SendSecure

Akash Badshah, Anurag Kashyap, Kenny Lam, Vikas Velagapudi

{akashbad, akashyap, kennylam, vvelaga}@mit.edu

The number of people utilizing webmail services is dramatically increasing, and more and more providers like Gmail are utilizing this email data to help better target their advertising. The storage and searching of users' webmail is not new, yet very few mainstream webmail users have taken steps to protect their data from such their providers. Several browser extensions exist to encrypt emails sent through services like Gmail to combat this storage, but they have not gained widespread adoption perhaps because they target too technologically savvy of an audience. We propose an extension for Gmail which seamlessly integrates with the existing interface and makes it very easy for users to send encrypted emails using asymmetric encryption. We describe a security policy for such an extension and evaluate our implementation against those security goals. Finally we propose avenues for continued work on this extension to help encrypted email become more popular and easier to use for the general populace.

Introduction & Motivation

Gmail is a ubiquitous web and mobile email client. On June 28, 2012, Google reported that Gmail had approximately 425 million monthly active users [1]. However, problematically, Google has great incentive to store Gmail users' emails as plaintext. First, doing so enables Google to efficiently query over users' emails, a key feature of the Gmail service. In addition, doing so enables Google to generate revenue by efficiently serving targeted advertisements to its Gmail users; as Google itself writes: "[I]f you've recently received a lot of messages about photography or cameras, a deal from a local camera store might be interesting" [2]. Both the efficient querying and advertisement serving would be compromised if Google did not store its Gmail users' emails as plaintext.

While Google claims "no humans read your email" [2], it is reasonable for a Gmail user to not trust Google with the plaintext of his/her emails, especially if said emails feature sensitive content, since the user cannot monitor Google's internal activity. In addition, given the current political climate, it is reasonable for Gmail users to be suspicious of third-party organizations like the NSA requesting access to Google's servers and thus to not want their emails available in plaintext to said organizations. That being said, such a user may desire to continue using Gmail for its more beneficial features (the user interface, the hierarchical organization of emails, etc.)

So in this paper, we describe SendSecure, a Google Chrome browser extension that enables Gmail users to continue sending emails through Gmail without Google ever storing the plaintext of said emails on its servers. SendSecure accomplishes this by performing public-key

encryption on emails users intend to send before the emails are actually sent through the Gmail service. The recipients of the email use the corresponding public-key decryption protocol to read the sent emails, but the plaintext of the emails are never stored on Google's servers, just the ciphertexts resulting from performing encryption on said emails. SendSecure does all this in a way that is user-friendly and consistent with Gmail's status quo workflow, thereby minimizing the friction preventing existing Gmail users from having a more secure experience with the service.

Prior Work

It is worth discussing three tools built previously that attempt to provide webmail encryption: LavaBit, Mymail-Crypt, and Mailvelope. These tools shaped how we tried to differentiate SendSecure and how we intend to potentially extend SendSecure in the future.

LavaBit

LavaBit was an email service, founded in 2004 [3], which "encrypt[s] user messages using their public key before writing them to disk" [4]. Thus, while LavaBit acted as the medium through which its users could send emails, LavaBit itself never saw the plaintext of those emails and never stored the plaintext of those emails on its servers. So Gmail users who do not trust Google could turn to LavaBit as an alternative, but (1) LavaBit is now defunct as it was shut down in 2013 [3] and (2) even if LavaBit were not shut down, it would be inconvenient for a user to migrate all of his/her emails to the LavaBit service. Hence, a tool that enabled a secure email experience on top of Gmail is needed.

Mymail-Crypt

Mymail-Crypt is one such tool that attempts to enable a secure email experience on top of Gmail [5]. It leverages the OpenPGP.js library to perform OpenPGP-compliant, public-key encryption on emails Gmail users intend to send before the emails are actually sent through Gmail. Recipients of said emails use Mymail-Crypt to perform the corresponding decryption, and so the plaintexts of the emails are never stored on Google's servers. But while Mymail-Crypt captures some of the functionality we desire, it still has several flaws.

First, Mymail-Crypt stores the plaintext of a user's private key on the user's browser's local storage, a simple key-value store. If an adversary managed to get possession of the user's machine, the adversary could then trivially obtain the user's private key. Moreover, even if the adversary does not get possession of the user's machine, the adversary could attempt a cross-site scripting attack, injecting JavaScript into the user's browser that read the user's private key and forwarded it to the adversary. With this private key, if the adversary ever obtained access to the user's Gmail account, the adversary could easily read all of the user's Mymail-Crypt-encrypted emails.

