geared toward database queries, and as such it does not match our needs because the third party can be considered as untrustworthy as the server itself.

Ballard *et al.* developed a correlation-resistant storage technique for a "survivable" storage network capable of using untrusted nodes [2]. Curtomola *et al.* describe a searchable symmetric encryption scheme with properties similar to Goh [9]. Although their approach is more efficient than SSARES, the system still requires the user to encrypt content locally in order to protect the secret key, and thus would not fit our goal for transparent operations. Other work in this area has been done for a remote file system [8] and for a distributed storage system [1].

Boneh *et al.* introduced a new method to perform IBE with a Weil Pairing [6]. Although a number of papers have leveraged this work, the work most relevant to SSARES is by Boneh *et al.* [5], which introduced the PEKS process. Park *et al.* enhanced the PEKS concept for use as device-specific private keys and describes an application for email gateways [13]. Park also introduced a mode of PEKS that allows for conjunctive keyword searching [14]. Although not currently implemented in SSARES, conjunctive keyword searching would be a valuable feature to add. Gu *et al.* showed how to remove covert channels from the PEKS encryption, and presented an efficient scheme that removed the pairing operation from the encryption procedure [11]. Waters *et al.* used a form of the PEKS procedure to build an encrypted searchable audit log [16]. It uses a three-party system, an encrypted database with PEKS referencing, an audit escrow agent that managed key distribution, and an authentication system.

2.1 PEKS

The PEKS process consists of four functions. They are as follows:

1. $KeyGen(s)$: generates a public/private-key pair, A_{pub} and A_{priv}, given a security parameter s

2. $PEKS(A_{pub}, W)$: given a public-key, A_{pub}, and a word, W, produces the searchable encryption, or the PEKS, called S

3. $Trapdoor(A_{priv}, W)$: given the private key A_{priv} and a word W, it produces an encrypted trapdoor T_W

4. $Test(A_{pub}, S, T_W)$: given the public key A_{pub}, a trapdoor T_W, and $S = PEKS(A_{pub}, W')$, outputs `match` if $W = W'$ and `no match` otherwise

The construction of PEKS that SSARES uses is based on a bilinear map of elliptic curves. It uses two groups G_1, G_2 of prime order p with a bilinear map $e : G_1 X G_1 \rightarrow G_2$ between them. There are also two hash functions that fit the following criteria:

1. $H_1 : \{0,1\}^* \rightarrow G_1$

2. $H_2 : G_2 \rightarrow \{0,1\} \log_p$

The more detailed construction is as follows:

1. $KeyGen(p)$: The security parameter determines the size, p, of the groups G_1 and G_2. It next picks a random α and a generator g of G_1, and it outputs $A_{pub} = [g, h = g^{\alpha}]$ and $A_{priv} = \alpha$.

2. $PEKS(A_{pub}, W)$: computes $t = e(H_1(W), h^r)$ where r is randomly generated, and outputs $PEKS(A_{pub}, W) = [g^r, H_2(t)] = S[A, B]$

3. $Trapdoor(A_{priv}, W)$: $T_W = H_1(W)\alpha$ which is contained in G_1

4. $Test(A_{pub}, S, T_W)$: if $H_2(e(T_W, A)) = B$ then it is a `match` otherwise it is `no match`

A message is parsed for keywords. The original message can be encrypted and each keyword i can be encrypted using the PEKS public-key, A_{pub}, to create a searchable encryption S_i. Each S_i can then be appended to the end of the encrypted message in a list form. For a message M and a public-key encryption algorithm E_{pub}, the resulting remote searchable message M' can be visualized like this:

$$M' = [E_{pub}(M), S_1, S_2, ..., S_i, ..., S_n]$$

To search for a keyword, a trapdoor can be created using the PEKS private key, A_{priv}, and the keyword. The trapdoor can then be tested against the PEKS list using A_{pub}, and if any match, then the keyword is contained in the message.

This construction is non-interactive and can thus be used autonomously between client and server due to the public/private-key structure.

2.2 Bloom Filter

A Bloom Filter is an array, B, of m bits that are initialized to zeros. There is also a set of k independent hash functions that are uniformly distributed between $[0, m-1]$.

