Using Untrusted and Unreliable Cloud Providers to Obtain Private Email

Nicolas Chiapputo[1], Yvo Desmedt[2] and Kirill Morozov[3]
[1]*Independent Researcher, U.S.A.*
[2]*Department of Computer Science, The University of Texas at Dallas, U.S.A.*
[3]*Department of Computer Science and Engineering, University of North Texas, U.S.A.*

Keywords: Cloud Security, Email Security, Secret Sharing, Perfectly Secure Message Transmission.

Abstract: A recent trend for organizations is to shift to cloud services which typically include email. As a result, the natural privacy concerns for users stem not only from outside attackers, but from insiders as well. Our solution does not rely on unproven assumptions and does not need a PKI. To achieve this, we partially rely on concepts from Private and Secure Message Transmission protocols, which are built on top of secret sharing. This technology allows us to distribute trust over email providers. Hence, the system remains secure as long as hackers are unable to penetrate a threshold number of providers, or this set of providers does not form a coalition to attack their users. The prototype of our proposed system has been implemented as an add-on for the Thunderbird email client, using Mozilla's Web Crypto API and Rempe's secret.js library. It currently supports the following secret sharing schemes: the 2-out-2 additive scheme (set as a default), the k-out-n threshold Shamir scheme, and the Rabin and Ben-Or robust scheme.

1 INTRODUCTION

Previously, public and private organizations, such as universities and companies, used to maintain their own email services. In recent years, we see a trend for these organizations to shift their computer system operations to the cloud. Private users have also utilized web-based email services since the 1990s. All this time, concerns for the user data privacy were growing, and webmail-related data breaches kept resurfacing in the recent years with alarming frequency.

For decades, email protection efforts were mainly limited to prevailing forms of encryption provided by the software such as PGP and its genus. However, these encryption techniques rely on unproven assumptions, i.e., on hardness of the computational aspects of some mathematical problems.

In this paper, we propose a different approach to email security which partly relies on techniques underlying Private and Secure Message Transmission (PSMT), namely the *secret sharing* technology. Roughly speaking, this allows us to distribute trust over several email providers.

Secret sharing is one of the cornerstones of theoretical cryptography. It uses shares such that a particular number of these, called the *threshold*, will be required to recover the message. At the same time, any number of shares below the threshold will reveal no information about the message, in the strongest sense (called unconditional security). Known to mathematicians as a mechanical problem (Liu, 1968), secret sharing was realized in the digital world by Blakley (Blakley, 1979) and Shamir (Shamir, 1979) (independently) in 1979. This technology is well tested by time. In particular, it was proposed to secure the launching of nuclear missiles since the 1980s (Simmons, 1990, p. 437).

When secret sharing is applied to email security, we assume that the receiver has e-mail addresses with different providers. An email message will be "split" into several shares each to be sent over a different provider. This way, the need for key management will be eliminated, and hence the Public-Key Infrastructure (PKI) will not be required.

1.1 Related Works

Although the works by Shamir (Shamir, 1979) and Blakley (Blakley, 1979) are very extensively cited, they are quite rarely used in industrial applications as stand-alone primitives. The only closely related work, which the authors are aware of, is the report by Oren and Wool (Oren and Wool, 2009) which used secret sharing in a similar email security setting. Their

Chiapputo, N., Desmedt, Y. and Morozov, K.
Using Untrusted and Unreliable Cloud Providers to Obtain Private Email.
DOI: 10.5220/0012090700003555
In *Proceedings of the 20th International Conference on Security and Cryptography (SECRYPT 2023)*, pages 171-182
ISBN: 978-989-758-666-8; ISSN: 2184-7711

work is different from ours in that they used only 2-out-of-2 secret sharing and did not implement a robust scheme (hence not providing data integrity). At the same time, they introduced a linguistic encoding which makes both of the secret-shared messages meaningful.[1]

This lack of interest by the industry is quite remarkable in the light of a straightforward application to protection of cloud storage with unconditional security—see, e.g., (Attasena and Harbi, 2017; **?**) for surveys of the literature on this topic.

1.2 Our Contribution

The proposed system is implemented as an add-on to the Thunderbird email client, and it is made available via Github (git, 2022). The code is written in Javascript using Mozilla's Web Crypto API (web, 2021b) and Rempe's secret.js library (Rempe, 2019). The add-on uses a modular design, which allows a flexible setting of the number of supported e-mail providers, the number of untrusted providers to be tolerated, as well as other security parameters. The add-on has an option of automatic deletion of the message copies (shares) on the email provider servers, further lowering the chances for unauthorized access. It is guaranteed that the email messages are protected against access by an unauthorized set of the email providers, with unconditional security.

Specifically, a current version of the add-on allows users to distribute their e-mail messages together with attachments using one of the following schemes: 2-out-of-2 additive scheme, k-out-of-n threshold Shamir scheme (Shamir, 1979), and the Rabin and Ben-Or robust scheme (Rabin and Ben-Or, 1989) (throughout this paper, we will refer to it as the RB scheme, for short). Also, it allows an easy integration of different secret sharing schemes, including those for arbitrary access structures.

1.3 Organization

This paper is organized as follows: our notation, the software used, and the underlying cryptographic primitives are described in Section 2. A high-level description of the proposed architecture is presented in Section 3. Details of the implemented functionality are discussed in Section 4. The cryptographic aspects are discussed in Section 5. Section 6 presents and discusses the simulation results. Finally, conclusions and future works are discussed in Section 7.

[1]This useful feature adds to the user privacy in case of the "Big Brother"-style surveillance.

2 PRELIMINARIES

2.1 Notation

A uniformly random selection of an element x from its domain X is denoted as $x \leftarrow_R X$. A bitwise XOR is denoted by "$\oplus$".

Parties are called *honest* when they follow the described protocol and do not attempt to eavesdrop a secret. Adversaries could be passive (sometimes called "semi-honest"), or active. Passive adversaries will follow the described protocol, but will attempt to recover a secret in an unauthorized way. Active adversaries do not have to follow the given protocol.