Second, a Mymail-Crypt user needs to manually upload the public keys of his/her intended recipients in order for Mymail-Crypt to encrypt the user's emails for said recipients. This is an extra burden on the user, potentially preventing Gmail users from adopting the encryption experience Mymail-Crypt provides.

Third, Mymail-Crypt's user interface differs substantially from Gmail's current interface. In particular, the buttons Mymail-Crypt provides its users to encrypt/decrypt emails differ in

appearance from Gmail's "Send" button (see Figure 1: Mymail-Crypt's encrypt buttons and Figure 2: Gmail's "Send" button). Such a difference detracts from a coherent user experience, potentially preventing Gmail users from adopting Mymail-Crypt's encryption service.

Figure 1: Mymail-Crypt's encrypt buttons

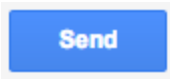

Figure 2: Gmail's "Send" button

Mailvelope

Mailvelope is another tool that attempts to enable a secure email experience on top of Gmail [6]. Like Mymail-Crypt, Mailvelope also uses OpenPGP.js to encrypt Gmail-sent emails so that Google never stores the plaintext of those emails on its servers. However, also like Mymail-Crypt, Mailvelope has its flaws.

In addition to the local storage and manual public key importation flaws also identified with Mymail-Crypt, Mailvelope suffers from a user interface that is markedly different from Gmail's current interface. For example, when a Mailvelope user decides to compose an encrypted email, Mailvelope pops up an external iframe within which the user composes his/her email (see Figure 3: Mailvelope's external email editor). This renders the Gmail experience anything but seamless, potentially preventing Gmail users from adopting Mailvelope's encryption service. (That being said, we do acknowledge the security advantage Mailvelope gains by using this iframe: it prevents Google code from learning anything about the composed email since the email is being written in isolation from the rest of Gmail's code. Preventing Google from adding code to Gmail that tracks its users' behavior (e.g., key-logging) to infer what the user is writing in his/her emails is not a problem SendSecure currently solves.)

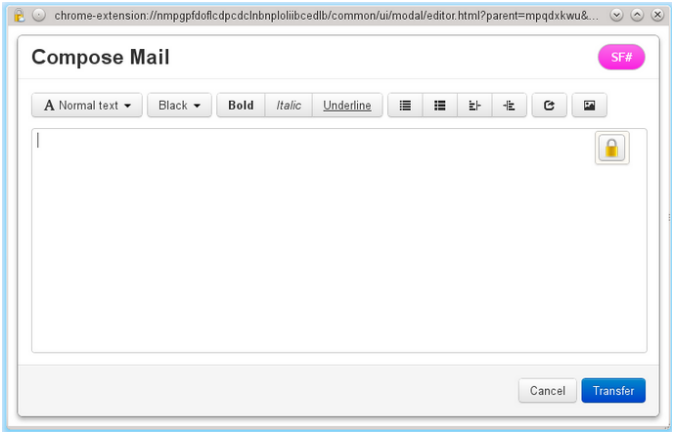

Figure 3: Mailvelope's external email editor

While each of these solutions has likely incurred some of its tradeoffs intentionally in order to make more firm security guarantees, each one in someway inhibits the user experience in a way that is likely prohibitive for most Gmail users. We built SendSecure to provide security for the non-technically savvy user and to encourage higher rates of adoption, under the assumption that more users encrypting their email would be better than a few using more secure methods. Overall we aim for SendSecure to be an email encryption tool on top of Gmail that minimally detracts from the current Gmail experience and workflow.

Security Policy

In designing SendSecure, we aim to provide the users (Gmail and SendSecure users) with email confidentiality against two main adversaries, Google and a "wire tapper." Therefore, our principals are:

- **Users**: Gmail webmail user with the SendSecure extension installed
- **Google:** the email provider
- **Wire-tapper:** some party with the ability to listen to network traffic

The primary objects of interest in our policy are the emails being sent between users, the Gmail client, and the SendSecure extension.