To add an entry into the Bloom Filter for a word W, calculate

$b_1 = H_1(W)$
$b_2 = H_2(W)$
...
$b_k = H_k(W)$

Each b_i represents an index into B, and each corresponding location in B is set to 1. To check if a word W is represented in the index, the same hashing procedure is used. If for each b_i there is a corresponding 1 at the location $B[b_i]$, then W is probabilistically represented in the Bloom Filter. If any of the b_i locations have 0, then W is definitively not represented in the Bloom Filter. The probability of a false-positive, or the error of the filter, can be calculated using this formula.

$$(1 - (1 - 1/m)))^{kn})^k$$

where n is the number of records, or words, that are currently represented in the index.

3 SSARES Design

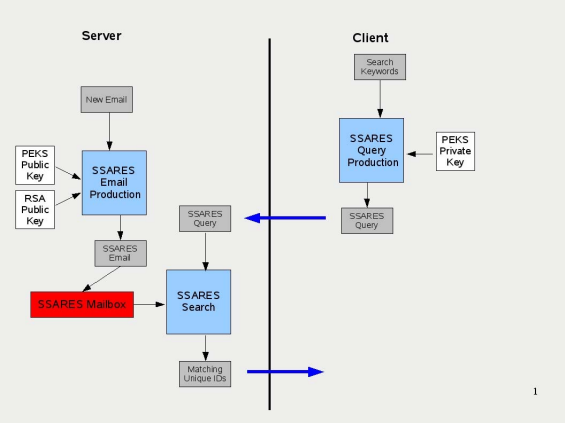
SSARES contains three major components that implement the storage protection and searching capabilities mentioned in Section 1. The first two components handle the encryption of newly arrived email as well as providing the server-side mechanisms for searching the email archive. The third component operates on the client side and handles the composition and issuance of search requests to the server.

The first component, SSARES Email Production, employs RSA-style public key encryption to completely encrypt incoming email, and also extracts keywords from the email to be encrypted using the PEKS public key. The result of the PEKS encryption is a secure searchable form of each keyword, a "peks", that can only be searched by the client using their PEKS private key. Each keyword's associated peks is then appended to the completely encrypted email in the form of a list, a "peks-list." The output form of the email, called an SSARES Email, is an encapsulation of the completely encrypted original email and the peks-list in a new email construct with a unique id associated with it. The SSARES Email is then stored in a SSARES Mailbox to

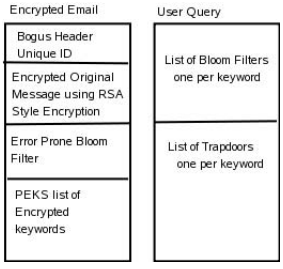
be referenced for searches. The original email is removed from the system and can no longer be accessed except by the client using their RSA-style private key.

The client-side component, SSARES Query Production, enables the user to create a "trapdoor," an encrypted form of a keyword that uses the PEKS private key. A trapdoor and a peks can be securely compared to determine if there are a match without revealing the underlying keyword. Each trapdoor created represents one keyword the user wishes to search for. The output of SSARES Query Production is a SSARES Query, which is simply a list of trapdoors, and is then sent to the server to be used in searching.

The last component, SSARES Search, runs on the server side. Using the client's search request, the SSARES Query, and the client's SSARES Mailbox, SSARES Search performs PEKS testing, a comparison of the trapdoors in the query to the peks list of each SSARES email in the mailbox. All resulting matches are returned to the client, and decrypted on the client side. Using PEKS, the server (or an adversary that controls it) does not learn what the matched keyword was. Figure 1(a) provides a visual representation of how information, keys, and requests are distributed between client and server in the SSARES system.



(a) *Diagram of SSARES System and Client, Server Interaction.*



(b) *Diagram of SSARES Email and Search Query.*

Figure 1. SSARES Design Diagrams

Even a small mailbox with 100 emails, containing 400 keywords each, could require up to 40,000 peks to test on each search, which would create significant delays. To alleviate this overhead, we introduce a Bloom Filter with an adjustable amount of error built in (we call this construct an "error-prone filter"). More precisely, an error-prone filter is a Bloom Filter with a high false-positive rate (25%) so that it becomes difficult to retrieve accurate results when an attacker initiates a dictionary attack. Each SSARES email produced has an associated error-prone filter and a peks-list. The SSARES query has a list of error-less Bloom Filters, query filters, as well as the list of trapdoors. SSARES Search first checks each query filter against the error-prone filter, and it eliminates 75% of the messages on average (only if a match occurs are the trapdoors and peks-list used). This approach allows SSARES to quickly ignore messages unrelated to the user's query. A visual description of the resulting SSARES Query and Email is available in Figure 1(b).

