

4-2014

SEED: Searching Encrypted Email Dependably. A design specification for secured webmail.

Alessandro Roux
College of William and Mary

Follow this and additional works at: <https://scholarworks.wm.edu/honorsthesis>



Part of the [Information Security Commons](#), [Software Engineering Commons](#), and the [Systems Architecture Commons](#)

Recommended Citation

Roux, Alessandro, "SEED: Searching Encrypted Email Dependably. A design specification for secured webmail." (2014). *Undergraduate Honors Theses*. Paper 110.
<https://scholarworks.wm.edu/honorsthesis/110>

This Honors Thesis is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Undergraduate Honors Theses by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

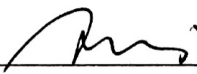
SEED: Searching Encrypted Email Dependably
A design specification for secured webmail

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelors of Science in Computer Science from
The College of William and Mary

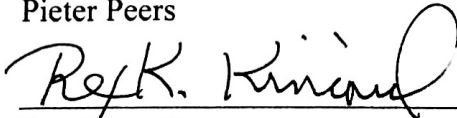
by

Alessandro Roux

Accepted for Honors
(Honors, High Honors, Highest Honors)


Qun Li, Director


Pieter Peers


Rex Kincaid

Williamsburg, VA
April 24, 2014

SEED: Searching Encrypted Email Dependably

A design specification for secured webmail

Alessandro Roux

Advisor: Dr. Qun Li

The College of William & Mary

Abstract

Webmail services are a convenient, internet-based access point for email management. A webmail user must trust the service provider to honor the user's individual privacy while accommodating their email contents. Webmail users are increasingly conscious of the risk to their privacy as many webmail services have fallen victim to cyberattacks where unwanted observers have exploited server vulnerabilities to steal user private data. The relationship of trust between webmail provider and webmail user has been further called into question with the reveal of NSA snooping of user email, often with the tacit approval of the webmail provider.

We augment a modern webmail service with end-to-end encryption of user email data. Our system, SEED, is designed to respect the original functionality of the webmail service. Most notably, we enable search of encrypted message bodies using the webmail service's built-in search engine. With an ancillary web browser extension called SEED add-on, the user is able to manage email in the webmail client while decrypting sensitive email information in a separate local process. The browser extension manages the user's encryption keys and decrypts email ciphertext automatically such that the user remains ignorant of the underlying cryptographic implementation as they browse their email.

Built upon Gmail, SEED stores a user's email data on Google's remote servers and guarantees that Google is unable to interpret it. When managing their email, the user still enjoys the full capabilities of the Gmail web client, including composing, reading, and robustly searching email by message metadata. The user is able to do all of this without revealing their usage habits to Google. The user is able to do all of this without revealing their emails to Google.

Using SEED, the user benefits from the conveniences of webmail and preserves the integrity of their private information stored online.

Acknowledgements

I would like to thank my advisor, Dr. Qun Li, for his encouragement and guidance throughout the span of the project. I grew as a programmer and critical thinker by watching him work during our many meetings. I thank Ed Novak for mentoring me on SEED. Besides helping me tackle the project's numerous details, he was an always valuable perspective on computer science and research life.

I appreciate the instructive feedback my committee members, Dr. Rex Kincaid and Dr. Pieter Peers, provided during my thesis defense. I am thankful to the both of them for their participation. This project was partially supported by the US National Science Foundation grants CNS-1320453 and CNS-1117412.

I am immeasurably grateful to my family and friends for their patience and support. Their invigorating good humor and timely coffee breaks fueled me throughout my final year at the college.

Contents

1	Introduction	5
1.1	Why we build SEED around Gmail	7
1.2	Contributions and paper overview	8
2	Related Works	9
3	System Overview	10
3.1	User configuration	10
3.2	Writing an email	12
3.3	Sending email	12
3.4	Receiving email	13
3.5	Searching email	13
4	System Design	13
4.1	Firefox add-on: “SEED add-on”	13
4.1.1	Our encryption scheme, “sjcl”, and “crypt”	17
4.1.2	Executing SEED add-on: the “main” process	20
4.1.3	inbox-reader	20
4.1.4	message-reader	21
4.1.5	searchbar-handler	21
4.1.6	compose-handler	23
4.2	The SEED server	23
4.2.1	Proxy mail {transfer, delivery} agent	23
4.2.2	Key Management Service and Mail Archival Agent	24
5	Evaluation	25
5.1	Methodology	25
5.2	Message size	26
5.3	Encryption and decryption performance	27
5.4	Search performance	28
5.5	Search accuracy	29
6	Conclusion	30
7	References	30

1 Introduction

Webmail is “[email] that is available for use online and stored in the Internet server mailbox, and that is not downloaded to an [email] program or used offline.” [34] Essentially, webmail relocates email off of a user’s personal machine and onto an off-site server curated by a third party. The webmail user benefits from storing and accessing email remotely. They are no longer responsible for their email data’s integrity but only for its content. The webmail provider invests in data protection for their storage infrastructure such that users are assured that their email data is always intact and accessible. The benefit to the user of a cloud “software as a service” approach is that they access a fully-featured email client without having to implement the application infrastructure themselves. They also avoid shouldering the high costs of server maintenance. In exchange, the webmail provider either charges the email user a fee for using their service or monetizes the information users store on their servers to generate revenue. The latter revenue source is the standard business model among modern webmail companies. A cloud company offers its “software as a service,” asking for the users to tolerate message scans and information parsing in exchange for free and unlimited access to their product.

In a sense, a user agreement with a cloud service company is similar to an agreement a person makes with a bank when trying to safeguard their money. They offer the bank a source of money that the bank can turn around and offer other customers as loans, profiting off the interest charged on these loans. A modern webmail provider stores and safeguards user email data under the assurance that no private information from user email is viewable by anybody but the user. The webmail provider then leverages their large store of email information, in which users reveal their private preferences, to cater advertisements that align with the specific interest of the webmail user. Webmail providers do this without breaking the privacy agreement they make with the user, utilizing automated parsing bots to process user information anonymously.

Therein lies the problem with modern webmail services. The user gains from using a webmail provider’s seemingly free webmail service, however they implicitly give valuable personal information to the webmail service under a vow of good faith that the information will never be employed maliciously, clandestinely, or illicitly. Unlike a bank, there is no government regulation of cloud service providers [29]. If user information is misappropriated, the legal recourse a user has is as ambiguous as the phrasing of the user agreements they originally consent to when signing up for webmail. The details about a user’s personal life that can be inferred from email metadata are staggering [30] and could be misused to the user’s detriment.

It has been shown that webmail and cloud service providers do not treat all private user data as sacrosanct. The recent revelation of the NSA mass surveillance apparatus and the tacit participation of cloud service providers in said snooping have undermined the public trust of the third parties that handle their private data, webmail included [33]. The public has grown skeptical of the efficacy of cloud service’s server security as company after company has fallen victim to criminal hacking. We

are reminded of the febleness of server infrastructure when “internet-breaking” vulnerabilities, such as Heartbleed [26], force internet users to rebuild their online security measures (e.g. changing their passwords, using two-factor authentication, etc.) in a mad-scramble to re-secure their private identity online.

Webmail providers try to counteract malicious attackers by encrypting the user data on their servers and building massive security complexes around their data centers. Unfortunately, latent bugs in server infrastructure, like Heartbleed, continue to be difficult to detect and difficult to prevent. In either case, the webmail user has no definitive assurance that the webmail provider takes sufficient precautions to protect the user’s data. Even if such precautions are stringent and appropriate, user data is always vulnerable to attacks from inside the webmail provider, such as those that can be performed by a rogue employee or, at worst, a deliberate policy of data exploitation mandated by the hosting corporation itself.

A user can forego a third-party webmail service and construct their own encrypted mail server. The requirements for maintaining such a server would make it an inconvenient and expensive solution for personal email, as well as one without the features of a modern webmail provider. The webmail user can use any encryption algorithm to conceal their message bodies, but naïve encryption often breaks useful webmail features (e.g. search) and requires the sender and recipient to manage encryption keys. Even with encrypted bodies, email identification metadata remain in plaintext in order for the message to be routed. Exposing this metadata reveals much about the nature of the email correspondence and is therefore unacceptable. How, then, are we to assure the user that their email is completely safeguarded while offering them definitive proof that they are the sole custodians of their data?

We propose an end-to-end encryption system for webmail. We create a system called SEED, built specifically for Gmail, that ensures that email data is only viewable by the webmail user at any given time. We call our system SEED as a thinly-veiled statement of purpose. The primary challenge when developing an feature-rich webmail system is to support search. We set out to make encrypted email search as robust as regular email search. To us, “Searching Encrypted Email Dependably” implies searching email contents accurately and quickly, yielding results that are correct and useful to the webmail user.

Our underlying motivation with SEED is to redefine the user’s expectation of privacy while online. We hope to catalyze a shift in perspective where the webmail user understands that conceding personal security is not a necessary requirement for benefiting from internet-based services. With SEED, we counteract the public’s resigned stance on securing privacy online and show that it is possible to engineer functional solutions to modern problems of privacy preservation.