- **Emails:** These are the electronic messages sent between users that exist in two forms, plaintext and encrypted, which will be referred to as plaintext and encrypted emails.
- **Gmail client**: This is the code that is sent from Google to the browser being used by the user. The code is executed to create the Gmail webmail client in which the user composes and reads emails.
- **SendSecure Chrome extension:** This is code we wrote to modify the Gmail client to enable the user to decrypt and encrypt emails.
- **SendSecure passphrase:** A passphrase set by the user.
- **SendSecure public and encrypted private keys:** These are the per user OpenPGP keys that are used to encrypt and decrypt emails. They are stored on the SendSecure server. The encrypted private key is encrypted using CFB block cipher that uses the user's SendSecure passphrase as the key. The encrypted private key is decrypted by the user in the Chrome extension before being used.

We assume that the following rules are in effect.

- Users can read and write emails to one another
- Users can use the SendSecure Chrome extension to encrypt or decrypt emails before sending
- Google can read or write all emails sent by users
- Google can read or modify the client code sent to users' browsers
- Wire-tappers can only read network traffic, which means they can read sent emails, the Gmail client code, and the SendSecure code.
- User knows the passphrase.
- Users, Google, and wire-tappers can read SendSecure public and encrypted private keys.

Design & Implementation

Given the existence of existing solutions for encrypting email in webmail clients, we wanted to design ours for a different target user: an average email user who may not understand cryptography but is interested in protecting their data from Google or those who might be able to access Gmail data. One of the key requirements of this user is for the extension to function in a way that requires little to no technical knowledge and is as seamless as writing a regular email. Our hope is that by making the setup and usage of the app very simple, and the design as familiar to the existing Gmail interface as possible, that we would be able to attract a wide variety of users who might find existing solutions too hard to use or too clunky.

Our extension was implemented in JavaScript as with all Chrome extensions, and most of the functionality takes place in a single JavaScript file which is injected into all pages on the *mail.google.com* domain. In addition, we included CSS to style the elements we inject into the page and use jQuery [7] to make manipulation of the page easier. Finally, we implemented a simple server using the micro-framework Flask [8] which essentially provides a REST interface to query the database where the keys are stored.

Setup

In order to use our service, the first step for any user is to generate a set of keys for asymmetric encryption. Many services require that users generate their own keys, but we felt that knowing what they are or how to do so was an unreasonable technical requirement. Instead we provided an interface where users could simply generate a key with the click of a button, and those users who felt more comfortable generating their own could do so with an additional option (see Figure 4: our registration page). We also required a passphrase from users in order to encrypt their private key. By doing so we were able to store both the public key and the encrypted private key for users in a database served by a simple server. Anyone is able to request the public key or encrypted private key of someone for whom they have an email address, but the private key would only be exposable if that person also had the users passphrase. By doing so we were able to save the user the potential vulnerability of storing the private key locally somewhere and also shield them from the potential complexity of handling keys. We generate the keys using the implementation of the OpenPGP standard in the OpenPGP.js library [9], which is also used for all of our encryption and decryption operations in the actual use of the application.