A number of optimizations are meant to make SSARES more practical in terms of search speeds. For example, using multiple peks-list per message, one for each message part (To, From, Subject, *etc.*), allows more precise searching by reducing the total number of peks to test. Also, we use a technique we refer to as Alpha-Sorting, where each peks has the first character of the unencrypted keyword left in the clear to be matched before testing an individual peks. A SSARES Query would not only have a query-filter but also the appropriate peks-list to test and the Alpha-Sorting technique per keyword the user wishes to search for.

3.1 SSARES Component Design

We next examine SSARES's key distribution and the details of what happens when the server receives new mail (SSARES Email Production), when the user requests a search (SSARES Query Production), and finally when the server searches the email (SSARES Search).

Key Distribution The server has three sets of registered keys with the user. The first is a standard RSA public key, the second is the PEKS public key, and the third is the Weil Pairing that is needed to perform PEKS operation. The user's keeps locally on their machine an RSA private key and the PEKS private key.

SSARES Email Production When the server receives email, it must convert it into a SSARES Email as discussed above. This process takes place in two phases. The first phase parses the email for keywords. The parsing choices made by SSARES are discussed in Section 4.1. The second phase completely encrypts the message (headers, body, and attachments), creates the error-prone filter, and generates the peks-lists, one per message part. It is during the second phase that Alpha-Sorting techniques are implemented. The completed SSARES Email format encapsulates the en-

crypted email as its body and the error-prone filter and peks-lists as attachments. SSARES synthesizes headers; the only meaningful headers are a unique message identifier and the date. The SSARES email is now ready for storage, and is placed in the SSARES mailbox. After these two phases finish, SSARES discards both the original unencrypted email and the unencrypted keyword list, securely deleting them from both memory and disk.

SSARES Query Production To assist the user in searching their email, the client-side search component creates a query filter and a trapdoor per keyword the client wishes to search for. The client also specifies which message part to search in for each keyword. Query Production then creates the trapdoor using the PEKS private key and the query-filters. The resulting SSARES Query is then sent to the server-side search component, which performs the actual search. If the user wishes to search for multiple keywords, a query filter and a trapdoor is produced per requested keyword and the Alpha-Sorting technique is used as well. In this case, the SSARES Query contains a query filter list and a trapdoor list with Alpha-Sorting. SSARES does not inject error into the query filters because doing so would most probably cause the server-side error-prone filters to not match the query. The potentially matching email would then be erroneously ignored.

However, refraining from injecting error into query filters offers an attacker the best opportunity to perform a dictionary attack. If the attacker were able to successfully deduce the keyword used to create the filter, he can perform a search and gain information about the encrypted email. The worst case scenario occurs when a user searches for one keyword. One filter and one trapdoor are produced, and the attacker knows that the results of the dictionary attack relates to this given trapdoor. Due to a random coefficient in the trapdoor encryption procedure, however, the attacker cannot match the broken trapdoor to later trapdoor requests. With multiple keyword searches, the attacker would not be able to match a query filter to a specific trapdoor, but the attacker can still replay a search and determine which messages (including more recently received ones) match the SSARES Query.

SSARES Search The server receives the SSARES query, performs a search of the SSARES mailbox, and returns matching messages to the client. For each message in the mailbox, the server first checks to see if any of the query filters matches the error-prone filter. On the first match of a query filter, the server next performs PEKS testing on the appropriate peks-list matching the message part specified in the query. Before each individual peks is tested, SSARES Search first checks the Alpha-Sorting component, and only if there is a match does the search component test the peks against the trapdoor. On the first match of any of the trapdoors, the email's unique ID is added to return list. If none of the trapdoors match any of the peks in the peks-list, then the SSARES query does not match this email, and the algorithm moves onto the next email in the SSARES mailbox. If none of the query filters matched the error-prone filter, then there is no need to perform the PEKS testing, and the algorithm moves on to the next email. Once there are no more emails to test in the SSARES mailbox, the server returns the list of matched email's unique IDs to the user.

Search speed depends on how many PEKS-encrypted keywords must be tested, since this operation is the most time-consuming. However, using the error-prone filter has the potential to eliminate 75% of the messages, and SSARES would only have to perform PEKS testing on 25% of the messages (at most). This reduction does not imply, however, that the search only tests 25% of the PEKS-encrypted keywords on a search. Each email will have a varied amount of keywords, and it could be the case that the 25% of email messages that do need to be checked consists of a much larger percentage of the total PEKS-encrypted keywords. To help combat this we implemented Alpha-Sorting, but as will be shown, Alpha-Sorting alone does not completely solve this problem.