2.2 Thunderbird

Thunderbird is an open-source e-mail client developed by Mozilla. This client allows users to read any of their e-mail accounts from separate domains into one location. Since it is open-source, it is highly customizable through user-made themes and add-ons to add extra functionality to the software. The add-ons are developed in JavaScript as a result of Thunderbird being built on top of the Mozilla web platform that is shared with Firefox. This allows the add-ons to be cross-platform so they can be developed for any operating system or device that Thunderbird is developed for.

We chose Thunderbird to host the add-on both because of the cross-platform ability, and because it is able to connect with multiple e-mail servers and to get direct access to a user's full e-mail history. This allows the add-on to easily search through the e-mails and find matching shares for reconstruction. The browser extensions do not normally have this ability, so that Thunderbird being an email client is the obvious choice.

Thunderbird recently underwent an overhaul of its add-on environment, moving from legacy extensions to a more unified WebExtension API (web, 2021a). Thunderbird has several advantages, which are beyond the scope of this paper, such as an active online community for add-on developers supported by Thunderbird (top, 2021).

Because Thunderbird incorporates the majority of Firefox's features, Thunderbird add-ons have access to Mozilla's WebCrypto API (web, 2021b). This API is developed by Mozilla to allow access to cryptographic primitives. Using this API, the add-on can generate cryptographically-strong random values using a Crypto object with the `getRandomValues()` method.

The add-on also has access to the TextEncoder

and TextDecoder APIs developed by Mozilla. These APIs allow encoding and decoding to and from strings and `Uint8Array`s (arrays of eight-bit unsigned integers representing the character codes of the string). They are used in this add-on to more easily compute XOR calculations using bytes instead of characters by converting strings into byte streams (of type `Uint8Array`) and vice versa as well as decoding the information in the attachment files generated during the share generation phase.

2.3 Secret Sharing and Access Structures

Secret sharing is a cryptographic protocol which allows one to split sensitive data (a secret) into pieces (called *shares*) which individually[2] provide no information about the secret.

More generally, one defines a certain number k, called a *threshold* such that less than k shares provide no information about the secret. At the same time, k shares (or more) allow an efficient reconstruction of the secret. Clearly, k cannot exceed the total number of parties n (email providers, in our case).

The *access structure* is a collection of sets (of parties) who can reconstruct the secret. The access structure just described is called a *threshold access structure* and the schemes which realize it are the *threshold schemes*. In particular, the Shamir scheme (Shamir, 1979) used in our implementation is of this type. For the k-out-of-n threshold structures and schemes described in the previous paragraph, we will use the notation (k,n), e.g., a (k,n)-threshold scheme.

The *general access structures* support arbitrary collections of access sets. These may reflect different levels of trust which the data owner may assign to different providers. For example, different thresholds may be assigned to different groups of providers. It is easy to extend our implementation to support such access structures.

2.3.1 2-out-of-2 Additive Secret Sharing

A secret value is shared between two parties in such a way that each individual share gives no information about the secret, while reconstruction is possible from both shares.

An easy way to implement it, assuming that the secret is a binary string $s \in \{0,1\}^m$, is to use a (binary) one-time pad scheme, where the key represents the first share $s_1 \leftarrow_R \{0,1\}^m$ and the ciphertext represents the second share $s_2 = s \oplus s_1$. The reconstruction is $s = s_1 \oplus s_2$.

Note that the above is a special case of a threshold scheme with both the threshold and the number of parties equal to 2, hence we will be referring to it as $(2,2)$-additive scheme.

Finally, we remark that the $(2,2)$-additive scheme was proposed during the US Clinton administration for key escrow (dep, 1994; Cli, 1993).

2.3.2 Shamir Threshold Secret Sharing Scheme

The $(2,2)$ scheme suffers from two potential security problems. First, two of the servers may collaborate to recover the secret. To increase the security, the aforementioned scheme can easily be adapted to an (n,n) scheme. Secondly, any (n,n) scheme does not provide a backup in case some of the serves is down. The scheme we now discuss allows to deal with these problems, provided we choose the parameters carefully.

System parameters (Shamir, 1979): A field $\mathbb{F}_p$, a number of shares n, a threshold k, where $k \leq n$, and a set $(\alpha_1, \ldots, \alpha_n)$ of distinct and public elements of $\mathbb{F}_p$, which serve as id's of the parties. To share a secret $s \in \mathbb{F}_p$ for some prime $p > n$, $k-1$, the coefficients $a_1, \cdots, a_{k-1} \in \mathbb{F}_p$ are chosen uniformly at random to form the polynomial $f(x) = s + a_1 x^1 + \cdots + a_{k-1} x^{k-1}$. The value $f(\alpha_i)$ is the share s_i that is given to party p_i. Any set of at most $k-1$ shares provide no extra information on the secret.

To reconstruct the secret, at least k honest parties must submit their share s_i. For every k distinct $\alpha_1, \cdots, \alpha_k$ and $s_1, \cdots, s_k$ values, there exists a unique polynomial $q(x)$ of degree at most $k-1$ such that $q(\alpha_i) = s_i$ for $1 \leq i \leq k$. Hence, the reconstruction algorithm uses Lagrange interpolation to compute the secret as $s = q(0)$.

2.3.3 Rabin and Ben-Or Robust Secret Sharing Scheme and its Variant

While Shamir secret sharing protect against accidental or unauthorized deletion (or unavailability) of shares by less than $n-k$ parties, it does not give any protection against active adversaries. It was observed by Tompa and Woll (Tompa and Woll, 1987) that incorrect shares submitted at the reconstruction in the Shamir scheme will result in incorrect secret.

A goal of the so called *robust* secret scheme is to ensure reconstruction of a correct secret. Rabin and Ben-Or (Rabin and Ben-Or, 1989) proposed to use the so-called check vectors for this purpose. In the context of unconditional message authentication codes (MAC), their schemes can be interpreted as fol-

[2] In special cases, it is possible that some individual shares may allow recovery of the secret, but this is just an academic possibility, which typically is not used.

lows: the share generation algorithm produces shares (according to some secret sharing scheme), as well as the MAC keys/tags for each pair of parties. This way, each party obtains their share, the tags which authenticate it for each respective party, and the keys which authenticate shares of all other parties.