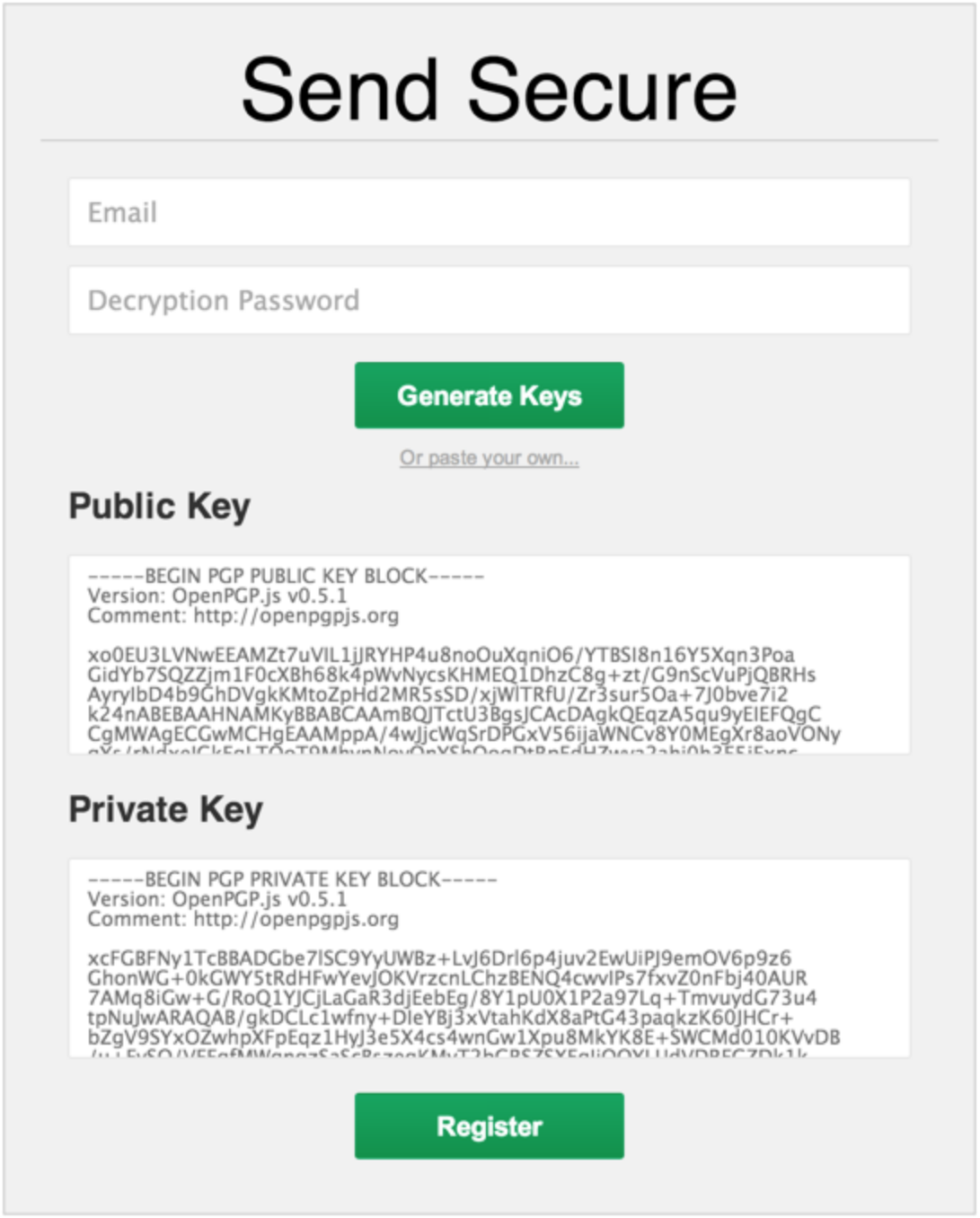

Figure 4: Our signup page, everything below the "Or paste your own…" line is hidden at first and drops down after the user hits the "Generate Keys" button or the link text

We also chose to style our signup page using the same colors, borders and general styles of Gmail to make the interface look crisp and familiar. We even went as far as to utilize the general button style that Gmail has for their "Send" button (as pictured in Figure 2), but using a different base color so as to encourage as much association with Gmail as possible. We initially hide the somewhat daunting Public and Private key blocks from the user and drop down the rest

of the form after they hit the “Generate Keys” button. This makes the signup process appear very easy and any user who does not wish to change their keys can simply complete the registration by hitting the register button. This sends both the public key, encrypted private key, and email to our server to be stored, and also triggers an automatic confirmation email which includes a unique link that the user must click in order for the keys to be usable. Until this link is clicked our database flags the keys and will not release them if requested. This additional email confirmation step serves as another authentication method to ensure our users have securely registered and not been compromised by an adversary.

Encryption

We wanted the process of encrypting the email to be no more effort than sending a normal email, and the UI to look as familiar as possible for users. To do so we took some inspiration from a popular Gmail extension [10] and added another row to the bottom of the Gmail compose windows, with a new button that had the text “Send Secure” (see Figure 5: A compose interface with our “Send Secure” appended). Our bar uses the same css styling as Gmail’s own bar, ensuring that it looks like a natural component of the interface. With Gmail’s new compose window approach, there might be multiple compose windows on screen at a time, and each one needs to have this extra row added. To do this elegantly, we followed an animation strategy [11] which involved identifying the class of these compose windows, and injecting CSS into the page which fires an event every time those windows are injected into the page. Our JavaScript listens to the event and injects additional html to show our extra row. We follow the same procedure for injecting the decryption UI.

When the user actually hits the Send Secure button, our injected javascript pulls all of the content from the compose box and acquires the recipient of the message. Using this recipient’s email address, it queries our server for their public key and use the OpenPGP.js library to encrypt the content with the public key of the recipient. Finally, we replace the content of the compose box with our encrypted text and trigger a send event on the email. By automatically doing all of those steps, we have made sending a secure email as simple as clicking one button, no more work than clicking the send button themselves. Also, we prevent the user from briefly seeing the encrypted text before they send it, which again makes the interface more seamless for novice users.