4 Implementation

The SSARES implementation proceeded in two stages. The first stage focused on the construction of two command-line applications written in C. These applications perform PEKS operations and the Bloom Filter creation and testing. These applications provide a core library that is invoked by the Python wrapper scripts written during the second stage of development. The wrapper scripts supervise email handling and parsing, provide input to the command-line applications, construct the SSARES formatted emails and queries, and execute the search.

The PEKS application leverages the Stanford PBC library [12], a C package implementing various Identity Based Encryption algorithms. Our PEKS application uses the bilinear map function, which is the heart of the PEKS encryption process. Using the PBC library, we created an extension that performs PEKS operations (we refer to this component as the PEKS library in the remainder of the paper). It uses `/dev/urandom` for randomness, and SHA1 for hash operations (although we intend to use SHA256 in the future). Our user-level application exposes the basic operations of the PEKS library (key generation, trapdoor generation, PEKS generation, and PEKS testing).

We also developed a new Bloom Filter library. We chose reasonable defaults for the configuration settings exposed by the library. The length of the filters is 200 bytes (1600 bits) and use 5 hash functions. The hashes are obtained using SHA1SUM. We split the 20-byte digest into 5 parts. Each 4-byte segment is interpreted as an unsigned integer

modulo 1600 to get the desired range. The minimum error building was chosen to be 25%, as we already mentioned. The library accomplishes error building by adding extra "words" to the Bloom Filter. A random word is generated by reading from `/dev/urandom`. After this noise is added to the filter, the library computes the error using the formula in Section 2.2. Once the filter error has reached or surpassed the minimum error level, the algorithm halts.

We used this library to provide the core of our second command-line application. This application works in three stages: generate filter, generate query, and test. Generating a filter requires a list of keywords as input, and outputs an error-prone filter (for the minimum false-positive rate specified). No error is built in when creating a query. Instead, the application processes a list of keywords and outputs a separate query filter per keyword. When testing, the application receives a list of query filters and an error-prone filter. Upon the first match of any of the query filters, it returns and reports a match. If all the query filters do not match, then the application returns and reports no match.

4.1 SSARES Email and Query Production

With the Bloom Filter and PEKS applications serving as a foundation, we developed wrapper scripts in Python. The scripts are designed to handle email and command line input to the application. The first script, `SSARES_email_parse.py`, transforms a new email into a SSARES email as described in Section 3. The script executes on the server and makes use of the RSA and PEKS public keys. The script first parses the email for keywords. Keywords are taken from the `To:`, `From:`, `Date:`, and `Subject:` headers. All words from the body of the email are considered keywords, except for common words of three letters or less. Attachment names are also considered keywords. Keywords are organized into fields depending on their position in the email, namely the section from which they were extracted. A separate peks-list is created for each field. Doing so enables more specific searching and can potentially reduce the number of PEKS testing needed to complete a search.

It is also at this phase that Alpha-Sorting is implemented. Within each peks-list, each individual PEKS will have the first letter of the unencrypted keyword exposed, so that during testing, if the unencrypted letter does not match the corresponding letter provided in the query, there is no need to perform the PEKS test. This must be done at this point in the procedure because the original email and the keyword lists are then purged from the system.

Only one error-prone filter is created per message, and it contains the hashes of all the keywords in the message regardless of their field. This organization allows SSARES to eliminate a message from a search with only one test of the Bloom Filter. In contrast, if each field has an error-prone filter, the probability of having to perform PEKS testing increases with each additional filter test. For example, with four Bloom Filters to test per message (each having a minimum error of 25%) every message would on average have at least one filter that matched, and thus at least one of the attached peks-list would need to be tested. We avoid this situation because it would render the filters useless.

A second script, `SSARES_email_search.py`, helps the user query the email archive. It uses the PEKS private key to create trapdoors. The user provides a list of keywords per field. The script then calls the Bloom Filter application to produce the query filter and calls the PEKS script to create the desired trapdoors. We separate each trapdoor into a list according to the queried field so that they may be tested against the proper peks-list on the server. Just as before, the trapdoors, like each PEKS in the peks-list of each SSARES Email, will have the first letter of the unencrypted keyword exposed so that Alpha-Sorting technique can be used. Obviously, since there is only one error-prone filter to test per message, the query filters are not separated. Finally, the script sends the resulting SSARES Query to the server to perform a SSARES search.