At the reconstruction, one accepts only the shares supported by majority of the parties. By "supported", we refer to the fact that the corresponding tag verifies correctly for the corresponding key. It is implicitly assumed that parties always support themselves (although no tag is generated for this case)—this is just counted as "plus one" vote for each share.

Finally, note that when using the RB scheme (which provides data integrity), the probability ε for dishonest e-mail providers to successfully modify messages (without being noticed) cannot be zero. The reason is that the RB scheme relies on MAC, which can only guarantee a positive ε. By increasing the length of the MAC, ε can be made sufficiently small. Note that the non-zero ε is what makes our approach different from the original work on Private and Secure Message Transmission (PSMT) (Dolev et al., 1993). We observe the later variants of PSMT, e.g., (Franklin and Wright, 2000), incorporate a non-zero ε as a relaxed reliability requirement. Further details on PSMT is beyond the scope of this paper.

2.3.4 Universal Hash Functions

Universal hash functions were introduced by Carter and Wegman (Carter and Wegman, 1979) and their use for authenticating messages by Wegman-Carter (Wegman and Carter, 1981). In one of their algorithms, a message $\boldsymbol{m}$ is a vector over a field $\mathbb{F}$ and a key k is a single element of $\mathbb{F}$. Specifically, in Carter and Wegman's original algorithm, for the message $\boldsymbol{m} \in \mathbb{F}^{n+1}$ and key $k \in \mathbb{F}$, the function is computed as $y = \sum_{i=0}^{n} m_i k^{n-i}$, where $m_i \in \mathbb{F}$ are the elements of $\boldsymbol{m}$ for $i = 0, \ldots, n$.

A lot of fast Universal Hash Functions were developed; see, e.g., (Bierbrauer et al., 1994; Johansson et al., 1994; Afanassiev et al., 1997). For our implementation, we selected PolyQ32 (Krovetz and Rogaway, 2001), which is designed to hash strings in 32-bit blocks.

Note that since the collision bounds for the Krovetz-Rogaway approach are linear in the message size n, it is important to implement a method of controlling the collision probability to improve security. The collision bound is also linear in the inverse of the size of the key space $|K_{32}|$. By choosing the key as multiple elements from this set, the collision probability can be decreased to $(1/|K_{32}|)^m$ with respect to the number of elements m chosen.

3 HIGH-LEVEL DESCRIPTION OF THE PROPOSED FUNCTIONALITY

The secret sharing Thunderbird add-on currently implements the $(2,2)$-additive scheme, the (k,n)-threshold Shamir scheme, and the RB robust secret sharing scheme (using Shamir shares and MAC based on universal hashing). The parties (share holders) of the secret sharing schemes are the email providers.

Hence, when using a (k,n)-threshold scheme, n is the number of e-mail providers with which the recipient has an e-mail account (or, e.g., the employer of the sender does).

3.1 Sending a Message

With the add-on, Thunderbird provides an easily-accessible button on the compose message window that allows the user to secret share a message, including attachments, in the compose window. When secret-sharing a message, the add-on has a feature to automatically fill out the `to:` fields of the different share messages. Each e-mail sent contains a unique 128-character hex string identifier (UID).

The add-on is also integrated with Thunderbird's address book system. The user is required to store their contact's information in a preferred address book that is selected using the popup preferences window. The email addresses associated with secret sharing communication are stored in a comma-delimited list in the "Notes" section of the contact information. The user then adds the contact's main email address as the receiver in the original message. Then, when the user secret-shares the message, the add-on will find the associated emails and auto-fill the fields when generating the new messages with the secret shares.

Since the purpose of the add-on is to prevent enough information to reconstruct the message being on any one e-mail server, the user is expected to send each message (or at least a number of the messages less than the threshold) from a different e-mail account (e.g., a Gmail, Yahoo, ProtonMail, or Outlook account). The recipient addresses should not include more than a threshold amount from any one provider.

3.2 Receiving a Message

When a user opens any one of the secret shared e-mails, they are provided with a custom button as part of the add-on. Clicking on this button initiates the reconstruction process. Users can view the reconstructed message and download any attachments

through a custom window opened after the reconstruction.

When reconstructing the email, the add-on is able to automatically detect the scheme and parameters used to share the message based on the body contents of the received message. To retrieve shares other than the one the user is viewing, the add-on searches through other email inboxes on the system to match with the aforementioned unique 128-character (64 byte) hex UID in the subject line. The add-on then takes the shares and, if applicable, the keys and tags from each of these files. If reconstruction is possible, the add-on is able to separate the subject, body text, and any attachments and display them in a custom window. The user can then select the attachments and download them to their local system exactly as they would with a regular, unsecured email communication.

3.3 Further details

The add-on implementation is centered around a background script and a pop-up window. Both of these, along with all the necessary permissions, image icons, and add-on information, are defined in the manifest file (`manifest.json`) included in the open source repository (git, 2022). The pop-up window allows the user to adjust the system parameters including the scheme, number of shares, and threshold for reconstruction. This pop-up window (`popup.html`) is written in HTML with custom CSS to improve the interface.

4 FUNCTIONALITY DETAILS

Let us describe the details of our implementation and justify some of its features.

4.1 Sending a Message

When the user clicks on the compose window button, the script first removes the subject line on the email. The body text of the email is prepended with "SUBJECT: " followed by the original subject line and followed by an empty line. The subject line is then replaced with a randomly generated 128-character (64-byte) hex string. This will be used to identify the shares on the receiving end during the reconstruction process.

Next, the add-on checks if any attachments have been added to the compose window. A header string is created with information about the attachments to allow the reconstruction to correctly parse the e-mail content and the attachments. The first line of the header contains the number of attachments being shared in the format `count=<n>` where `<n>` is the number of attachments.

Following this, there is a line for each attachment that contains the name, MIME type, and size in bytes of the respective attachment with each piece of information delimited by a comma. This information is prepended to the contents of the currently written message (subject line and body text) to create the current secret value.