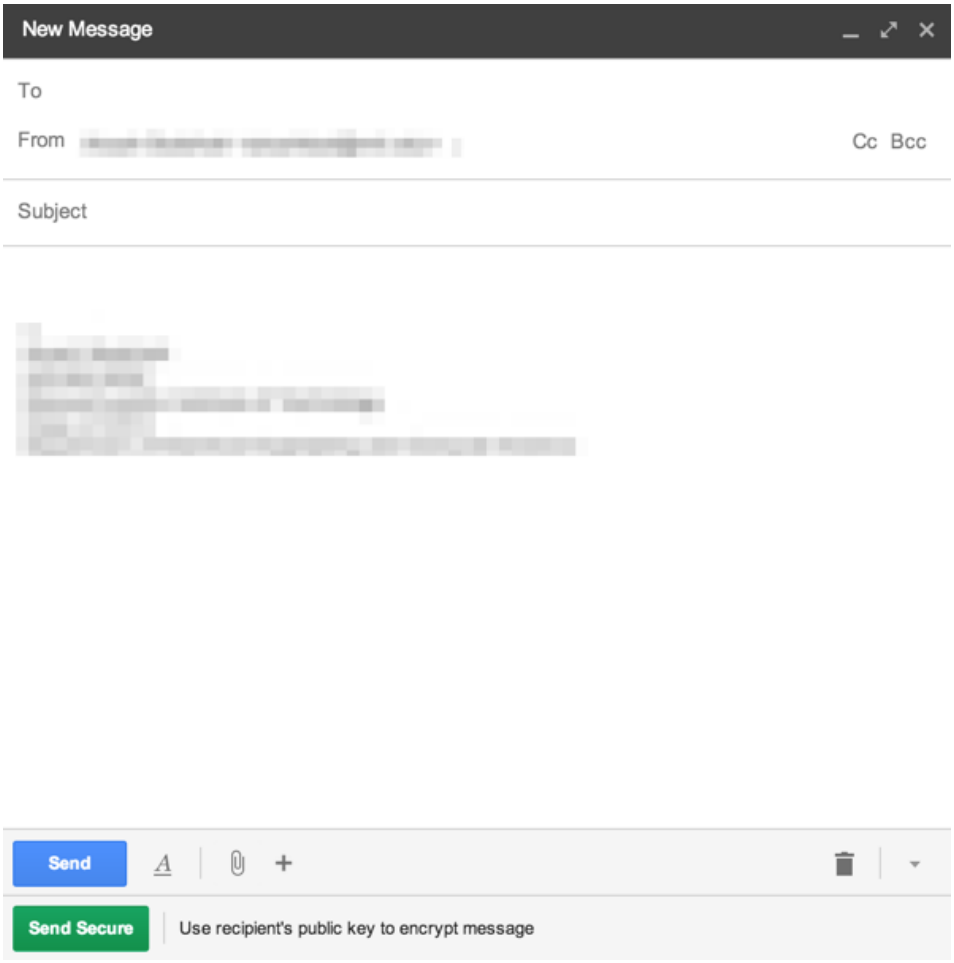

Figure 5: The Gmail compose window with our Send Secure row appended

Decryption

We follow a similar approach to decrypting the emails, inserting UI elements that look similar to the existing UI of Gmail and borrowing from their css styling (see Figure 6: A Gmail message with our decryption UI injected). Again we strived for simplicity and in particular for a set of elements that would be non-intrusive if the user was not reading an encrypted message. In order to decrypt the message, we need the passphrase the user originally used when storing their private key on the server, so we provide an input box for them next to the button. When a user enters their passphrase and clicks the decryption button, our javascript grabs the content of the message, as well as the address to which the email was sent. We send a request to our server to get the encrypted private key for the users email address and then decrypt the key on the client side using the passphrase. Finally we use the OpenPGP.js library to decrypt the content and then reinject it back into the message.