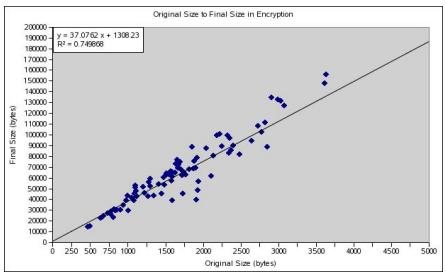
Searching is accomplished using the `SSARES_mailbox_test.py` script. The script accepts as input an SSARES Email created using `SSARES_email_parse.py` and an SSARES Query created using `SSARES_email_search.py` and tests for a match using the algorithm described in Section 3. The output of the script is a list of SSARES Email unique IDs, which can then be returned to the client. As a result, the user can request each matching email individually and decrypt and read it locally.

5 Evaluation

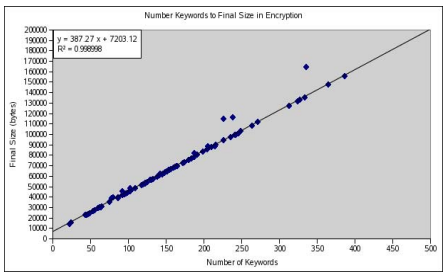
We evaluated SSARES in three parts: email production, query production, and searching. The sample set of emails we used consisted of 100 messages from the Enron Data Set [7]. All tests were run on a Red Hat Enterprise Linux machine with a Pentium 4 CPU at 3.00 GHz.

5.1 SSARES Email Production

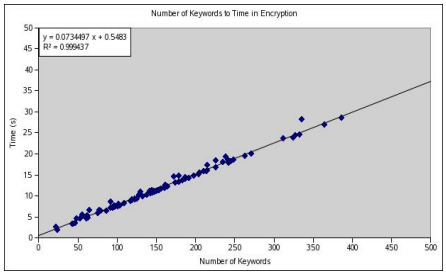
We ran the `SSARES_email_parse.py` script over our sample dataset to produce a SSARES Mailbox. Figure 2 shows the results. Figure 2(a) and Figure 2(b) show the relationship between the original size of the email in bytes and the number of keywords, respectively, and the size of the resulting SSARES Email in bytes with a best fit line. Both graphs display a strong linear relationship, but the number of keywords appears to have more influence on the final size



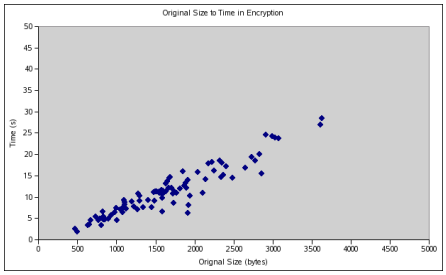
(a) Original Email Size to SSARES Email Size in Bytes



(b) Number of Keywords to SSARES Email Size in Bytes



(c) Number of Keywords to Time of SSARES Encryption



(d) Original Email Size to Time of SSARES Encryption

Figure 2. Graphs for Email Production

of the SSARES Email than the original size of the unencrypted email. This effect is due to the fixed size of the PEKS-encrypted keywords. A larger number of keywords implies more PEKS-encryption results, leading to a larger SSARES Email.

The average original email size was 2523 bytes, while the average SSARES email size was 94,863 bytes, representing an average increase factor of 37, as shown by the slope of Figure 2(a). Given the current and foreseeable cost of storage, we believe this tradeoff to be reasonable for some environments, but not entirely satisfactory. Nonetheless, we are investigating techniques for minimizing this overhead. The slope of Figure 2(b), at 387.27, represents the average amount of space one keyword in the original email takes up in the final encrypted email. Even a relatively small message that is excessively "wordy" can potentially take up more space than an unencrypted message that might be larger but contain fewer distinct words.

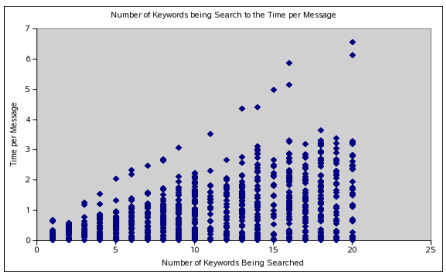
Figure 2(c) and Figure 2(d) display time dependencies in SSARES Email Production. Figure 2(c) shows a direct relationship between the number of keywords and the time of email production. The slope, at 0.07, indicates the average amount of time in seconds per keyword in encryption. Figure 2(d) also displays a linear relationship between the original size of the email and the time of encryption, but the linear relationship is most likely linked to the fact that larger emails tend to have more keywords. The average time of encryption was 17.17 seconds with a standard deviation of 24.17 seconds. The worst case was 179.31 seconds, but this particular email contained over 1,000 keywords. Speed of encryption is not a critical performance factor because email is a transport medium that already operates on the order of minutes. A message arriving after an additional minute would not render the system unusable.