By building SEED around Gmail, we reap the benefits of Google’s elaborate server infrastructure and their data safeguards that ensure the integrity of the data stored on their servers. We support as many of Gmail’s features as possible in our system, including support for full search of encrypted email driven by Gmail’s built-in search engine. Our goal is to shift email data ownership back to the user by giving them the encryption keys necessary for managing and interpreting their email data. We remove all identifying characteristics from email messages such that Google is unable to identify individual

SEED users on their network and, more crucially, is unable to interpret the private email information we store on Gmail servers.

1.1 Why we build SEED around Gmail

Although SEED can be generalized to work on any popular webmail service, we implemented SEED with Gmail in mind. We did this because Google is at the forefront of cloud-based software services. Part of the appeal of Google’s software ecosystem is the tight integration between its products. Google offers a streamlined and ubiquitous hub for modern information services. It is the consummate example of an internet company that harnesses its vast amount of user behavioral data to create its product. Google’s product is a predictive web platform for search, news, weather, media, productivity, and navigation.

Gmail is the base of Google’s massive software complex. It is the most widely used email service in the world with a user base of nearly 500 million accounts. It is a robustly-featured yet intuitive web interface for email management. The Gmail website is praised for its responsive design, pioneering the modern wave of high-performance, desktop-quality web apps run in the browser. The web interface is accessible from any computer with an reasonably modern browser, in addition to all major mobile platforms. It allows instant access to email without the sense of a physical dependence on a mail server.

User experience. Gmail particularly supports features that facilitate email management. Such features include the sorting of email by folders and labels, intelligent and automatic filtering of messages according to importance, and the ability to quickly search inbox contents using message metadata. Google has built a web client with productivity in mind, integrating into their software various interface conveniences such as keyboard shortcuts and a floating window UI that allows users to perform parallel tasks while browsing (e.g. composing email in the foreground while searching the inbox for in the background pane). Any Gmail account comes with 15 GB of free storage, giving users more than sufficient space to store their email.

Server infrastructure. Gmail is built upon Google’s advanced server infrastructure that is synonymous in the industry with data stability (Google’s uptime is guaranteed to be 99.9% of the year [17]). Google has put into effect a sophisticated system of data organization that safeguards user data. Driven by the Google File System, which is implemented on a highly specialized and secure version of the Linux operating system, Google fragments its user data into “chunks” across hundreds of Google servers worldwide with no discernible internal organization. By distributing chunks of user data, as well as copying them several times for data redundancy, Google ensures that customer data is near impossible to lose, yet nimble to access [14]. File chunks are assigned random names so that specific file contents are not easily interpretable by system administrators. Google also encrypts all information stored on Google servers to prevent data from being human-interpretable. Google does not hold onto any information a user erases from their account. They have adopted extensive data destruction policies such

that no physical remnants of user data remains after erasure or hardware failure (e.g. a hard drive is decommissioned or a server rack malfunctions).

From a security perspective, data distribution also makes it near impossible for an attacker to gain physical access to all of a user's data at once without coordinating a multi data center attack. Even if possible, the hacker would need to be a formidable psychic in order to anticipate which data chunk belongs to which user's data. Gathering this level of insight would be difficult as only privileged master machines understand how an individual's information is scattered across servers. The master machine, in turn, provides file location information only to a server-authenticated client.

Data center employees. In order to work at Google data centers, Google employees have to undergo a rigorous honesty validation process. Access rights are strictly limited such that anyone that does not need access to data centers cannot enter one (this policy extends even to Google's CEO). A Google employee must undergo a complex process of approvals in order to enter into an area where Google servers are active. Even more intensive administrative oversight is required for any employee to directly interact with Google customer data. Data centers are physically guarded by extensive security measures, including a 24/7 security team, on-site surveillance, biometric access controls, and perimeter fencing [18, 16].

Conclusion. It is clear that building SEED on Gmail infrastructure is of huge benefit to our system. The precautions Google takes with customer data ensures that our system has a highly stable and cost-free storage source. Gmail's advanced email management features and massive user base make it the logical choice for a webmail provider around which to build SEED.

1.2 Contributions and paper overview

In this honors project, we make the following contributions:

- We outline the design of a system, SEED, that secures user data stored on a webmail provider's servers and that gives the user sole access to their data.
- We build a functional prototype of the SEED add-on Firefox add-on that interfaces with the Gmail web client. It supports most of Gmail's native features including full encrypted search capabilities.
- We evaluate our system prototype, showing that encrypted search can be accurately performed with an inconsequential effect on performance, that encryption of most messages takes less than 5 seconds, and that decryption of email rarely takes longer than one second.

We will outline past work in encrypted search and email in section 2. We discuss our system overview in section 3, including a full description of how SEED is used. In section 4, we detail the design decisions that informed the SEED add-on design and the organization of SEED server infrastructure. In section 5, we evaluate the performance of SEED and in section 6, we discuss our conclusions.

2 Related Works

PGP is among the older software solutions for encrypting email [32]. Built by Phil Zimmerman in 1991, PGP was designed to provide encryption and authentication for all manner of data. It was adopted as the de facto standard for email signing and encryption. It uses both asymmetric and symmetric encryption to secure message contents. Inherent to the appeal of PGP's scheme is the idea of a web of trust, where multiple parties can verify the integrity of one user's public key by signing it with their own key. PGP has since been adapted into the IETF OpenPGP standard, which is implemented throughout industry. A notable open-source implementation of the OpenPGP standard is GnuPG. PGP has also been extended into several email services. In general, these implementations do not support searching of email contents and, instead, favor encrypting message bodies whole-cloth.

There are several academic papers that are similar in nature to our system. gVault builds a mountable encrypted file system that uses Gmail as free, networked dumb storage [22]. In their work, they discuss their design for a webmail-based file structure that preserves data integrity while enforcing security with a purpose-made key management scheme. While we relate to their work in our desire to use Gmail as convenient storage, our system's goal is to augment Gmail's capabilities with user-controlled security. We deal with the specific challenges of how to search encrypted email and protect sender and receiver privacy.

Author Ramchandran et al. build the system Chaavi [23]. The authors implement an encrypted webmail system that supports encrypted search. It differs from our work in that we try to interface with an already existing webmail service provider. We face the challenge of having to integrate our system with the specific and closed-source limitations of the Gmail UI. We are unable to perform any email manipulation once email is submitted to Google servers. Our system supports the full gamut of Gmail features, whereas the authors must reimplement any high-level webmail feature in their Chaavi.

The authors Song et al. first established the problem of encrypted search in "Practical techniques for searches on encrypted data." [31] The authors identify the fundamental problem of wanting to keep data secure when it is stored off-site while avoiding the limitations on remote computation that encrypting this data imposes. They deduce a scheme for search on encrypted data and prove the scheme's integrity. Many papers have since improved on encrypted search. The papers focus on efficiency gains and utility improvements. [5, 10, 8, 9, 36, 24, 15, 25, 11, 12, 4, 1, 3, 6] Each of these papers presuppose that the client and server scheme are developed in tandem. While our system uses a server to coordinate message encryption for webmail, we do not direct the activities of the server storing our email data. We interact with the input and output of a black box webmail provider, encrypting any information that will be stored on or interpreted by it. Brinkman surveys the topic of encrypted search in his PhD thesis [7]. In it, he identifies three varieties of encrypted search. All three approaches require the system administrator to define the behavior of the client and the storing server. Our formulation of the encrypted search problem, where internal server activity is not known to the client, is apart from the three encrypted search methods

he highlights.

There are several projects in industry that implement some form of email encryption. SecureGmail [2] and Mailvelope [27] are browser extensions that incorporate encryption schemes into popular webmail services. Mailvelope uses asymmetric encryption via OpenPGP to encrypt email bodies while SecureGmail uses AES symmetric encryption to password-protect messages. Both extensions encrypt the body text of a message with no consideration for webmail features, like search, that rely on data coherence. In addition, each requires the user to actively manage encryption keys. In order to use Mailvelope, the user must understand the dynamics of public-key encryption to send protected data to a recipient. SecureGmail requires that the sender can discretely communicate the encryption password (used to symmetrically encrypt the message data) to the intended recipient. Our system attempts to build full message encryption into webmail without breaking search.

CipherCloud is a company that touts a method for full encryption of webmail and other cloud services. They support mobile applications and ensure that the full feature parity between the encrypted versions of cloud services and their normal variations. It is difficult to verify the quality of their solution as it is closed-source, commercial technology.

3 System Overview

In order to ensure that no part of Gmail's data flow remains in cleartext, we delegate the job of encrypting/decrypting webmail information between two SEED subsystems. The first is a Firefox add-on that interacts with the Gmail web client. We call this add-on the SEED add-on. SEED's add-on prevents any input into the Gmail Basic HTML web client from being submitted unencrypted to Google servers. It also decrypts encrypted emails when they are displayed in the Gmail web client. The second subsystem, the SEED server, ensures that any email traffic into or out of a Gmail account is properly encrypted. We build the SEED server as an intermediate email server so that we can receive email for an intended user, secure the messages, and reroute them into a user's Gmail account. The SEED server is responsible for user management by acting as the central store for user cryptographic keys. Key management is overseen by a server process we call the Key Management Service. The server also executes a process that we call the Email Archival Agent. The Archival Agent is used to overcome certain attacks against the SEED system.