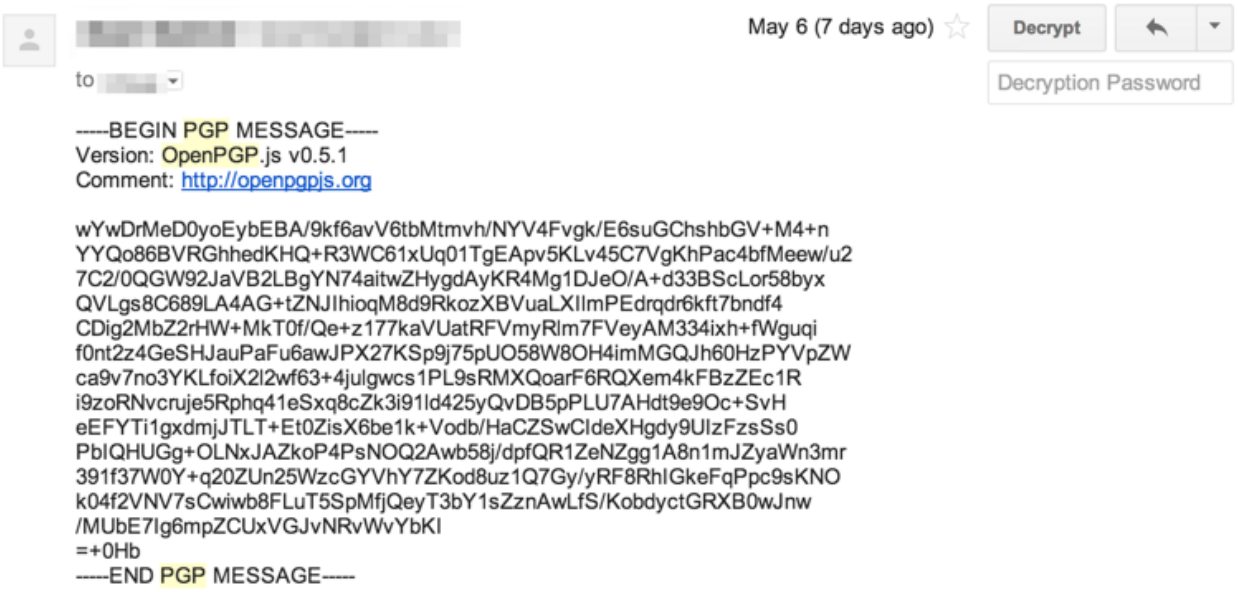

Figure 6: A Gmail message with injected decryption interface in the top left

Security of Design

SendSecure is built on top of OpenPGP.js and assumes OpenPGP.js correctly implements the OpenPGP specification [12]. Given a correct implementation of OpenPGP, SendSecure uses public key encryption to encrypt email. With public key encryption using 2048-bit keys, it is computationally infeasible for an adversary to figure out the plaintext of an encrypted email. Similarly given the correct implementation OpenPGP, it is computationally infeasible for an adversary to compute the private key from the encrypted private key without the SendSecure passphrase.

Under our security policy, Google has write access to the webmail client code. As such, Google may choose to modify the code in such a way that it prevents or worse circumvents SendSecure's encryption. Currently, the SendSecure code executes in the same environment as the Google webmail client code. This means that during runtime the user's unencrypted private key is available to Google's code. Google could specifically engineer their webmail client code to look for SendSecure and extract the private key from the runtime. Additionally, Google could avoid the encryption all together and just log key presses on the website to rebuild user messages. However, while Google is an adversary and we would like to prevent it's reading of unencrypted user emails as much as possible, SendSecure's goal is to make encryption easy and more accessible to users. Any Gmail user with serious confidentiality concerns will choose to use an email client with encryption that runs natively on their computer.

One way we could easily resolve this issue of Google having access to the same runtime environment as SendSecure is by using an iframe. The iframe would execute the SendSecure cryptographic code in an isolated environment and Google would be unable to access the unencrypted private key. However, we chose not to do this to provide better usability to the user. We decided the tradeoff here in terms of getting users to adopt SendSecure and use some level

of encryption in their emails is worth the risk of Google going to great lengths to steal every users' unencrypted private key, decrypt and read each email.

Wire-tappers, on the other hand, do not have have access to the runtime environment of SendSecure and the Gmail webmail client. Therefore, they only have access to a user's encrypted private key. As stated earlier, it is computationally infeasible for the wire-tapper to compute a user's private key from the encrypted private key. Additionally, it is computationally infeasible for the wire-tapper to discover the plaintext of an encrypted email without the private key. So we know the user's emails is guaranteed confidentiality from wire-tappers.

Future Work

There is a rich area of future work available for SendSecure and its application within the Gmail environment; though we currently support the encryption of email text, many of the surrounding features can also be masked to help with user privacy. Chat and attachments could also be handled by SendSecure, moving away from such strict reliance on Google. Further build-out of infrastructure is also necessary if SendSecure would be continued to scale and provide proper service. Lastly, we consider usability improvements which may be of utility to the user.