5.2 SSARES Query Production

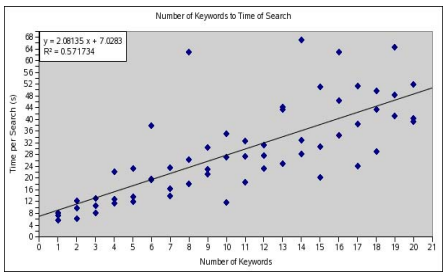
To test query production, we obtained keywords during the email production test by collecting keywords while parsing the body and subject of the email. Three different forms of SSARES Queries with varying amounts of keywords (1 to 20) were produced with the subject and body as the search fields, both with and without Alpha-Sorting.

The first form of SSARES Queries were "first-match" queries in that the first keyword provided was a match for at least one message. The second was "last-match" queries: the last keyword provided is a match for an email. The third form is "no match" queries, where none of the keywords are a match. We produced non-matching keywords by appending numerals to the end of the search keywords. The variety of SSARES Queries were chosen so that the produced queries can be used during searching and provide varying results on different styles of searches.

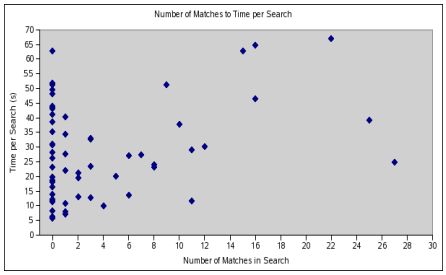
Figure 3 displays the results from the SSARES query production test. Figure 3(a) shows the relationship between the number of keywords and the time it takes to produce a query. Query production is relatively fast, even with an excessive amount of keywords. Using twenty keywords,



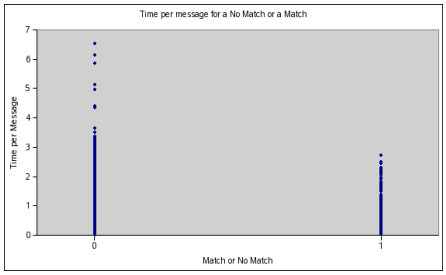
(a) Number of Keywords Being Searched to Time (sec) per Message in Search



(b) Number of Keywords being Searched to Time of Search



(c) Number of Matches to Time (s) of Search



(d) Time (s) per Message for a Match or No Match

Figure 4. Graphs for Overall Searching on the Subject without Alpha Sorting

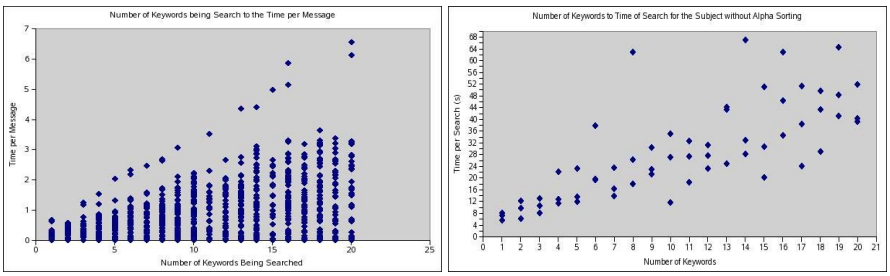
it took less then 2 seconds to produce a query. This time is an important performance factor, because total searching speed is dependent on how fast the client can produce a query. The size, in bytes, of the resulting queries is displayed in Figure 3(b) as it relates to the size, in bytes, of the keywords being searched for. There is a semblance of a linear relationship, most likely due to the fixed amount of information that must be produced (the filter and the trapdoor) and the fixed size of these. The size of queries seems to be reasonable, at most 9 KB, and should not pose serious performance issues.