3.1 User configuration

The first problem we encountered when designing SEED was how to encrypt email being sent to a Gmail address. Such a message would be routed directly to Google servers, leaving us no opportunity to pre-process the information. We decided that the only way to guarantee that we made first contact with received email was to ensure that it was sent to our intermediate server. Our intermediate server,

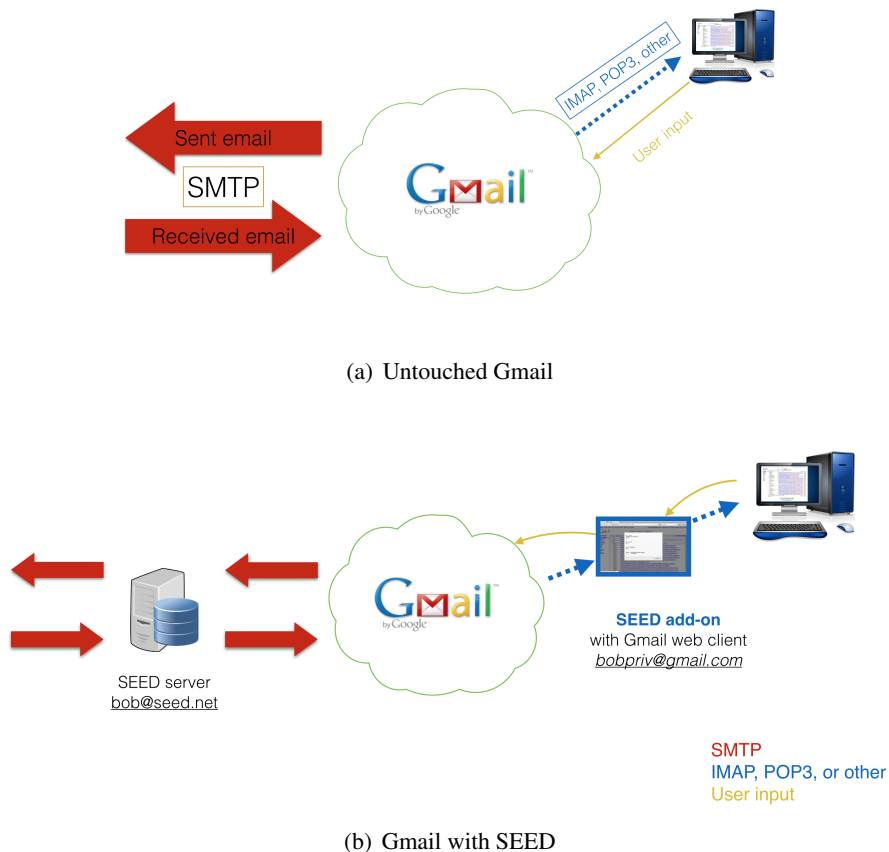


Figure 1: Tracing email data flow in Gmail and in Gmail reinforced with SEED.

the SEED server, could then perform the necessary cryptographic operations on the received messages before forwarding them to a Gmail account. A SEED user, therefore, must have two email addresses: one that is public at which they receive email; another that is private that the SEED user accesses for managing email. The public address will use the SEED server as the destination. The private address is a Gmail account such that the SEED user manages their email in the Gmail interface (Fig. 1).

Although having two email addresses introduces a level of complexity to our system that may confuse a user, one could think of it as analogous to possessing a public and private key in asymmetric encryption schemes. The SEED user would never have to deal with configuring their public address because it is completely managed by the server. The user only remembers the login information for their SEED-augmented webmail service of choice. For additional security, it is a good idea that the two addresses have different names so that email traffic between SEED and Gmail is hard to track. We emphasize the importance of concealing connections between SEED and Gmail when we present our security analysis of SEED.

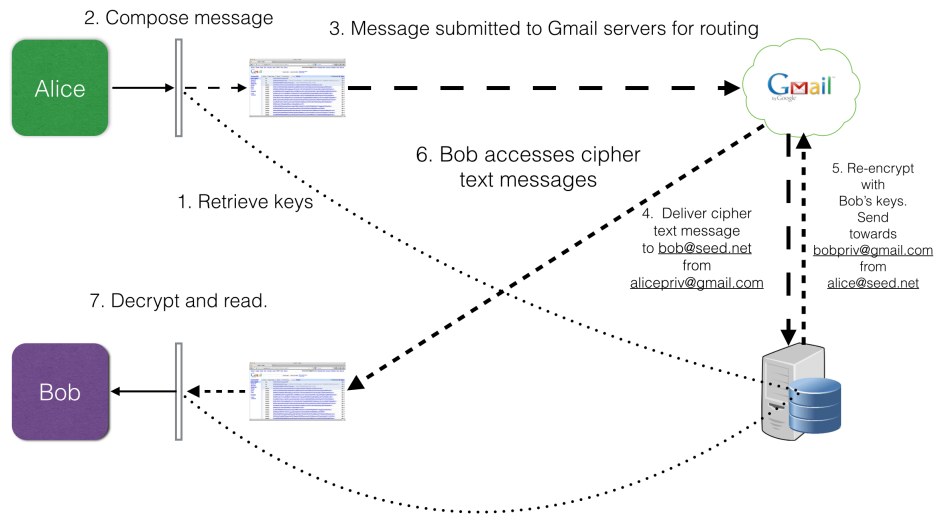


Figure 2: Emailing with SEED: Alice sends Bob a message.

3.2 Writing an email

SEED add-on's main purpose is to prevent any text that the user inputs into Gmail from being submitted to Google servers unencrypted. User Alice installs SEED add-on as a browser add-on for Firefox. As an add-on, SEED add-on runs automatically in the background of Alice's browser session and will activate when she logs onto Gmail.

Alice opens the compose window in Gmail to write an email. When she is finished writing, she can either send the message, save the message as a draft, or delete her draft entirely. When she selects one of the three actions by clicking their respective buttons in the Gmail interface, SEED add-on pulls the text from the message entry boxes, encrypts it using our scheme, and fills the empty entry boxes with the message ciphertext. The ciphertext version of the message, which conceals any information that can identify the content of Alice's email, is submitted to Gmail's servers in lieu of the original message. If Alice wishes to edit a saved draft in plaintext, the ciphertext draft is decrypted automatically the next time she views it.

3.3 Sending email

User Alice wants to send a message to user Bob with SEED. She composes an email in the Gmail web client. When Alice sends Bob the message, our browser add-on, SEED add-on, encodes the message with her encryption keys (retrieved from the SEED server and the Key Management Service when she logs into Gmail). The encrypted message is then sent to Bob's public email address, which routes the message to the SEED email server. The SEED server decrypts the message and re-encrypts it using the Bob's keys. The message is forwarded to Bob's private address, where he can now decrypt it with the key that he retrieves from the server on his instance of SEED add-on (Fig. 2).

The server-side key switch is necessary because we encode messages with AES symmetric encryption. In symmetric encryption, Alice uses a single key to both encrypt and decrypt her data. Alice cannot share her key with Bob because, with it, he could access any of her encrypted private information. Bob has his own key and can only decrypt data encoded with it. When one SEED user sends a message to another SEED user, the SEED server re-encrypts the message with the recipient's keys so that the recipient can decrypt the encoded information and read the message originally encrypted with the sender's keys.

3.4 Receiving email

Alice receives email at her public email address which directs messages to the SEED server. If the messages are not from SEED users, the server encrypts each message with Alice's encryption keys. If the messages are from another SEED user, they are decrypted before being re-encrypted with Alice's keys. The encrypted messages are then sent by the SEED server to Alice's private address. When Alice opens up the Gmail web client with SEED add-on running in her browser, SEED add-on decrypts the message ciphertext displayed in Gmail and redisplay the decrypted information to Alice. Alice can now read the contents of her encrypted messages.

3.5 Searching email

To enable search, Bob submits a search query that is encrypted respecting Gmail search syntax (e.g. "from:Alice@example.com" \Rightarrow "E(Alice@example.com)"). We hold the salt and IV fixed in our AES implementation so that the ciphertext output is not randomized and is, therefore, predictable. Bob queries Gmail search with plaintext, which matches the ciphertext output query with email body ciphertext. The ciphertext matching yields the correct search results because word mappings from plaintext to ciphertext are one-to-one.