Another feature which is common within the Gmail interface is to chat with other Gmail users or Google+ users. As with email, the chats are stored at the Google servers in plain-text, and we believe that Google has no motivation to encrypt these messages either. In their communications, Google has set forth that your chats are subject to search just as much as as your email, and evidence has shown that chats are indeed read by Google [13,14]. However, we foresee several major obstacles which could arise in implementing a seamless chat interface for the user where they can ignore the encryption and decryption: the potential bottleneck of performance given our library, presenting a responsive interface to a user, and the correct handling of events and elements within the page. We have not experienced any issues with email, but the natural pace of a chat requires a much more rapid processing of messages, and we are unaware of any benchmarks or any performance tests on OpenPGP.js at the time of writing. The potential performance implications lead us to the next point of presenting a responsive interface to users. A solution to the performance bottlenecks would be to batch messages, but this provides a non-responsive chat for the user. The issue of correct handling of elements and events on the page relevant to chat is non-trivial. On receipt of any chat, the chat interface is automatically populates an element with the chat's contents. In order to provide for an interface where the user has no notion of the underlying encryption, the extension would need to intercept the event, stop it, decrypt the ciphertext, and re-fire the event with the newly decrypted plain-text. An alternative would be to hide all chat elements from Google and replace them with custom, mirrored elements which receive custom event firings. This would be the equivalent of masking all Gmail chat elements and instead recreating them. Lastly, users may also wish to view their chat history, which may present a combination of all three aforementioned problems when scrolling through many messages.

Our system also does not currently support encryption of attachments. Ideally, a purely client-side encryption of attachments would occur, and the extension would read a file, encrypt a version of it, and upload that version to the Google servers. The File API provided by modern

HTML5 browsers provides what we need, as it now supports the ability to read local files as a series of bytes [15]. However, we are unclear as to why some services only support specific types of attachments, but not others. Services such as docTrackr also claim to provide for revocation of distributed documents or even the option to remotely destroy documents [16]. We are unclear on how exactly this is possible or performed after a document has already been distributed, or if this is a claim which is only limited to documents docTrackr controls. The latter is not applicable since we never intend to host documents directly. We could also try to integrate with a different trusted third party to host documents separately from Gmail, such as Dropbox. Such a system could intercept attachments and upload them to the receiving user's Dropbox storage, and we believe such functionality is possible with Dropbox's API given [17].

We also do not currently provide strong server support for a number of features. Rate limiting can help to detect if there is one malicious user or a number of users seeking to scrape our database of encrypted private keys. Better logging functionality would also help with this issue, so we could better detect if a user is requesting data on users with whom they have never communicated. Additionally, the server currently acts as a single point of failure, inhibiting utility if unavailable; the move to a more distributed service would help to mitigate the effect of a DDoS attack or server failures. We also do not check the origin of a request. Again, we are primarily concerned with the requests for an encrypted version of a private key, and not with requests for public keys. The referer header could be confirmed whenever a request enters our system, though we consider that header forgery is very possible [18]. Instead, an alternative verification should be used, such as OAuth or OpenID, both of which Google supports.

Another common task in the inbox is to perform a search for key terms in the body of an email. Given our encryption scheme, where the entire body of an email is encrypted, no partial information within the messages can be recovered without decrypting an entire email. This is problematic for users who wish to perform a search over their entire inbox, which may be very large in the number of messages, and again we encounter a potential performance bottleneck if we were to decrypt all messages. Our solution to this problem would be to implement something similar to inverse document frequency at the email level, and append plain-text tags outside of the ASCII-armor surrounding the encrypted messages [19]. Alternatively, users can be allowed to tag their own messages for later searching, with plain-text tags which are meaningful to them. These presents a leak of information with regard to the email, but provides a significant increase in the usability of their inbox.

Integration onto more platforms is also a viable area of exploration. Firefox functionality should mimic that which is provided in the Chrome extension, and we do not anticipate any major drawbacks with a Firefox extension for the current feature set. Internet Explorer, however, may require porting the code to C# instead of its existing JavaScript embodiment [20]. Primarily, the growing ubiquity of mobile motivates us to move to support the Android and iOS platforms. To our knowledge, the aforementioned extensions do not support the mobile environment, so emails remain encrypted when viewing through the Gmail mobile application or using any browser on mobile. Services such as LastPass offer an interface through which passwords from a LastPass account can be input into password fields on other apps through the presentation of an overlay, so we anticipate that a SendSecure app could provide a similar interaction where the plain-text of an email can be overlaid, given the proper permissions [21].