5.3 SSARES Search

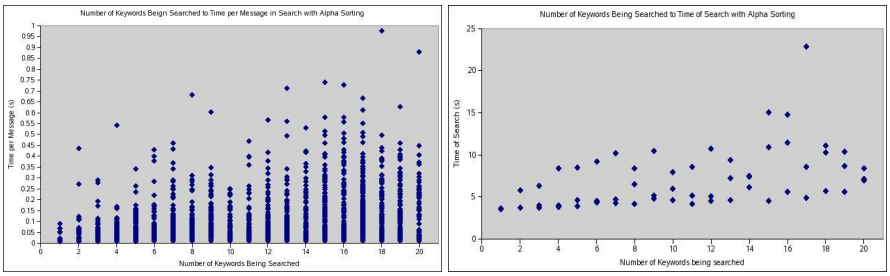
The SSARES Queries produced by the SSARES Query Production testing were used to run the search testing over the SSARES Mailbox produced in the email production testing. Figures 4(a) and 4(b) show how the time of searching in seconds relates directly to the number of keywords being searched for, but it is also dependent on the number of keywords per message in the mailbox, as expected. There is a general decrease in search speed (increase in search time) as the number of keywords increases, and the outlying points above the grouping represent the searching of the email that contains over a thousand keywords. Even without Alpha-Sorting (using just the error-prone filters) it only takes about 7 seconds to search the largest message and only 2.7 seconds on average, but one can see clearly in Figure 4(b) that these times add up to a fairly slow overall searching speed. A small search of 1 to 6 keywords over 100 emails can take half a minute. It should also be noted that these search results were done on the subject of the emails, whose associated peks-list is much shorter then the body of the email. One particular search with 8 keywords took over a minute, which is probably due to matching the messages with longest peks-list. A query with just 1 keyword can have this effect, and a search that takes over a minute is a significant delay for reasonable use and justifies the use of filters and Alpha-Sorting to increase the speed of the search. Without the filters, a search could take hours, but even with filters alone SSARES is not fast enough to be reasonably practical.

The effects of the Bloom Filter on searching speed is drastic. Figure 5(a) shows this best. When a query fails the filter, we can eliminate it from the search in under a second, but if the filter was passed, then the time per message can be between 1 and 7 seconds. Figure 5(b) is even more encouraging, as it shows how many messages per search we were able to eliminate using the filter. On average, we were able to eliminate 76% of the messages, which also matches our minimum error rate of 25%. Changing the minimum error of the filter should have effects on the overall searching speed, at the cost of reduced privacy.

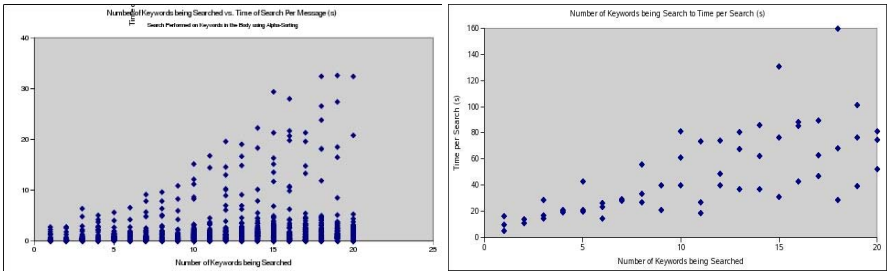
Additionally, search testing was performed with the added Alpha-Sorting technique, which had a profound effect on search speed. Looking at Figure 6, one can see that with just the filters, average search speed for searching the Subject of the email were 28.88 (s) and 0.27 (s) per email. Alpha-Sorting improved the speed to 7.00 (s) per



(a) Search Speed for Subject Search without Alpha-Sorting



(b) Search Speed for Subject Search with Alpha-Sorting



(c) Search Speed for Body Search with Alpha-Sorting

Figure 6. Search Speed Comparison with Alpha-Sorting

search and 0.05 (s) per email. But, search testing on the body, whose peks-list contains many more keywords than the subject, shows how there is still a need for speed improvement as the number of peks to test increases. Using Alpha-Sorting, the 46.47 (s) average search and 0.45 (s) average per email is still too slow to offer true practicality when it comes to searching the largest peks-list of the email, namely the body.

6 Conclusion

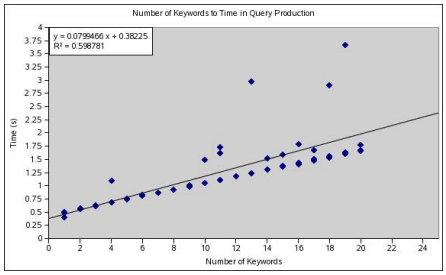
We have presented SSARES, a novel system that takes a practical approach to addressing the problem of simultaneously securing email at rest and allowing search of that email. The combination of Bloom Filter with intentionally added errors, PEKS encryption, and Alpha Sorting provides an automated and transparent process so that normal email practices of sender and receiver need not change. Instead, a list of automatically generated keywords is compared (in a secure manner) with a search query. SSARES helps improve the security of server–side email storage, and if properly implemented, can have a significant impact on the privacy of user email and other information stored with third party providers.