To differentiate between the end of the e-mail contents and the beginning of the attachment contents, a single null character delimiter is appended to the end of the current secret value. For each attachment, the file object is taken and its contents are appended to the end of the secret. There are no delimiters necessary between the file contents as the size of each of the files is stored in the header information. This allows us to avoid any issues caused by setting delimiters that users may put into their e-mails (e.g., random strings of characters, new line characters, lengths of equal signs, etc.).

This data is then converted into byte data by converting the string contents into a `Uint8Array` using the `TextEncoder` API where each index in the array is a single byte. The body text is then changed to a message that tells the recipient the name of the scheme, the number of shares created, and the threshold for reconstruction.

If the user wishes to attempt to automatically find the addresses of where to send the secret shared material, the add-on requires the user to fill in the `to:` field of the original compose email with an email address of a contact added in the user-selected address book (as selected from the preferences popup window). To retrieve the email addresses the user has saved (if any) to use when secret sharing with the selected recipient, the add-on first retrieves the selected address book. When the user selected a preferred address book in the settings menu on the popup window, the add-on stored the internal address book ID value to the `addressBook` local storage value.

The add-on then uses the WebExtension folders API to search for this address book. If the search fails, an error will return. The add-on listens for this error and, if none is found, the list of contacts is returned back to the main thread. However, if the address book is not found, this likely means that it has been deleted. This is because the add-on internally stores the ID value instead of the name of the address book, as the name can be changed but the ID can not.

In the event that the selected address book has

been deleted, the add-on will recognize this issue and search through the current available address books for the "Personal Address Book." This is the default address book in Thunderbird and can not be deleted, so it is guaranteed to exist. The local storage value for the address book preference is then overwritten with the ID of the default address book. It is important to note that this check for the existence of the selected address book is also performed when the add-on loads and when the preferences popup window is opened to ensure the preference option remains current. Then, the contacts of the selected address book are returned back to the main thread.

Once the main thread receives back the list of contacts, it will first parse the original `to:` field for the display name and email address of the receiver. Thunderbird formats this string as `DisplayName <email@address>` where the angle brackets surround the contact's email address. This can be easily parsed to retrieve the display name and email address. With this information, we can then iterate through the address book (there is no search API with the given information) to look for a contact with a matching display name and email address.

Once the contact is found, the add-on will access the content of the "Notes" section of the contact. This content is assumed to be a comma-delimited list of email addresses and is returned as a string. The string is split on each comma and the substrings at each index are trimmed of leading and trailing whitespace to clean the data. For user readability, the display name parsed earlier is appended in front of each email address and the address is surrounded by angled brackets to create the format `DisplayName <email@address>` that Thunderbird prefers for the `to:` field.

After this step, the add-on will take the combined data from the header text, subject line, body text, and attachment data and send the combined string through the selected secret sharing algorithm (either (2,2) Additive, Shamir, or Robust) along with the number of shares, the threshold, and the original compose details (to preserve the hex subject identifier and informative body text). After the secret sharing algorithm is complete and the necessary number of shares are generated, the add-on will open one compose window for each share.

In each window, each of the pieces of the respective share (share, keys, and tags as applicable to the current scheme) are added as attachments to the new e-mail. The subject line for each e-mail contains a common 64 byte hex string. This string is used as an identifier during reconstruction to allow the add-on to find which e-mails contain shares to be used in the same instance of reconstruction process.

4.2 Receiving a Message

To reconstruct the messages, the user opens any one of the messages that contains a share and clicks on the customized button in the message display view. Since some e-mail providers append to the subject line (e.g., "[EXT] " in Outlook, or "Fwd: " for forwarded e-mails), the add-on uses the regular expression `/[0-9a-f]{128}/` to eliminate all but a 128 character long hex string from the current subject line. This result is then used as a query to search for messages with matching subject lines. This query searches for any e-mails with matching subject lines under any account from any incoming folder.

Thunderbird helpfully provides attributes for each folder in each account and assigns them a type value that can take the value, among other less important values, "sent". In the event that a user sends these protected e-mails to themselves, the e-mails will appear in both an outbound and inbound folder. When querying for messages given a unique subject line, messages sent from the user to their own e-mail address, or addresses, will appear twice. This will result in an error during share generation as the implementation will find twice as many messages as it needs and will not know which ones are duplicates without parsing the attachments.

To fix this, we can change the message query to only look for folders of type "inbox". However, many users wish to use custom folders to sort their e-mails. These custom folders commonly have an undefined type. It is then easier to remove all query results from "sent"-type folders to ensure that there are no duplicate messages.

The messages found from non-sent-type folders are parsed for attachments with titles matching the format of the shares, keys, or tags to find the relevant reconstruction information. Since there is no API for parsing the individual attachments, the raw attachment data must be retrieved and parsed to find the individual pieces of information and then converted from Base64 format.

To facilitate parsing the raw message data for the attachments, the raw data is sent to the `getAttachmentData()` function. The raw data is then split at every instance of the string "Content-Disposition: attachment;". While the raw data file does actually define a specific boundary string in the beginning of the file, it is easier to simply split on sections where we know there are attachments rather than look through each section between two boundaries and figure out if it is an attachment.

Splitting the raw data on the content disposition string results in $n+1$ strings for a message with n attachments where the remaining string is the header information followed by the email contents and other information related to the delivery of the email. The format of each string resulting from the split is such that the first and third lines are blank. The second line provides the name of the file using the format `filename="<f>"` where `<f>` is the name of the file. To then get the name of the file, we first split the current string into another array of strings based on the newline delimiter. Then, we can simply take the substring from the eleventh character to the third to last character, inclusively. We omit the final two characters as these are the closing quotation mark and a newline character.

Before iterating through the attachments to store the content data, we must first verify that this is a share, tag, or key file sent from the add-on since it is possible that a user may have attached a separate file to the secret share message (this is not suggested, but is possible for users to do). This is done through a simple regular expression that checks if the filename is "share"; "tag-" followed by one or more digits, then a "-", and one or more digits; or "k-" (for key) followed by one or more digits, then a "-", and one or more digits. In the regular expression format, this is written `/(share|tag-[0-9]+-[0-9]+|k-[0-9]+-[0-9]+)/`.