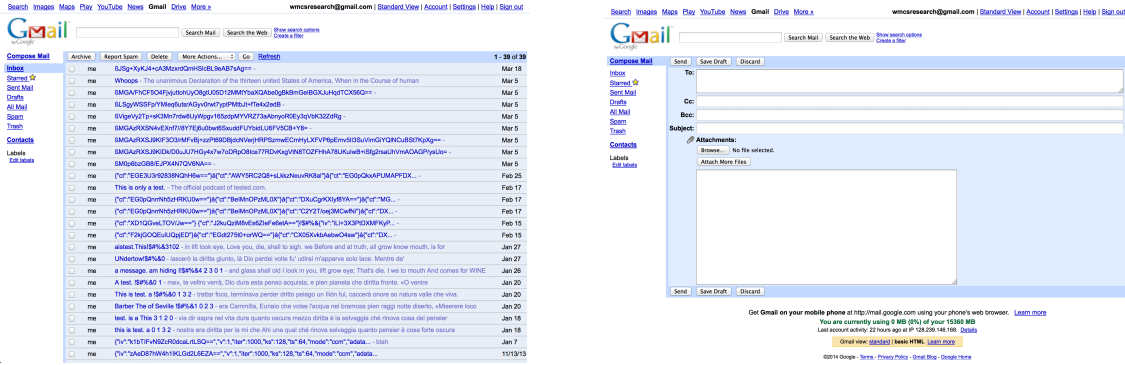
4 System Design

We present the specifics of our implementation beginning with the design of our browser extension, SEED add-on. We discuss several aspects of the Gmail user experience we interface with in order to ensure all browser-side activity remains secure. We then detail our specific encryption scheme, explaining how its construction enables Gmail search without revealing the email contents to attackers. We finish by describing our server infrastructure.

4.1 Firefox add-on: "SEED add-on"

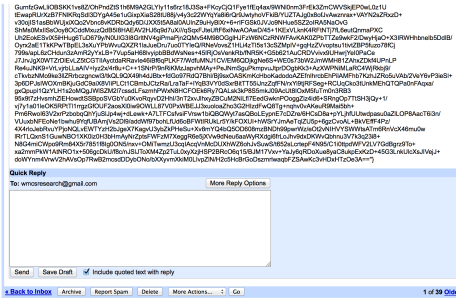
We needed to efficiently access and interpret the HTML in Gmail's web client when it is displayed to the user. We chose to develop a Firefox add-on as it is a well-established platform ran by Mozilla,

offer is a more secure web client that requires the browser to load and render an HTML document per Gmail view (i.e. one page refresh per link clicked in the interface). This traditional approach to web architecture performs extremely well as it adheres to the basic web page requirements conceptualized at the advent of the world wide web. Even better for us, it segments the Gmail web experience into predictable, clearly-defined, and, thus, manipulable HTML documents (Fig. 3).



(a) The Search Bar and Inbox

(b) Compose Message Window



(c) The Quick Reply Window

Figure 4: User input in Basic HTML Gmail.

The purpose of our add-on, SEED add-on is to intercept any information entered into the Gmail website and to decode ciphertext in the GUI so that it is user-interpretable. In Gmail’s Basic HTML view, there are three sections of the interface where the user inputs information into web page forms. The search bar is the most ubiquitous GUI element, displayed prominently at the top of every Gmail interface instance. The second is the message drafting form, displayed when composing a message or editing an email draft. The last, the “Quick Reply” form, is visible when reading messages. It is shown when clicking into messages from the inbox and is located at the end of the email message body. Quick replies streamline the email work flow by allowing the user to respond in-line to an email (Fig. 4).

SEED add-on takes advantage of the Firefox add-on infrastructure’s Common JS modular design to compartmentalize its obligations across a duty-delegating main process and several purpose-specific content scripts. As Mozilla clarifies in the add-on SDK documentation, “an add-on which needs to in-

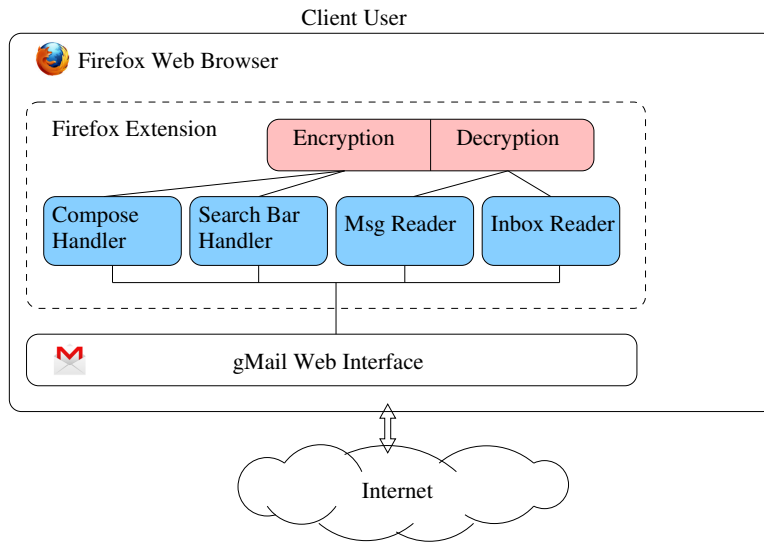


Figure 5: Structure of the SEED Firefox Add-on

Interact with web content needs to be structured in two parts: [a] main script runs in the [main] add-on process; any code that needs to interact with web content is loaded into the web content process as a separate script. These separate scripts are called content scripts.” [37] Our content scripts provide one of three services to the extension. A “reader” content script parses encrypted information on Google’s website and presents it to the user decrypted (*inbox-reader* and *message-reader*). A “handler” content script ensures that any information entered into the web client is encrypted before submitted to Google servers (*searchbar-handler* and *compose-handler*). The last variety of content script contain our encryption libraries (*crypt* and *sjcl*). We keep our encryption libraries in content scripts to overcome process permissioning models in the Mozilla add-on architecture and to enforce a certain amount of modularity in design (Fig. 5). As stated earlier, Mozilla distinguishes between the add-on’s main process and any process that needs to access web page content. Mozilla did this to reflect Firefox’s slow migration to a multi-process model. Called “Electrolysis,” the project believes that a multi-process paradigm where web content, the browser, and add-ons all operate in different processes will affect substantial usage and performance benefits to web browsing pipeline. The Electrolysis team state in the project description that:

The two major advantages of this model are security and performance. Security would improve because the content processes could be sandboxed (although sandboxing the content processes is a separate project from Electrolysis). Performance would improve because the browser UI would not be affected by poor performance of content code (be it layout or JavaScript). Also, content processes could be isolated from each other, which would have similar security and performance benefits. [35]

Although Firefox has yet to migrate to a multi-process architecture, the Mozilla add-on SDK has adopted its process sandboxing design far in advance in order to increase add-on security and stability.

Each content script is able to directly share information with other scripts, as long as they run in parallel and respect a certain execution hierarchy. Content scripts executed earlier from the main add-on process are accessible by those launched later, enabling code sharing and reuse. Through this feature, we are able to provide our encryption implementations to each content script in our script pool.

4.1.1 Our encryption scheme, “sjcl”, and “crypt”

SEED’s encryption scheme needs to output predictably enough to support Gmail search, yet remain immune to any frequency analysis that could be used to reveal mappings from plaintext to ciphertext. We also needed a proven encryption algorithm that we could build our scheme on top of, especially one that could be fixed to generate one ciphertext output from one consistent plaintext input.

After considering asymmetric, public/private key encryption, we decided to use AES symmetric encryption. AES is desirable as it is widely-used, sufficiently uncrackable by brute-force attacks, and can be fixed to generate repeatable ciphertext. We found a well-vetted JavaScript AES library from academia called the Stanford JavaScript Cryptography Library, or SJCL, that we used as our AES implementation in our encryption scheme. AES takes a 16, 24, or 32 bit key as input, as well as two randomly generated values, a salt and initialization vector (IV), that vary the ciphertext output across each encryption [13]. We can create reproducible output with SJCL by manually setting all of the encryption parameters it uses when encrypting and decrypting. We predefine specifically the IV and salt rather than randomly generate them each time we encrypt data. Providing default values to AES removes the randomness from the output and yields the one-to-one plaintext/ciphertext mappings we desire.

We cannot simply encrypt each message from plaintext to ciphertext and expect our cipher to be unbreakable. If we were to encrypt messages in such a way that preserves the original word ordering, our system would be susceptible to frequency analysis. Frequency analysis is a cipher-cracking technique that tries to deduce plaintext from ciphertext by establishing the relationships between ciphertext words that correspond with relationships between plaintext words. For instance, one can assume that the most often repeated ciphertext word in a message will correspond to a frequently used English word, such as “the.” An attacker can reconstruct messages one word at a time, gathering every mapping from cipher to plain from the context of the previous words he decrypts. The cracking process is expedited given more ciphertext data that can be analyzed for common linguistic patterns. In the case of email, an inbox full of ciphertext should be a large enough sample to generate concrete frequency relationships that are interpretable. We develop a message format that eliminates the semantic meaning of an email message and evens out word frequencies. The format render any frequency analysis of our email contents unlikely to be successful, if not completely implausible.

