2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)
Allerton Park and Retreat Center
Monticello, IL, USA, September 24-27, 2019

Collaborative Privacy for Web Applications

Yihao Hu
Electrical and Computer Engineering
Boston University
yihaoh@bu.edu

Ari Trachtenberg
Electrical and Computer Engineering
Boston University
trachten@bu.edu

Prakash Ishwar
Electrical and Computer Engineering
Boston University
pi@bu.edu

***Abstract*—Real-time, online-editing web apps provide free and convenient services for collaboratively editing, sharing and storing files. The benefits of these web applications do not come for free: not only do service providers have full access to the users' files, but they also control access, transmission, and storage mechanisms for them. As a result, user data may be at risk of data mining, third-party interception, or even manipulation. To combat this, we propose a new system for helping to preserve the privacy of user data within collaborative environments. There are several distinct challenges in producing such a system, including developing an encryption mechanism that does not interfere with the back-end (and often proprietary) control mechanisms utilized by the service, and identifying transparent code hooks through which to obfuscate user data. Toward the first challenge, we develop a character-level encryption scheme that is more resilient to the types of attacks that plague classical substitution ciphers. For the second challenge, we design a browser extension that robustly demonstrates the feasibility of our approach, and show a concrete implementation for Google Chrome and the widely-used Google Docs platform. Our example tangibly demonstrates how several users with a shared key can collaboratively and transparently edit a Google Docs document without revealing the plaintext directly to Google.**

I. Introduction

Collaborative web applications (apps) such as the Google productivity suite (Docs, Sheets, and Slides) enable multiple users to simultaneously edit a number of common documents. To enable server-side features, such as compression or version control, the contents of these documents are typically available in plaintext to the app provider. As a result, the provider, affiliated third parties, or malicious parties who have infiltrated the provider, may also be able to mine the plaintext for behavioral advertising, social engineering, or even identity theft.

To help preserve their privacy, some users encrypt their data client-side, allowing only users who know a shared private key to read the plaintext. However standard encryption often inhibits the human usability experience [21, 26] and its block or streaming encoding is likely to impair or completely break the collaborative functionality provided by a web service. Likewise, anonymization overlays like Tor [6] or private browsing may only superficially obfuscate the connection between users and data, as the data itself may very well contain deanonymizing features.

In contrast to these existing approaches, we propose a transparent and light-weight encryption layer between clients and providers that protects user data *without* breaking collaborative features. Users with access to the document secret may view and edit the document within the collaborative framework as if no encryption layer is present. On the other hand, users who do not know the document secret, and this may include the app provider, see obfuscated text. This layer is implemented through browser extension and it makes extensive use of the standard `XMLHttpRequest` API [7] used by a variety of web applications (e.g., Google productivity suite, Conceptboard, MeetingWords, Collabedits, Codepen, etc. [1, 3, 5]) to transmit user edits.

Our approach is based on a novel character-level variation of the venerable polyalphabetic substitution cipher [18]. The benefit of encrypting without the need for context, and at the smallest unit of information of many collaborative apps (*i.e.,* one Unicode character), is that our approach maintains functionality and provider bandwidth usage while avoiding heavy-duty reverse-engineering of app-related code or network protocols (which may be obfuscated).

Though the substitution cipher itself is vulnerable to a number of well-known attacks, such as statistical attacks and chosen plaintext attacks [22], we provide approaches for strengthening the cipher through standard mitigations approaches such as homophony and mapping randomization in addition to novel approaches based on range extension. In the latter case, the plaintext is extended from the typically narrow band of the Unicode character space (*e.g.,* those associated with the English and/or Greek alphabets) to the entirety of the Unicode space in a manner than helps equalize character and multi-character distributions in order to complicate statistical attacks. Finally, we demonstrate the effectiveness of our system through the implementation of a Chrome browser extension that showcases its use in preserving privacy for the popular Google Docs collaborative platform.

The following are our **main contributions**:

- We identify a robust mechanism for encrypting/decrypting user data within collaborative environments that utilize the `XMLHttpRequest` API without affecting server-side control traffic.
- We develop and analyze a novel character-level variation of the polyalphabetic substitution cipher that is more resilient to classical attacks on the cipher.
- We concretely demonstrate an integration of the two previous contributions as a prototype privacy-preserving Chrome extension for managing Google Docs.

978-1-7281-3151-1/19/$31.00 ©2019 IEEE

We begin in Section II with a review of some of the related work from the literature. Next we present the architecture of our system in Section III, including descriptions of our software interface, followed by our new character-level encryption scheme in Section IV. Section V describes our prototype together with screenshots of it in action. We conclude in Section VI with some final thoughts, including limitations of our approach.