More research is necessary in cross-application permissions to provide a firm conclusion of whether such an app is possible, though, especially on iOS, which has more restrictive permissions [22].

A number of usability improvements within the extension can also be made. Currently, encryption is a one-way operation, so users who want to edit their encrypted message have to restart from scratch. The text could simply be held in memory and cleared upon the send of the email. SendSecure also parses out HTML tags right now, which strips all formatting out of messages, which is likely not desired. Embedded images are also removed right now, since they exist as strictly HTML elements.

Conclusion

We provide the SendSecure extension for the Chrome browser in this paper. It provides for a layer of privacy within the Gmail environment by encrypting messages to be stored on Google's servers. Our motivations for this project primarily revolve around the want for extended privacy away from Google's eyes, while still utilizing such a ubiquitous service. Though other services exist for providing similar functionality, we believe we have lowered the barrier of entry for users, which may help more widespread adoption of the service. Lastly, we consider many potential avenues for increased privacy within the Gmail environment.

Bibliography

[1] Sean Ludwig. "Gmail finally blows past Hotmail to become the world's largest email service." VentureBeat. June 28, 2012. http://venturebeat.com/2012/06/28/gmail-hotmail-yahoo-email-users/

[2] "Ads in Gmail." https://support.google.com/mail/answer/6603?hl=en.

[3] Joe Mullin. "Lavabit founder, under gag order, speaks out about shutdown decision." Ars Technica. August 13, 2013.

[4] "In Memoriam: Lavabit Architecture - Creating A Scalable Email Service." High Scalability. http://highscalability.com/blog/2013/8/13/in-memoriam-lavabit-architecture-creating-a-scalable-email-s.html

[5] Mymail-Crypt Chrome Extension page on Chrome Web Store. https://chrome.google.com/webstore/detail/mymail-crypt-for-gmail/jcaobjhdnlpmopmjhijplpjhlplfkhba?hl=en-US

[6] Mailvelope. https://www.mailvelope.com/

[7] jQuery. http://jquery.com/

[8] Flask. http://flask.pocoo.org/

[9] OpenPGP.js. http://openpgpjs.org/

[10] Boomerang. http://www.boomeranggmail.com/

[11] David Walsh. "Detect DOM Node Insertions with Javascript and CSS Animations". http://davidwalsh.name/detect-node-insertion

[12] OpenPGP Message Format. http://www.ietf.org/rfc/rfc4880.txt November 2007

[13] Cecelia Kang. "Google announces privacy changes across products; users can't opt out." Washington Post. January 24, 2012.

http://www.washingtonpost.com/business/technology/google-tracks-consumers-across-products-users-cant-opt-out/2012/01/24/gIQArgJHOQ_story.html
[14] Adrian Chen. “GCreep: Google Engineer Stalked Teens, Spied on Chats (Updated).” Gawker. September 14, 2010. http://gawker.com/5637234/gcreep-google-engineer-stalked-teens-spied-on-chats
[15] Eric Bidelman. “Reading files in JavaScript using the File APIs.” HTML5 Rocks.June 18, 2010. http://www.html5rocks.com/en/tutorials/file/dndfiles/
[16] docTrackr Plans & Pricing. http://www.doctrackr.com/document-control/pricing/
[17] Alex Wilhelm. “Ooberdocs Takes Incoming Email Attachments And Copies Them Into Your Dropbox Account.” TechCrunch. May 4, 2014. http://techcrunch.com/2014/05/04/ooberdocs-takes-incoming-email-attachments-and-copies-them-into-your-dropbox-account/
[18] HTTP referer. http://en.wikipedia.org/wiki/HTTP_referer
[19] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze. Introduction to Information Retrieval. 2008. http://nlp.stanford.edu/IR-book/html/htmledition/inverse-document-frequency-1.html
[20] “Developing Internet Explorer Extensions?” Stackoverflow. Aug 9, 2012. http://stackoverflow.com/questions/5643819/developing-internet-explorer-extensions
[21] “Logging Into Android Apps Just Got Easier.” LastPass Blog. March 25, 2014. http://blog.lastpass.com/2014/03/logging-into-android-apps-just-got.html
[22] Todd Jackson. “Why is Cover Android only?” November 6, 2013. https://cover.zendesk.com/hc/en-us/articles/200795937-Why-is-Cover-Android-only-