If the filename matches this regular expression, then the add-on iterates through the array of strings split from the current section of the raw message data starting from the fourth line as the first three were two empty lines and the filename. At each line, we first check if the string starts with 14 hyphens. This is because the ending delimiter for the attachment data section is formatted such that it starts with 14 hyphens followed by 24 hex characters (A through F). While we could again use a regular expression, it has actually been shown that using the JavaScript `String.startsWith()` function is much faster, especially on small strings. If this returns true, then we break out of the loop. If the ending barrier has not been reached, then we simply append the current line to the data from the previous lines.

Once we reach the end of the attachment data, the final newline is removed from the data as this has the possibility of corrupting the attachment data. The attachment content data is then pushed to an array where each index contains the contents of an attachment with the corresponding filename pushed to a separate array. After each attachment has been parsed, these two arrays are returned back to the calling function.

Input: Secret data s
Output: Randomness r
$r \leftarrow \varnothing$
$n \leftarrow s.length \;/\; 65,536$
for $i \leftarrow 1 \ldots n$ **do** $r \leftarrow r \cap$ getRandomValues(65,536)
$r \leftarrow r \cap$ getRandomValues($s.length$ **mod** 65,536)

Figure 1: Key generation algorithm for the (2,2)-additive scheme to work around the 65,536 byte limitation of `getRandomValues()`.

The body of the current message is then parsed to find which secret sharing scheme was used to generate the shares. The detected scheme is then used to reconstruct the message using the parsed shares, keys, and tags as appropriate.

After reconstruction, the attachments and e-mail contents need to be separated. The attachment header information is used to get information on the attachments in the message. The e-mail content is found by looking for the first occurrence of the null character. The remaining attachment data is parsed using the size attributes stored in the header information.

After the attachment parsing, a custom window is opened with a message area that displays the reconstructed message along with an area for attachment files to be downloaded from. If there was an error somewhere along the reconstruction, the area will instead be populated with the error message (e.g., "Missing tag", etc.).

5 CRYPTO DETAILS

We now explain how the different secret sharing schemes explained in Section 2.3 have been implemented and other details about the cryptography being used for these schemes.

Note that when sending a message, we first follow the steps outlined in Section 4.1. Similarly, we follow the steps in Section 4.2 when receiving a message.

5.1 (2, 2) Additive Secret Sharing

To first test the ability of the Thunderbird ecosystem to handle secret sharing schemes, we implemented a simple $(2,2)$-additive secret sharing scheme as described in Sec. 2.3.1.

Note that a $(2,2)$ scheme is the same as the one-time pad (Vernam, 1926) encryption scheme in which one share is the key and the other the ciphertext. Since the one-time pad scheme is well known, we use this terminology.

While the XOR operation is quite simple, the complexity of this implementation lies in the generation of the key with which to XOR the secret. To do this, an empty byte array is created of the same length as the secret byte data. To generate the random values for the key, the Crypto API's `getRandomValues()` function is called. However, this function will return a DOMException (quota exceeded error) if more than 65,536 bytes are asked to be generated at once. To work around this issue, we generate n sets of 65,536 bytes where $n = l/65,536$ is the result of the integer division of the length of the secret content l in bytes and the maximum number of bytes we can generate per function call. After those n sets, we generate the final remaining number of bytes equal to l **mod** $65,536$. This solution can be seen in Fig. 1.

Since the secret content is passed to the function as an array of bytes and the key is also of the same dataset, we can easily iterate through each byte and use the built-in XOR operation in JavaScript to compute the ciphertext. Once the ciphertext is computed, two files are created: one with the ciphertext byte data and one with the key byte data. Two compose windows are also created. If available, the secret email addresses from the user-selected address book are added as the recipients of these two messages. Additionally, the hex ID is saved to the subject line and the informative body text is added. Finally, the ciphertext and key byte files are added as attachments with one to each new compose window. These windows then display for the user to finalize and send off to the recipient.

The reconstruction for this scheme is straightforward. Once the user clicks the reconstruction button in the message view window, the add-on will search for another message in the user's inboxes with a matching hex subject identifier. If no other message is found, then the reconstruction halts. If the other message is found, then the attachments for the two messages are parsed from the raw message data. The add-on then takes the byte data from the two files and XOR's them together to produce the original secret message. This information if then parsed to check the header information for any initial attachments. If attachments are found, then they are saved in local storage. The reconstructed message view page will then open. If there were any attachments with the original message, then they will be available on this page for download alongside the original message content.

5.2 Shamir's Secret Sharing

The implementation of Shamir secret sharing scheme takes the e-mail content and creates n shares as determined by the scheme parameters selected by the user. The development time of this scheme is significantly improved through the use of the "secrets.js" library (Rempe, 2019) that is used to construct the shares and reconstruct the secret. To construct the shares, the secret is first required to be converted to a hex string. While it does come with a string to hex conversion algorithm, it was quite slow. Instead, a new algorithm was written that experimentally showed to be about 1.5 times faster and is also easier to follow.

The first stage of the construction follows the additive scheme. The user first clicks secret sharing button in the compose window and the e-mail content is combined into a byte string. The Shamir secret sharing algorithm is then given as input the content, the updated compose window details with the hex ID subject line and informative body text, the number of shares, and the threshold for reconstruction. The e-mail content is then converted from a byte string to a hex string. This hex string secret is passed to the "secrets.js" library along with the number of shares and reconstruction threshold to generate the shares.

The library first converts the hex string into a binary string that is prepended with a 1 as a marker to preserve the length of the message and then padded to a multiple of 128 bits. The default implementation of this padding appends a pre-generated 1,024-length string of zeros to the string and then uses the `slice()` method of the String object to reduce the length back to the desired padding. This method can be somewhat slow, so it is replaced by a single line command using the function `padStart()`, also from the String object. This resulted in a time improvement of up to 25% and is more noticeable with larger secret sizes.