Our encryption scheme is implemented in a content script called *crypt*. *crypt*, in turn, uses functions defined in the SJCL encryption library, so we run the Stanford library as its own content script, named *sjcl*. *crypt* features several encryption functions, each used to conceal a separate part of an email such

1. An unencrypted message:

To: r_1
From: s_1
Subject: $w_1w_2w_3w_4$
Time: t_1
Body: $w_1w_2w_3w_4w_5w_2w_3w_6$

2. Hashed word array:

$$\{s_1, w_4, t_1, w_6, w_3, w_1, w_2, r_1, w_5\}$$

3. Making the index string:

$$\{s_1, w_4, t_1, w_6, w_3, w_1, w_2, r_1, w_5\}\#56418643\#5641\#2\#7$$

4. Encrypted message:

To: r_{SEED}
From: s_1
Subject: $E(w_1w_2w_3w_4)$
Time: t_2
Body: $E_f(s_1), E_f(w_4), E_f(t_1), E_f(w_6), E_f(w_3), E_f(w_1), E_f(w_2), E_f(r_1), E_f(w_5)...padding...#E(56418643)\#E(5641)\#E(2)\#E(7)$

Figure 6: SEED MESSAGE ENCRYPTION ALGORITHM : w_n , Plaintext word; r_n , Receiver address; s_1 , SEED sender address; t_2 , Time stamp; $E(-)$, Randomized AES ciphertext; $E_f(-)$, Fixed AES ciphertext; # , Delimiting Unicode characters

that no information regarding the sender, time, subject, or message body is interpretable by Google or any other attacker. We conceal email metadata to hide patterns of communication in our email body. It has been shown that email metadata can be leveraged to identify the user.

We encrypt message bodies using `crypt.encryptMsg()`, a function that takes as input a message subject, time, sender, and body. `encryptMsg` combines the text of all four fields and hashes each word into an array such that no word repeats and the original text ordering is lost. We use a secure hash function in SJCL that collects user browser input as entropy to achieve a truly random ordering. We then rewrite the original message text, replacing each word in the message with its index in the hashed array of words. The final index string can be understood as the original message, recomposed in terms of indices into our scrambled word array. We encrypt the index string using fully randomized AES so that an attacker would have to break AES encryption, as well as deduce ciphertext/plaintext mappings to reconstruct the original message. We also construct a separate index-based text for the subject and note the indices of the sender and timestamp in the hashed array.

Last, we encrypt all the words in the word hash and output our message. In our output, we first list each ciphertext word in the hashed word array, separating them with commas so that they are indexed by Gmail's search engine (Gmail seems to demarcate words in a message with whitespace and commas).

Then, we encrypt each index string entirely with fully randomized AES encryption, listing each after the words separated by Unicode characters we can easily identify during decryption. We use Unicode characters to demarcate different sections of our message because the SJCL ciphertext is guaranteed to yield ASCII values. We can therefore split our message on Unicode characters during decryption to gain access to each portion of the encrypted message (Fig. 6).

We decrypt our message by separating the array of ciphertext from the encrypted index strings. We decrypt the index strings and reconstruct the message by looking up the index in the ciphertext array to gain the word originally in that place. We reconstruct the subject and identify the sender and timestamp in the same way. We decrypt the ciphertext words and output the message, along with the subject, sender, and time to the user.

crypt contains a second encryption function for obscuring subject lines. We cannot encrypt subjects displayed in inboxes like those displayed in message views because of the text shortening Gmail performs to fit the subject text to the inbox display. When we tried to encrypt subjects like messages (outputting a scrambled word array followed by a index string that preserves the original message word ordering), Gmail often truncated the subject such that the index string was no longer visible when viewed from the inbox. This truncation made it impossible to restore the original word ordering of the message. To overcome this, we encrypt subjects in two formats in order to support the two ways that Gmail displays subject lines. The first is shortened, 78 character subject preview displayed from the inbox. The second is the full subject line displayed when individual email are opened.

For the 78 character preview, handled in `crypt.encryptSubject()`, we prune the plaintext subject string to 57 characters such that the ciphertext output will be 78 characters long (we use a conversion $c_l = \frac{4}{3}[p_l - 1]$ where p_l is the length of the plaintext, c_l is the length of the ciphertext) [21]. We then encrypt the string as a whole and insert the ciphertext output into the subject field of a message. We encrypt the full subject text with the message body using the method outlined above. It is worth noting that Gmail does not support subjects longer than 998 characters and will shorten any that are longer to fit their size limits. The subject length limits are not arbitrary, but outlined in the Internet Task Force document, RFC 2822, that formalizes email structure [20]. We obey the IETF convention, stemming plaintext subjects at 998 characters before they are encrypted with the message body. Shortening the subject has the provided benefit of limiting the impact a subject has on message size.

Security precautions. It is worth noting that we pad messages with randomly generated gibberish words to a 100 word minimum length. We do this to prevent a specific man-in-the-middle attack that Google can perform to break our cipher. If Google were to send a one-word message to a paranoiac-enhanced Gmail account, they could identify the ciphertext version of message after it is uploaded by the trusted server to Gmail. With ciphertext and plaintext versions of a word in hand, they could slowly break our system's word mapping scheme. Even worse, they would be able to disassemble our secure, long-term mappings between words when the message is encrypted with the long-term key and re-uploaded. The insertion of random words into short messages makes identifying a one-word message

much more difficult for Google. We pad to 100 words arbitrarily: more words will guarantee better message security but will fill server space faster. Padding with less words is more space economic, but gives Google (or any snooper) a higher probability of identifying their plaintext amongst the ciphertext array. We leave the user to decide which convention is better for their application.

4.1.2 Executing SEED add-on: the “main” process

The main process (defined in `/lib/main.js`) initializes the add-on. It constructs the add-on’s GUI components (a Mozilla “widget” which, in our case, is used as an on/off switch) and attaches an event listener to the current browser tab (the organizational unit that describes a browser window). The event listener will wait until the tab’s DOM content is loaded before executing a URL parser, which identifies whether the browser is currently displaying Gmail and, if so, identifies which part of the Gmail interface the user is looking at. There are three content views within Basic HTML Gmail: an inbox view; a message view; and a compose view. The add-on distinguishes the current content view by examining the directory structure of the URL. By correctly identifying which URL directory corresponds with which content view, the SEED add-on will launch a content script that will perform the necessary DOM manipulations for a secure Gmail session. There is additionally one content script targeting the search bar, appropriately called *searchbar-handler*. It is launched for every content view due to the fact that the search bar is present throughout Gmail’s web client. It will be discussed further when we outline the functions of each content script.

4.1.3 inbox-reader

The primary screen and hub of the Gmail interface is the email inbox. This is the first screen displayed to the user after login, displaying up to the first 50 messages of the user’s email in a table format. Each table entry displays the message’s sender and subject line, as well as the time the message was sent or received. Our first goal was to isolate this information from the rest of the HTML DOM to work with later.

The content script called *inbox-reader* parses and manipulates all data associated with the inbox view. *inbox-reader* moves through each message header in the inbox and decrypts the subject line. When it finishes execution, it displays the decrypted contents of the inbox to the user. In order to retrieve the individual message headers from the displayed inbox, *inbox-reader* identifies the HTML object storing the message values. In this case, the basic HTML inbox is an HTML `<table>` object in the DOM. The content script then iterates through each cell of the table, pulling each message’s sender, time, and subject information, and decrypts the subject header’s ciphertext. The information is displayed to the user, unencrypted, in a window alert, requiring no modifications of Gmail’s web page source.

4.1.4 message-reader

The message interface in Gmail is accessed when the user clicks into a single message thread from the inbox. The user is presented with the contents of the selected email, as well as any prior email in the conversation chain collapsed in a stack above the message header. In order to decrypt the message body for the user, the content script iterates through the DOM collecting the message body and references to buttons in the quick reply dialogue. By retrieving the message body and removing all HTML tags inserted into the text, the content script can then decrypt the ciphertext and display the subject, timestamp, and email body and display them to the user. The script eschews the subject header in favor of the subject text hidden in the message body because the subject header is abbreviated to 78 characters. The subject text stored in the message body is complete, as we explain in the description of SEED’s encryption scheme. Likewise, we forego the timestamp displayed in the email header for the timestamp in the message body, which reflects the actual “message received” time.

We access the “Send” and “Save Draft” buttons from the quick reply dialogues so that quick replies will be encrypted when submitted to Google servers.

4.1.5 searchbar-handler

The content script, *searchbar-handler*, is active wherever the Google search bar is present in Gmail. *searchbar-handler* pulls references to the search bar HTML `<input>` tag, the “Search Mail” button, and the “Search Web” button. It attaches event listeners to these buttons that wait for the user click. As soon as the user clicks a search button, the SEED add-on encrypts the query, observing several search format requirements.

We attempt to support Gmail search keywords in order to allow metadata-based searching of the inbox. For example, if we were to search all messages written by Bob@gmail.com containing the word “tennis” that also have attachments, Gmail allows us to search: “tennis from:Bob@gmail.com has:attachment” and receive the subset of messages in our inbox where the query conditionals hold true. Gmail is able to do this using elaborate message indexing that seems to sort messages according to text and metadata content. Gmail’s search capability is so extensive that one can search for individual words in message bodies or, more recently, chunks of text in attachments of a supported file format. While a majority of the search keywords are functional in our implementation —31 of 52 search operators, albeit some with structure modifications to support our message format conventions —9 of 52 keywords are broken by our message encryption format. Because we store a significant portion of the message’s metadata in the encrypted message bodies, search queries that utilize metadata to retrieve results must instead match strings within the message body text. As an example, a search for messages received from Carlo@gmail.com would normally be conducted with the query, “from:Carlo@gmail.com”. Now that sender information is integrated into the message body, querying all messages from a sender becomes a query matching the sender email address with a ciphertext

string in the message body (“from:Carlo@gmail.com” would become “E(Carlo@gmail.com)”). The following search operators obey this substitution pattern, where we query for strings in message bodies instead of searching the relevant email metadata field: “from:”; “to:”; “subject”; “list”; “cc”; “bcc”; and “deliveredto”.