References

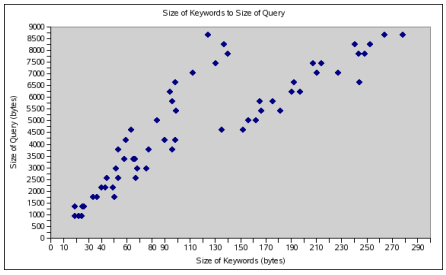
[1] S. Artzi, A. Kiezum, C. Newport, and D. Schultz. Encrypted Keyword Search in Distributed Storage System. *MIT CSAIL Tech Report*, MIT-CSAIL-TR-2006-010, February 2006.

[2] L. Ballard, M. Green, B. de Medeiros, and F. Monrose. Correlation-Resistant Storage via Keyword Searchable Encryption. *Cryptology ePrint Archive*, Report 2005/417, 2005.

[3] S. M. Bellovin and W. R. Cheswick. Privacy-Enhanced Searches Using Encrypted Bloom Filters. *<www.cs.columbia.edu/ smb/papers/>*, DRAFT, 2006.

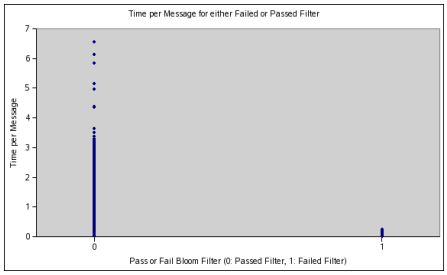


(a) Number of Keywords to Time in Query Production

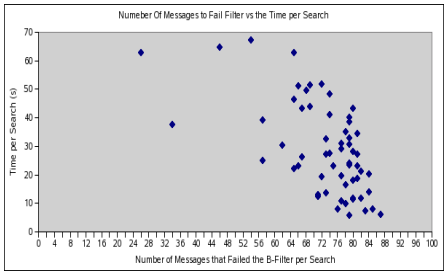


(b) Size of Keywords (bytes) to Size of Query (bytes)

Figure 3. Graphs for Query Production



(a) Time per Message for Either Failed of Passed Bloom Filter



(b) Number of Messages to Fail Filter to Time(s) per Search

Figure 5. Effects of the Bloom Filter

[4] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[5] D. Boneh, G. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In C. Cachin and J. Camenisch, editors, *Proceedings of Eurocrypt*, pages 506–522, 2004.

[6] D. Boneh and M. Franklin. Identity-Based Encryption from Weil Pairing. *SIAM Journal of Computing*, 32(3):586–615, 2003.

[7] CALO Project. Enron Data Set. <Enronmail.com>, 2004.

[8] Y.-C. Chang and M. Mitzenmacher. Privacy Preserving Keyword Searches. In *Proceedings of ACNS*, pages 442–455, June 2005.

[9] R. Cutmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable Symmetric Encryption: Improved Definition and Efficient Constructions. In *Proceedings of ACM CCS*, November 2006.

[10] E.-J. Goh. Secure Indexes. *Cryptology ePrint Archive*, Report 2003/216, 2003.

[11] C. Gu, Y. Zhu, and Y. Zhang. Efficient Public Key Encryption with Keyword Search Scheme from Pairings. *Cryptology ePrint Archive*, Report 2006/108, 2006.

[12] B. Lynn. The Pairing Based Cryptography Library. *<crypto.stanford.edu/pbc/>*.

[13] D. Park, J. Cha, and P. Lee. Searchable Keyword-Based Encryption. *Cryptography ePrint Archive*, Report 2005/367, 2005. Available at http://eprint.iacr.org.

[14] D. Parl, K. Kim, and P. Lee. Public Key Encryption with Conjuctive Field Keyword Search. In C. Lim and M. Yung, editors, *Proceedings of WISA*, pages 73–86, 2004.

[15] D. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *IEEE Symposium on Security and Privacy*, pages 44–55, May 2000.

[16] B. Waters, D. Balfanz, G. Durfee, and D. Smetters. Building and Encrypted and Searchable Audit Log. *Proceedings of ISOC NDSS*, February 2004.