The binary string is then read starting with the least significant bit, converting every byte into an integer and populating an array with the data. For each integer in the array, the library generates a polynomial of degree $k-1$ and then evaluates the polynomial at n locations using Horner's method. This results in n Shamir shares for each integer in the integer array.

The n shares for each integer are referred to as subshares. Each subshare is converted from a number to a binary string and padded to a multiple of eight bits. The i^{th} subshare is then prepended to the i^{th} share. At the conclusion of the share generation, each share is composed of n subshares in binary string format.

Since the library is designed to be configurable to different data representations, the constructed shares have two pieces of extra data prepended to them. First, the number of bits per integer is converted to a base 36 value. For this implementation, eight bits is always used so this will be a constant. This value

also limits the maximum number of shares to 2^8. The second piece of data is an identifying value. This is simply the index of the share over the range $[1,n]$. This value is converted to a two-character hex string as the maximum value would then be 255, covering the range of the maximum number of shares. The first three characters of the n shares are given by $801, 802, \cdots, 8xx$ where xx is the hex representation of n. The remainder of the share is the hex string conversion of the binary string share calculated previously.

The n shares are then returned to the main implementation. The hex strings are converted into byte arrays that are used to create a file for each share. The shares are then added as attachments to new compose windows. The original compose window is then closed and the user is able to send the shares to their selected intermediaries.

The reconstruction also begins much the same as the additive scheme. The user opens an e-mail and selects the reconstruction button. E-mails in any incoming folder with matching subject identifiers are collected. The same attachment parsing method as described in the additive scheme is then used to find the share data from the attachments. The resulting hex string shares are then passed as an array to the Shamir reconstruction algorithm.

The reconstruction algorithm calls the reconstruction function from the secrets.js library. The library first extracts the two pieces of extra information at the front of each of the shares in order to know how to convert the string into an integer array. The set of integer arrays are then placed in a matrix with each array representing a row. The matrix is then transposed such that the number of columns is equal to n and the number of rows is equal to the length of the integer arrays.

The Lagrange interpolation is then evaluated using each row of integers. Each result is then converted to a binary string, padded to a multiple of eight bits, and prepended to a string holding the results of all of the evaluations. The final binary string result is converted back into a hex string and returned to the main implementation. This hex string is converted into an ASCII string as the return value for the Shamir reconstruction algorithm.

5.3 Robust Secret Sharing

The robust secret sharing (RSS) implementation builds on top of the Shamir Secret Sharing scheme. When the user selects the option to construct the shares, the system generates the Shamir shares as in the previous scheme. Then, keys (k_{ij}) and tags (tag_{ij}) are generated for each pair of parties $i, j \in [n]$ where $i \neq j$ using the fast universal hashing algorithm `PolyQ32` as described in (Krovetz and Rogaway, 2001). Fortunately, the algorithm as defined in the paper can be mapped directly into JavaScript, so there are no translation or syntactical issues to work around. After the keys and tags and generated, the share for party i after the construction consists of s_i, $n-1$ keys $k_{ij}\ \forall i \neq j$, and $n-1$ tags $tag_{ij}\ \forall i \neq j$ for a total of $2n-1$ attachments.

The fast universal hashing function from (Krovetz and Rogaway, 2001) allows the system to use a small key size of 32-bits with a message in 32-bit blocks. This plays well with the Shamir implementation that generates shares whose lengths are multiples of 128 (this is customizable in the Shamir implementation, but is not changed for the purpose of this add-on). This allows the system to generate the keys using the `getRandomValues()` function and send the key and share directly to the `PolyQ32` function.

The function as described in (Krovetz and Rogaway, 2001) is directly implementable in the JavaScript implementation. Because the share is in binary format, the only addition to the function is converting 32-bit blocks of the share into integers to perform the calculations. The arithmetic of the function is computing modulo $2^{32}-5$, the largest prime number under 2^{32}. Since JavaScript can represent integers up to $2^{53}-1$ without needing other objects, these 32-bit calculations will not overflow and lose any accuracy.

The construction of `PolyQ32` is such that the collision probability increases with the length of the message. To improve the collision bounds, multiple keys in $\mathbb{Z}_{32}$ are used to generate multiple tags for each share. These tags are then concatenated before sending to generate a longer tag. In the key and tag attachment files, they are delimited by a newline to allow the reconstructor to parse them. For the reconstruction, each of the key-tag pairs must match for tag_{ij} to be accepted. If a majority of the tags for one share are accepted (at least $\frac{n}{2}-1$ are verified), then the share is accepted into the reconstruction. Otherwise, the share is not included in the reconstruction.

As with the other schemes, the reconstruction begins when the user opens an e-mail. The add-on first searches for any messages with a matching subject ID in any incoming folder. It then parses the attachments for all of the files. Since RSS requires multiple attachments (share, keys, and tags), the attachment parsing had to be reconfigured for this scheme to allow both the contents and the titles of the attachments to be retrieved. This allows the system to then parse the attachments with the knowledge that the attachment is

Table 1: Average share generation time in milliseconds over 500 iterations for each implemented scheme with secret sizes of 1 KB, 10 KB, 100 KB, and 500 KB.

Scheme	Secret Size				
	100 B	1 KB	10 KB	100 KB	500 KB
(2, 2) Additive	4.58	4.70	5.82	14.07	53.15
Shamir	6.68	13.74	83.23	728.24	3,527.35
Robust	8.00	17.09	114.58	1,113.62	6,025.45

Table 2: Average reconstruction time in milliseconds over 500 iterations for each implemented scheme with secret sizes of 1 KB, 10 KB, 100 KB, and 500 KB.

Scheme	Secret Size				
	100 B	1 KB	10 KB	100 KB	500 KB
(2, 2) Additive	52.67	54.78	56.68	74.86	179.41
Shamir	55.57	64.03	145.61	913.43	2,797.70
Robust	58.09	74.62	234.25	1,529.21	5,143.46

a share, tag, or key. The tags and keys are parsed into matrices where each index represents the list of keys and tags used to verify the shares. The tags for each key and share are then calculated and compared to the tags received from the e-mail attachments. If a majority of the tags match, then the share is added to the list of accepted shares.