We support time-based searches of message contents, also submitted with structure modifications to reflect the idiosyncrasies of our system. We alter any time-based query to reflect the message upload time from our email server. This is because our system downloads any email older than three days, encrypts it using the user’s long-term key, and uploads them back to Google with IMAP. Although the original time that the message was received is safeguarded in the message body, it is impractical, for example, to search for any message older than a date by matching every previous day’s date string with potential dates in email bodies. Instead, we round the query date to the closest three day upload occurring after the date. Rounding guarantees that we will only search results from messages before the original message’s upload date, whose timestamp on Gmail will be up to three days after its original arrival time. When the date is within the three day short-term period (“after:twoDaysBefore”), we can search message times normally as the Google timestamp will be close enough to the original receipt time (message received at 7 PM will be uploaded to Google servers by midnight of the same day). If the time query specifies a period that includes messages from the short-term key pool and the long-term key pool (e.g. “before:twoDaysBefore”), the usual message reception time can be searched for. This is true because all long-term messages will be older than the short-term date we are searching by and therefore will be included among the body of messages to be searched.

We can prevent the delay in messages being forwarded to Gmail if the user needs to ensure the timely delivery of email. Instead of waiting for multiple messages to collect at the SEED server, we forward the message as soon as it is processed by SEED with several dummy messages to the Gmail account. We mark the dummy messages so that our add-on can ignore them when decrypting the Gmail inbox. While we can ensure that the time stamps of messages encrypted with the short-term key will now reflect the original message arrival time, the additional dummy messages incur an additional storage overhead on Gmail’s end. It is also more difficult to gracefully handle dummy messages in the SEED add-on user interface. Again, we allow the user to determine which message forwarding scheme fits better with their needs.

We cannot support search operators that rely on relationships between Google Apps metadata, such as searches chat messages, Google+ messages, etc.. Furthermore, we do not support message labelling and therefore cannot support label searches. For all search keywords we do support, *searchbar-handler* performs the appropriate query conversions using a relational hash table, where keywords serve as keys and the query conversion is the associated value. The script looks up the search keyword in the hash table (implemented with a JavaScript object) and is returned the modified search query. The modified query is then submitted to Google servers, returning the expected results of the original search keyword. If we are searching strings of text, we encrypt each word into its short-term and long-term ciphertext

form and submit the ciphertext as our query.

4.1.6 compose-handler

compose-handler is launched when viewing the message composition screen in Gmail. It iterates through the DOM and gains access to the “Send,” “Save Draft,” and “Discard Draft” buttons on the page (there are two of each button: three above the message input boxes and three below). The script then attaches an event listener to each button such that when they are clicked, the email is encrypted before it is submitted to Google. When encrypting, the message’s sender field is replaced with the email address of the trusted server so that all outgoing traffic is rerouted through our server. We do this so that messages can be reformatted depending on whether the recipient is a SEED user, a process we will discuss in the server specification. We also replace the subject with the truncated 78-character version, pull the timestamp from the page HTML, and encrypt the message body with the sender email address, the full subject, and the timestamp.

If, instead, we save a draft, the same encryption process occurs. However, *compose-handler* will look to see that the message is decrypted after it is saved so that the user can continue to edit the draft. This also prevents double encryption of the message when the user clicks one of the buttons twice. We encrypt drafts with a one-time key before they are discarded. This is so that we can ensure our data remains uninterpretable in the off chance that Google keeps a record of discarded drafts.

4.2 The SEED server

Our system relies on a third party server to perform several tasks. The following sections will outline our theoretical design for this server. As with the SEED add-on add-on, we will outline the structure of the server by discussing each of its three modules: proxy mail, email archive, and key management.

4.2.1 Proxy mail {transfer, delivery} agent

The SEED server serves primarily as an email proxy server that administers all incoming and outgoing mail traffic to our user’s Gmail account. It is an augmented email server, modified to fulfill the security requirements of our encryption scheme. The mail transfer agent is the engine of the email architecture, sending and receiving all messages from other MTAs (email servers).

The proxy component of the trusted server has a email address specific to the server. This email address is the public email address for the user of our system. That is, any mail that the user wishes to receive should be sent to the trusted server’s domain such that it can be encrypted before being uploaded to Gmail. Gmail and the Gmail address is the front end by which a user of our system accesses and reads their email. Any message composed on the Gmail web client is sent through our server. Therefore, the trusted server receives email such from two sources: those composed on a user’s private Gmail account and those sent to it from external email addresses. When the server receives email from external

addresses, it checks for the system user that the message is addressed to and then encrypts the message with that user's short-term key. It places the message in a sending queue that is not forwarded to Gmail until midnight of the same day. We forward messages to Gmail in clusters to make it more difficult to trace a single message sent to the server onto the Gmail account. This is to directly prevent a ciphertext-gathering man-in-the-middle attack by Google that they perform to break our encryption scheme. If the message sent to the server was sent by another user of our system, it will be encrypted with a key that will not be decodable by the recipient. In this case, the trusted server's email proxy decrypts the message with the sender's key before re-encrypting it with the receiver's key. This keeps the message contents secure while specifying them to a format where the receiver can still interpret the email information.

4.2.2 Key Management Service and Mail Archival Agent

The trusted server manages all user's encryption keys in a separate process we call the Key Management Service. This process authenticates users of the SEED add-on, sending them their short-term and long-term encryption keys in an encrypted format with a server call when they log on to Gmail. The key management service uses an internal database to keep track of each user's respective keys. The database can be encrypted for further server-side security. A key in our system is all the randomly generated parameters that we feed to SJCL's AES implementation. There are three parts to a key: a salt, an initialization vector, and a 32-character password. Each user has two keys with which they protect their information: a long-term key which is fixed from the onset and a short-term key that is regenerated every three days. The short-term key is used for data transport. It is the key that is used when messages are sent from the SEED add-on. It is used when the server encrypts received messages before uploading them to Gmail. The long-term key is used for long-term storage of email on Gmail.

Our server's last process is called the email archival agent. The agent downloads all messages older than our short-term key threshold, decrypts them, and re-encrypts them using the long-term key. These messages are then re-uploaded to Gmail for long-term storage.

We encrypt messages with our short-term key for a time span of three days. After this period, we re-encrypt these messages with a second key to overcome a specific vulnerability that Google can use to gather a ciphertext-plaintext combination and break our system. We pad messages with randomly generated gibberish so that word counts cannot be used to track messages sent through our server to the Gmail account. We also upload messages in bulk so that a timestamp cannot serve as a unique identifier for a message. Although these measures could prove sufficient for keeping Google from identifying a message they sent themselves through our server, encrypting our information on two separate occasions increases the amount of indirection Google would have to overcome to break our scheme. Imagine that Google sends our system a message of 125 words. When encrypted on our trusted server, the message length increases to 167 words and is forwarded to Gmail. Google may be able to identify the message by building a cross inbox database of ciphertext words that it encounters in each message. When the

message is re-encrypted with the long-term key, the contents are scrambled again and the message is padded to a different length —this time 212 words —which forces Google to try and identify its message again.

Google could still identify its message using a cross inbox analysis of all the repeated ciphertext words in each email. If the original 125-word message contained 72 unique words, then the ciphertext version of that message will have 72 repeated ciphertext words that are likely to repeat in other messages. The randomly generated gibberish words will not be repeated in other messages. We can overcome this vulnerability by establishing a dictionary of gibberish words to pad email such that fake word frequencies are present in the email contents. As long as we choose words from the dictionary with a heuristic that approximates real word frequencies, we should be able to make cross message word frequency analysis impossible.

5 Evaluation

We benchmark the performance of the SEED add-on Firefox add-on by tracking the size changes in messages after encryption, measuring the amount of time it takes to encrypt and decrypt messages, and noting the speed and accuracy of encrypted search. We show the plausibility of our system by implementing our encryption algorithm on the browser. From our results, we extrapolate the unique performance challenges the server-based portion of our system may face. All tests were run on a Macbook Pro with a 2.3 GHz Intel Core i7 quad-core processor and 8 GB of RAM.

5.1 Methodology

In order to create a rigorous and representative test environment for the SEED add-on, we created two samples of encrypted email. The first sample is 10,171 encrypted email generated using the fortune UNIX utility. It uses text from the UNIX utility fortune to create message bodies and subject lines. The email subject is a slice of the first 53 characters from the “fortune.” Upon invocation on the command line, fortune displays a random quote to the user. These quotes are pulled from several text files stored in the fortune directory. Each “fortune” is anywhere between 3 bytes in size (the message “42”) to 2490 bytes in size. The fortunes are ideal email bodies because they cover a range of sizes that represent typical email lengths (people more often write email that is several sentences in length than 500 or more lines long). We use a subset of 10,171 fortune quotes as our email bodies. In our results, tests involving these 10,171 messages will be called our “fortune” results.