Once all the accepted shares are found, they are sent to the Shamir decryption implementation as a part of the library in (Rempe, 2019) as described previously. The results are then sent back to the main program and stored in local storage. The reconstruction is also printed in the developer console to allow for debugging.

6 SIMULATION RESULTS

In this section, we present experimental results on the execution time of the add-on relative to the selected scheme and the size of the secret. We compare each of the three implemented schemes with secret sizes of 1 KB, 10 KB, 100 KB, and 1 MB. To remove some of the overhead caused by Thunderbird and focus more on the implementation, we remove output logging and new windows are not opened at the end of both the sharing and reconstruction phase. Each experiment is tested 500 times. During the experimentation, it was noted that around the 75th iteration during testing, the execution time would consistently slow down significantly. In an attempt to negate this behavior, the 500 experiments are split into 10 sets of 50 iterations. The simulations are executed using Thunderbird version 81.0b2.

The results in Table 1 shows the experimental results for generating shares using the three schemes for

Table 3: Average time spent only on share generation (excluding attachment parsing) in milliseconds over 500 iterations for each implemented scheme.

Scheme	Secret Size				
	100 B	1 KB	10 KB	100 KB	500 KB
(2, 2) Additive	0.03	0.10	0.74	6.18	30.38
Shamir	0.80	5.99	69.36	710.44	3,465.80
Robust	1.35	9.57	100.67	1,109.91	5,972

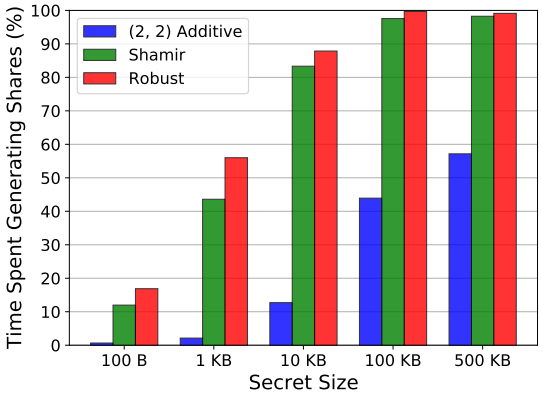
Table 4: Average time spent only on reconstruction (excluding content parsing, message querying, and attachment saving) in milliseconds over 500 iterations after removing message and attachment parsing and message querying.

Scheme	Secret Size				
	100 B	1 KB	10 KB	100 KB	500 KB
(2, 2) Additive	0.07	0.23	1.68	13.08	63.50
Shamir	0.45	3.09	27.43	338.82	2,072.65
Robust	0.87	5.97	57.29	714.61	4,881.69

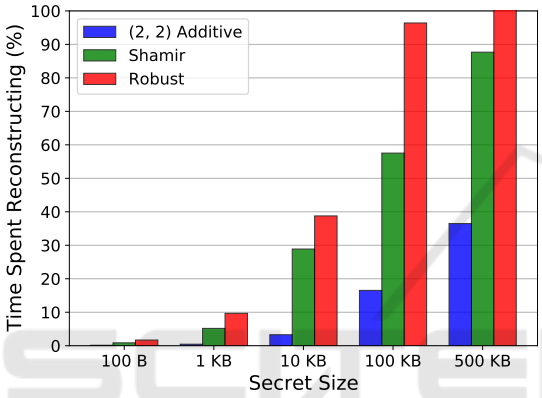
secret sizes of 100 B up to 500 KB. Table 2 shows the same results for reconstructing the shares generated from Table 1. These results show that the implementation has decent performance for lower secret sizes. Based on the percentage increase in the execution time compared to the percent increase in the secret sizes, it can be surmised that a large portion of the execution time for small secret sizes is composed of overhead caused by Thunderbird or the test system. Since Thunderbird is a single-threaded process, it is likely that other operations introduce delay into the secret sharing processing.

In addition, the execution times for larger secret sizes (in particular 100 KB and 500 KB) had significant variation between individual reconstructions. For the 500 KB test set, the individual execution times ranged from 4,400 milliseconds to 5,600 milliseconds. This can likely be explained as the result of Thunderbird frequently attempting to save drafted messages and query for new messages, causing some inconsistent overhead. The results for lower secret sizes also varied with a similar percentage of the overall time. It is important to note that this implementation has not been thoroughly optimized, so there is likely still some significant room for improvement in these execution times by improving the iteration and conversion operations.

In an attempt to isolate the reasons for the longer execution times for larger secrets, we also recorded the time for just the share generation and reconstruction without Thunderbird-related operations such as querying for matching messages, parsing the raw message contents, parsing the message body for the scheme information, and saving the files to local storage. This removes overhead that can not be improved



(a)



(b)

Figure 1: (a) The percentage of time spent on only share generation. (b) The percentage of time spent only on reconstruction. Both ignore message parsing, Thunderbird APIs, and attachment data saving in order to examine the overhead cost.

much given the current Thunderbird API and instead focuses on just the secret sharing implementations. The share generation execution time results for this experiment are recorded in Table 3 and the reconstruction execution time results are recorded in Table 4. These results are illustrated as a percentage of the total execution time in Fig. 1 for the three schemes.

These results show that, for small secret sizes around 1 KB and below, the Shamir and robust scheme reconstructions take less than 10% of the overall time. The remaining time is spent querying and parsing the messages and saving the attachments to local storage. At a secret size of 10 KB and above, the Shamir and robust scheme reconstructions take the majority of the time and reach 90% and greater around 500 KB.

Since the reconstruction has significantly less message parsing and file saving, the share generation has noticeably less overhead. The Shamir and robust scheme share generation accounts for around 50% of the total execution time with a 100 KB secret size. This value only decreases to around 10% for a 100 B secret.

7 CONCLUSIONS AND FUTURE WORK

The proposed private email communication system is implemented in the form of the Thunderbird add-on which currently supports 2-out-of-2 additive scheme, Shamir's scheme, as well as the Rabin and Ben-OR robust scheme (based on Shamir's sharing and fast universal hashing by Krovetz and Rogaway). The shares (and also keys, and tags in the robust scheme) are sent as attachments in multiple e-mails via email providers (that represent different parties in a secret sharing scheme). Each e-mail contains a unique 128-character hex string identifier (UID) to link them together during the reconstruction process.

This implementation has a distinct advantage over other forms of email security apps. Indeed, our Thunderbird add-on keeps the contents secret not only from the communication channel eavesdroppers, but also from the email hosts and servers who facilitate the delivery and receipt of the email. To guarantee this, the user needs to chooses not to send a threshold number of the shares through untrusted providers.

Another advantage of our add-on is that in current existing email security apps, such as PGP, the security relies on computational security, which remains unproven. This entails the need to perpetually evaluate the security of the cryptographic primitives and their security parameters (such as key length). The use of unconditional security by our implementation eliminates these concerns (provided the random generator provides true uniformly random independent of anything).

It is worth noting that our proposal can also be combined with the existing (computationally secure) application to enhance their security. For example, an email message can be both encrypted using PGP and then secret-shared using our implementation. Then, the adversary will have no information about the message unless (s)he is able to access the threshold number of shares. After that, (s)he would have to break the encryption scheme to finally access the message. A proper combination of unconditionally and computationally secure cryptographic primitives for the purposes of email security may be worth a further study.

The add-on is available via Github (git, 2022).

Future work could focus on speeding up the add-on, and on evaluating how good the random gener-

ation is. We now discuss the speeding up in more details.

One topic is focusing on optimizing the Shamir library to reduce the execution time. In addition, the message parsing needs to be optimized as currently it is required for the system to parse the entire raw message contents to get attachments on incoming messages. In the future, this may be made more simple through expanded WebExtension APIs. Currently, due to the recent major version change and add-on overhaul, the Thunder WebExtension APIs do not have any direct access to message attachments. Another potential route for optimization is selecting another secret sharing scheme that may be more optimized for a JavaScript implementation and for file sizes of up to a few megabytes. Finally, alternative universal hashing functions should be considered.

REFERENCES

(1993). A proposed federal information processing standard for an escrowed encryption standard (ees). Federal Register.

(1994). Department of justice briefing re escrowed encryption standard. Department of Commerce, Washington D.C.

(2021). "add-on developers." topicbox.com. [online]. https://thunderbird.topicbox.com/groups/addons.

(2021a). Thunderbird webextension apis - thunderbird webextensions latest documentation. readthedocs.io. [Online]. Available at https://thunderbird-webextensions.readthedocs.io/en/latest/.

(2021b). Web crypto api. Mozilla Developer Network [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API.

(2022). Proposed thunderbird add-on. [Online]. Available: https://github.com/NTSAS/Thunderbird-Secret-Sharing.

Afanassiev, V., Gehrmann, C., and Smeets, B. (1997). Fast message authentication using efficient polynomial evaluation. In Biham, E., editor, *Fast Software Encryption*, pages 190–204, Berlin, Heidelberg. Springer Berlin Heidelberg.

Attasena, V., D. J. and Harbi, N. (2017). Secret sharing for cloud data security: a survey. *The VLDB Journal*, 26:657—-681.

Bierbrauer, J., Johansson, T., Kabatianskii, G., and Smeets, B. (1994). On families of hash functions via geometric codes and concatenation. In Stinson, D. R., editor, *Advances in Cryptology — CRYPTO' 93*, pages 331–342, Berlin, Heidelberg. Springer Berlin Heidelberg.

Blakley, G. R. (1979). Safeguarding cryptographic keys. In *1979 International Workshop on Managing Requirements Knowledge (MARK)*, pages 313–318.

Carter, J. and Wegman, M. N. (1979). Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154.

Dolev, D., Dwork, C., Waarts, O., and Yung, M. (1993). Perfectly secure message transmission. *J. ACM*, 40(1):17–47.

Franklin, M. K. and Wright, R. N. (2000). Secure communication in minimal connectivity models. *Journal of Cryptology*, 13:9–30.

Johansson, T., Kabatianskii, G., and Smeets, B. (1994). On the relation between a-codes and codes correcting independent errors. In Helleseth, T., editor, *Advances in Cryptology — EUROCRYPT '93*, pages 1–11, Berlin, Heidelberg. Springer Berlin Heidelberg.

Krovetz, T. and Rogaway, P. (2001). Fast universal hashing with small keys and no preprocessing: The polyr construction. In Won, D., editor, *Information Security and Cryptology — ICISC 2000*, pages 73–89, Berlin, Heidelberg. Springer Berlin Heidelberg.

Liu, C. L. (1968). *Introduction to Combinatorial Mathematics*. Computer science series. McGraw-Hill.

Oren, Y. and Wool, A. (2009). Perfect privacy for webmail with secret sharing. Technical Report, Tel-Aviv University, Available at https://85c6e2e3-099c-4499-b7e5-046bb17abf53.filesusr.com/ugd/5dd4a3_318130cb05614ab58e275c1d5994247f.pdf.

Rabin, T. and Ben-Or, M. (1989). Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, page 73–85, New York, NY, USA. Association for Computing Machinery.

Rempe, G. (2019). Secret sharing for javascript. github.com. [Online] Available at https://github.com/grempe/secrets.js.

Shamir, A. (1979). How to share a secret. *Commun. ACM*, 22(11):612–613.

Simmons, G. J. (1990). Prepositioned shared secret and/or shared control schemes. In Quisquater, J.-J. and Vandewalle, J., editors, *Advances in Cryptology — EUROCRYPT '89*, pages 436–467, Berlin, Heidelberg. Springer Berlin Heidelberg.

Tompa, M. and Woll, H. (1987). How to share a secret with cheaters. In *Proceedings on Advances in Cryptology—CRYPTO '86*, page 261–265, Berlin, Heidelberg. Springer-Verlag.

Vernam, G. S. (1926). Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Transactions of the American Institute of Electrical Engineers*, XLV:295–301.

Wegman, M. N. and Carter, J. (1981). New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279.

Čuřík, P., Ploszek, R., and Zajac, P. (2022). Practical use of secret sharing for enhancing privacy in clouds. *Electronics*, 11(17).