When inspecting our own personal email inbox, we noticed that our largest message size was closer to 50 KB, about 25 times larger than the largest message in our fortune dataset (it is an HTML formatted message with embedded content). We created a second sample of 1027 messages to account for the difference in size between our test bed and what we consider an upper limit on message size in a real-

world usage scenario. Our second message set is generated in 50 byte size increments from 7 bytes to 51,156 bytes. The messages are filled with words that are randomly selected from an English dictionary and written into text files with whitespace between each word. By deliberately creating messages that scale in size linearly, we hoped to track the size complexity of our encryption algorithm as messages became bigger. In the paper, we will refer to this dataset as the “50k-spread” results.

We coded a second Firefox extension, called Email Generator, for testing purposes. This Firefox add-on reads through a text file containing the desired message text separated by some delimiter and sticks it into an email body. The email are then encrypted using the *crypt* and *sjcl* modules from SEED add-on and are written out to a text file. These text files are then, in turn, parsed by a Python script and sent to a testbed Gmail account using Python’s built-in SMTP library. The cryptographic performance of Email Generator should reflect the performance of SEED add-on as Email Generator and SEED add-on share the same libraries and software platform.

We devised three experiments that stress test Gmail’s search engine. The first involves searching a word common to many or all email so that Gmail compiles a high volume of results to return to the user. For this test, we searched the word “the.” In our second test, we search a word that we know is not represented in our inbox. The word choice forces Gmail to parse the bodies of all email in the inbox to verify that no messages match our query. For this, We searched the word “tawer” and verified that it was not in our message bodies with *grep*. Our final test gauged the effect of search query size on the amount of time it takes to return search results. We used a python script to randomly generate queries of nonsense words ranging from 3 to 10 characters in length. For the first two tests, we performed as many trials as it took for the standard deviation of all trial times to converge. In both cases, convergence occurred at ~11 trials.

We test search performance in the Gmail web client while running an instance of the SEED add-on Firefox add-on. All searches were conducted on an inbox populated with the 10,171 fortune messages. Total search time was determined to be the period between submitting a search query (i.e. when the user hits “Enter” or clicks the “Search” button) and Gmail displaying the results to the user. All the search tests were performed on a stable, high-speed internet connection to reduce timing variability due to network-related lag. We estimate the normal performance of Gmail search by turning off our add-on and searching the ciphertext strings that correspond to the plaintext queries we test with SEED add-on running. Our internet speed at the time of testing was 65.47 Mbps download and 57.74 Mbps upload (gathered using speedtest.net).

5.2 Message size

Our results show that across the 10,171 fortune email messages, encrypted messages were 3.54 times larger than their plaintext counterparts. The largest difference in size was found in a ciphertext message 14.88 times larger than its plaintext (the transformation was from 8 bytes of plaintext to 119 bytes of

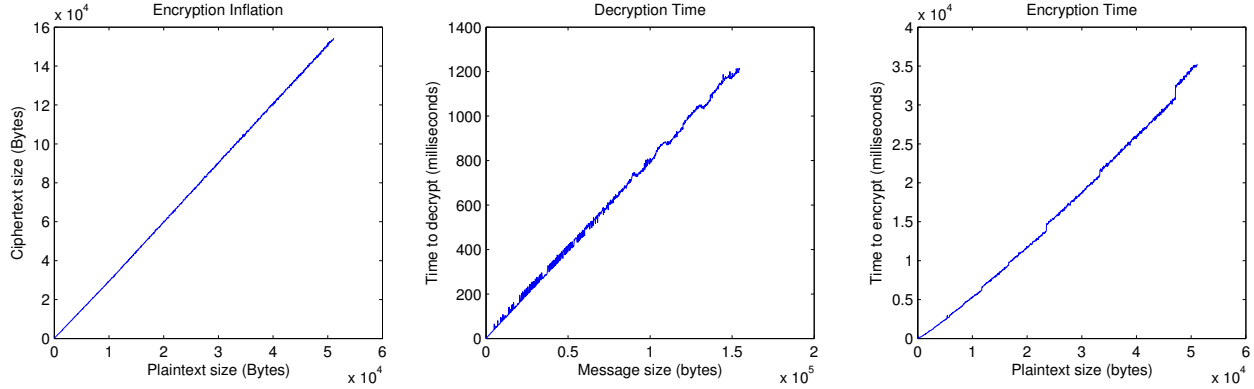


Figure 7: *Left: the relationship between ciphertext and plaintext message size. Center: decryption time as a function of ciphertext message size. Right: encryption time as a function of ciphertext message size.*

ciphertext). The smallest change in size was a ciphertext 1.77 times the size of the plaintext. When the same tests were repeated with our 50k-spread messages, we saw that encrypted messages were on average 3.021 times larger than their cleartext counterparts (Fig. 7).

Again, the largest change in message size from plaintext to ciphertext occurred when encrypting the smallest messages. The smallest message (7 bytes in size) encrypted to a ciphertext size of 141 bytes, expanding to 21 times the size of the original email. A similar transformation is seen when encrypting the second email, where the ciphertext size is 6 times the plaintext size. By the eighth email, the ciphertext/plaintext ratio is within one standard deviation of the average. A large difference between ciphertext size and plaintext size occurs in the first email due to the baseline length of the encrypted index string in our message bodies. Due to the additional encryption parameters appended to the ciphertext, the index string is about two-thirds the size of the message in the smallest encrypted message. As the length of the encrypted ciphertext begins to dominate the size of the message, the ciphertext/plaintext ratio approaches the population mean.

5.3 Encryption and decryption performance

When encrypting the fortune messages, our average encryption time was 190.364 ms. Our average decryption time was an order of magnitude less, at 10.088 ms per message. The largest message in our fortune data set encrypted in 1734 ms and decrypted in 103 ms. The 50k-spread results yielded an average encryption time of 16153.387 ms and an average decryption time of 615.980 ms, with the largest message encrypting in 35285 ms and decrypting in 1209 ms (Fig. 7).

While running these tests, we identified a disparity in cryptographic performance between our Email Generator testbed and the SEED add-on Firefox extension. When the largest message from the 50k spread data was entered into a Gmail compose window, SEED add-on could encrypt the message in 5118 ms and decrypt it in 1438 ms. The difference in execution time is caused by how SJCL gathers entropy for random number generation. They seed their in-house random number generator by tracking

user mouse movement in the browser window. Email Generator, however, runs as a script without a user interface. The lack of mouse input forces SJCL to gather entropy from system time stamps, a seeding process that takes much longer than collecting high-entropy mouse movement. The longer encryption times in our results reflect the delay caused by time-based entropy collection.

In comparison, the smallest 50k spread message could be encrypted in 192 ms and decrypted in 185 ms. On average, SEED add-on would encrypt large messages quicker than Email Generator by a factor of seven. Decryption in SEED add-on was about ten times slower than decryption in Email Generator. With small message sizes, SEED add-on was consistently slower than Email Generator. Despite the poorer performance, the amount of time required to encrypt small messages is negligibly small.

Despite the performance gap between our test bed and the actual performance of the SEED add-on, the linear trend in encryption/decryption time and message size is found in the performance of the SEED add-on. Therefore, we can still gauge how our system scales given the slower results we collected.

5.4 Search performance

We found that, even when searching a large inbox, there is very little performance hit resulting from encrypted search from SEED add-on. In the left chart of Fig. 8, we see that there is no discernable difference between a plaintext “normal” search of Gmail and the encrypted search performed with SEED add-on. During some trials, encrypted search outperforms regular search by ~60 ms. When searching for words not in our inbox, encrypted search was not any slower than regular search (Fig. 8, center). Again, at times it performed better. We only begin to see a difference in performance when searching queries with more than 25 words in it (Fig. 8, right). The gap in time can likely be attributed to the amount of time necessary to encrypt the query. Even so, the performance difference is negligible at around half a second slower. It was impossible to test queries above 75 words as Gmail is unable to process them. It is likely that most users’ search queries are under 10 terms, where using SEED add-on would pose no noticeable performance hit.

Analysis. Gmail is likely able to gather results quickly by distributing the search computation across each server that stores a chunk of email data. Parallelizing search, in addition to liberal caching and other heuristics that the Google File System uses to prioritize information, suggest that our search time is dominated by network latency (the round trip time between submitting search results to Google servers, Google processing the search, and returning the page of results to the browser instance). Fluctuations in latency would explain the variation and inconsistencies in our search timing results.

Given that Google returned results in under half a second for an inbox with ~8 MB of text data, we postulate that Google search performs quickly independent of overall inbox size (the more our data is distributed across many servers, the more distributed the search operation can be). Furthermore, we noticed that the number of search results as the user pages back through all the results. This phenomenon suggests that Google continues to load results in the background as the user browses through the first

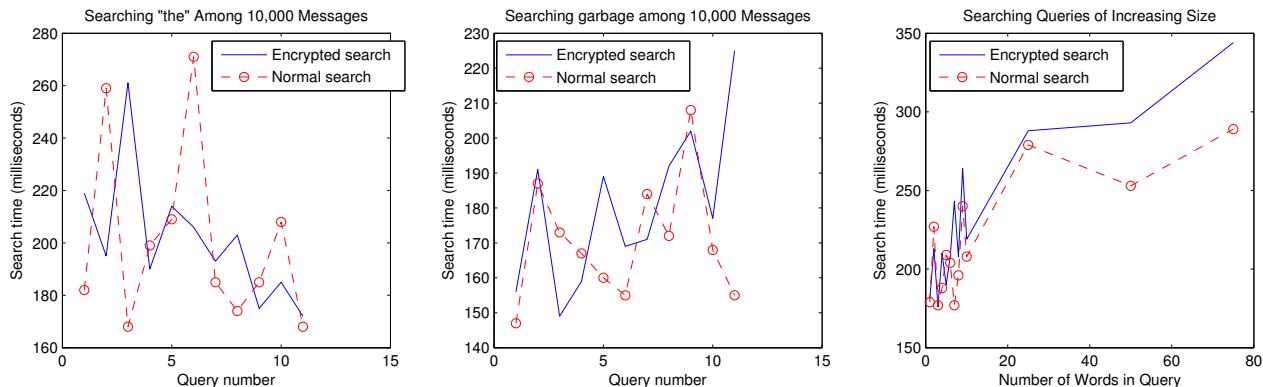


Figure 8: *Left: Timing results for searching “the” in an inbox of 10,000 messages; Center: Timing results for searching gibberish in an inbox of 10,000 messages, the gibberish is guaranteed not to match any word in any message; Right: Timing results for search queries of differing sizes. Each word in the query is gibberish not found in any message.*

returned messages. If Gmail loads search results in the background when a high volume of messages is returned, it is still impressive that Google returns the results in chronological order.

5.5 Search accuracy

For the most part, encrypted search via SEED add-on yields the expected results in that "results" messages match encrypted queries. Certain aspects of Google search that make it robust, such as word-stemming and punctuation removal from queries, are not supported by our add-on. This means that a search for the word “flowers” will no longer yield messages containing “flowers” and “flower”, but only messages with the ciphertext “E(flowers)”. While it is easy to make the association between the singular, plural, and punctuated forms of a word in plaintext, the resulting ciphertext will not be related to each other. The structural relationship between letters in a word is eliminated by the “confusion” and “diffusion” necessary in encryption algorithm design [28]. However, it is impossible to replicate the word-stemming or punctuation removal that Gmail performs with plaintext search in naïve encryption of search queries.

We cannot identify what portion of the ciphertext word to remove to retrieve the word’s stem, nor can we retroactively strip words of their plurality or punctuation when we encrypt messages because we would be unable to reconstruct the original word forms. A possible solution would be to query the singular and plural forms of a word’s ciphertext to mimic the effect of word-stemming. We could also strip words of their punctuation and plural endings when we encrypt email, keeping some sort of record of the original word forms in the message body. In any case, we agree that a handicapped but secure ability to search encrypted email is a significant improvement over the lack of search in previous encrypted Gmail implementations.

6 Conclusion

We designed an encrypted webmail system, SEED, that uses Gmail as background storage infrastructure. We prevent any user data from being visible to Google servers by encrypting all message metadata. The trusted SEED server can decrypt encrypted message routing metadata and send messages to the intended recipient. The trusted server coordinates an email archival system to overcome frequency analysis-related vulnerabilities and performs key management so that the SEED user does not have to busy themselves with data cryptography.

The SEED add-on Firefox extension is the interface through which a SEED user can decrypt their email. We show that the encryption protocols that the SEED add-on uses have a minimal performance impact on the operation of the webmail client. Via the SEED add-on, we enable encrypted search of email that matches text in message bodies with near-Gmail accuracy and minor execution overhead. The code for our SEED add-on and our encryption testing apparatus, Email Generator, will be posted on GitHub shortly.

7 References

- [1] Michel Abdalla, Mihir Bellare, Dario Catalano, Eike Kiltz, Tadayoshi Kohno, Tanja Lange, John Malone-Lee, Gregory Neven, Pascal Paillier, and Haixia Shi. Searchable Encryption Revisited: Consistency Properties, Relation to Anonymous IBE, and Extensions. *J. Cryptol.*, 21(3):350–391, March 2008.
- [2] Zachary Vance Aleem Mawani, Omar Ismail. SecureGmail, 2013.
- [3] Ballard, Lucas and Kamara, Seny and Monrose, Fabian. Achieving Efficient Conjunctive Keyword Searches over Encrypted Data. In *Proceedings of the 7th International Conference on Information and Communications Security, ICICS’05*, pages 414–426, Berlin, Heidelberg, 2005. Springer-Verlag.
- [4] Mihir Bellare, Alexandra Boldyreva, and Adam O’Neill. Deterministic and Efficiently Searchable Encryption. In *Proceedings of the 27th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO’07*, pages 535–552, Berlin, Heidelberg, 2007. Springer-Verlag.
- [5] Dan Boneh, Eyal Kushilevitz, Rafail Ostrovsky, and William E. Skeith, III. Public Key Encryption That Allows PIR Queries. In *Proceedings of the 27th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO’07*, pages 50–67, Berlin, Heidelberg, 2007. Springer-Verlag.
- [6] Dan Boneh and Brent Waters. Conjunctive, Subset, and Range Queries on Encrypted Data. In *Proceedings of the 4th Conference on Theory of Cryptography, TCC’07*, pages 535–554, Berlin, Heidelberg, 2007. Springer-Verlag.

- [7] R. Brinkman. *Brinkman PhD Thesis*. PhD thesis, University of Twente, Enschede, June 2007.
- [8] Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. In *INFOCOM, 2011 Proceedings IEEE*, pages 829–837, April 2011.
- [9] Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. In *INFOCOM, 2011 Proceedings IEEE*, pages 829–837, April 2011.
- [10] Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. Privacy-Preserving Multi-Keyword Ranked Search over Encrypted Cloud Data. *Parallel and Distributed Systems, IEEE Transactions on*, 25(1):222–233, Jan 2014.
- [11] Yan-Cheng Chang and Michael Mitzenmacher. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *Proceedings of the Third International Conference on Applied Cryptography and Network Security, ACNS’05*, pages 442–455, Berlin, Heidelberg, 2005. Springer-Verlag.
- [12] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS ’06*, pages 79–88, New York, NY, USA, 2006. ACM.
- [13] Dan Boneh Emily Stark, Michael Hamburg. Symmetric Cryptography in Javascript. In *Annual Computer Security Applications Conference 2009*, 2009.
- [14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP ’03 Proceedings of the nineteenth ACM symposium on operating systems’ principles*, pages 29–43, 2003.
- [15] Philippe Golle, Jessica Staddon, and Brent Waters. Secure Conjunctive Keyword Search over Encrypted Data. In *ACNS 04: 2nd International Conference on Applied Cryptography and Network Security*, pages 31–45. Springer-Verlag, 2004.
- [16] Google. Security Whitepaper: Google Apps Messaging and Collaboration Products, 2011.
- [17] Google. Google Apps Service Level Agreement, 2014.
- [18] Google. Google Data Centers: Data and Security, 2014.
- [19] Google. Standard view and basic HTML view, 2014.
- [20] IETF Network Working Group and P. Resnick. RFC 2822, 2001.

- [21] Mike Hamburg. prediction output length, 2011.
- [22] Ravi Chandra Jammalamadaka, Roberto Gamboni, Sharad Mehrotra, Kent E. Seamons, and Nalini Venkatasubramanian. gVault: A Gmail Based Cryptographic Network File System. In *Proceedings of the 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 161–176, Berlin, Heidelberg, 2007. Springer-Verlag.
- [23] Mark Perry Karthick Ramachandran, Hanan Lutfiyya. Chaavi: A Privacy Preserving architecture for Webmail Systems. In *Proceedings of the 2nd International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011.
- [24] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy Keyword Search over Encrypted Data in Cloud Computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–5, March 2010.
- [25] Ming Li, Shucheng Yu, Ning Cao, and Wenjing Lou. Authorized Private Keyword Search over Encrypted Data in Cloud Computing. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 383–392, June 2011.
- [26] Robert McMillan. How Heartbleed Broke the Internet – And Why It Can Happen Again. *Wired*, 2014.
- [27] Thomas Oberndörfer. Mailvelope project, 2013.
- [28] Christof Paar. Introduction to Cryptography, 2014.
- [29] Doc Sheldon. Can Google be Trusted to Do No Evil?
- [30] Daniel Smilkov, Deepak Jagdish, and César Hidalgo. Immersion, 2013.
- [31] Dawn Xiaoding Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55, 2000.
- [32] Open source community. GNU Privacy Guard, PGP, GnuPG, OpenPGP, 2013.
- [33] The Washington Post staff. NSA slides explain the PRISM data-collection program. *The Washington Post*, 2013.
- [34] Angus Stevenson and Christine Lindberg. *New Oxford American Dictionary*. Oxford University Press, 3rd edition, 2011.
- [35] Vlad Vukicevic, Johnathan Nightingale, Gavin Sharp, and Chris Peterson. Electrolysis.

[36] Cong Wang, Ning Cao, Jin Li, Kui Ren, and Wenjing Lou. Secure Ranked Keyword Search over Encrypted Cloud Data. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 253–262, June 2010.

[37] wbamberg and evold. Content Scripts.