II. Related Work

We next outline several representative (but hardly exhaustive) approaches to web-based privacy preservation from the literature. Our approach is specifically attuned to online collaborative environments, and our use of a memoryless character-level encryption is thus one key point of departure with the related work below.

a) M-Aegis: M-Aegis [13] aims to protect data from cloud providers by using a transparent window that sits atop an existing application and encrypts input data in transit. Our approach differs from M-Aegis in a number of ways. First, M-Aegis focuses on native Android apps, whereas our approach focuses on browser-hosted apps and is not operating-system dependent. Rather than mimicking portions of an app's interface with a GUI overlay, we hook directly into the web application to intercept user input as transparently as possible.Moreover, in real-time collaborative environments, edits may occur character-by-character and at different locations in a document. Block-based schemes, like those utilized by M-Aegis, require the ability to discern the context of edits in and re-encrypt on the fly.

b) MessageGuard: Another closely related work is MessageGuard [17], which implements a system that layers end-to-end encryption on top of existing web applications, using the browser as a global control point and deploying as either a browser extension or a bookmarklet. MessageGuard uses the `iFrame` HTML element as a middleware overlay between the user and web app, modifying data before it reaches the application. We, on the other hand, intercept data between the application and server, modifying it in transit. In this manner, our users' interaction with the app does not change, meaning that their experience is preserved.

c) Fully-homomorphic Encryption: There are also a number of methods of ensuring data confidentiality with the help of the cloud provider, most notably based on the use of fully-homomorphic encryption (see, for example, [10, 11, 23, 25]), although there is a wealth of additional literature within their citations and reverse-citations).

These methods aim to have the server agnostically compute functions of a user's data, and they are aimed toward an honest-but-curious provider. Our approach does not require any server-side modifications while maintaining transparency to the user or multiple collaborating users. We also avoid the heavy computational machinery required for these schemes.

d) Classical Encryption: More generally, there are quite a number of tools that aim to encrypt user data, as typified by PGP [27] and S/MIME [16]. These tools are all meant for one operating user at a time, rather than collaborating users, and they are not designed up-front to function within the back-end's existing processing methods.

III. System Architecture

Our proposed system has two fundamental components:

1) A **browser interface**, which intercepts and modifies data that enters or leaves the app within the browser. Our specific prototype extension makes use of standard Chrome features to insert interface code between the app and the provider, and, thus, it may be expected to persist over several browser revisions. Indeed, these features are also common in the popular Firefox browser, and our system should be portable to it as well.
2) a character by character **collaborative encryption** scheme that runs within the interface to encrypt and decrypt data streams using an extension of the substitution cipher.

We next present details of our architecture, starting in Section III-A with an overview of our threat model. Section III-B describes the browser interface together with the software hooks that enable it. Thereafter, Section IV describes the collaborative encryption scheme, together with analyses and approaches to strengthening its security.

A. Threat Model

We assume that the collaborating users have an out-of-band method for sharing a common secret key for encryption and decryption, and the strength of our encryption scheme is based on some standard assumptions about the statistical properties of the text being edited (elaborated in Section IV within each relevant subsection), which are known to the attacker.

1) In Scope: Our threat model includes an honest-but-curious cloud collaborative service provider or third party that observes and mines data at rest on the service's servers. Third parties could include attackers with access to the provider's data servers, partners in a business relationship with the provider or law enforcement agencies.

2) Out of Scope: Since we only focus on the collaborative real-time editors, threats to other kinds of app, such as Facebook Messenger and Emails, are not considered in this paper. Moreover, we primarily protect against attacks at rest on the service provider's servers, and thus do not handle:

- *Browser attacks* - we assume that the browser reliably executes both the application and our browser extension, even though the provider might also provide the browser (e.g. Google and the Google Chrome browser).
- *Side-channel attacks* - either by the provider or by a "man-in-the-middle" attacker. These include active attacks based on statistically correlating key-strokes or client-server communication with user activities based on fine-grained timing. We also do not consider information leakage from formatting, style, table structure, or other "area affects", and, instead, focus on text alone. We believe that these side-channel attacks should be addressed by orthogonal mechanisms.

- *The client-side app* - although we do not assume the app is trustworthy, we do assume that the implementer of our framework can reverse engineer the application's protocols to the level of identifying the paths through which input data is transported. Our approach does not cover providers maintaining concealed channels for transferring this data or encrypted metadata, which we would not be expected to access.

B. Browser Interface

Our approach uses a browser extension-based content script [2, 4] to inject JavaScript payloads into web applications. The payloads hook specific functions of JavaScript objects that serve as interfaces for app data. With hooks in place, we can filter and modify the data, which contains event messages from an app's proprietary protocol.

We have implemented our framework as a browser extension that provides application data interception and modification functionality for the Google Chrome browser. Content scripts typically can access the Document Object Model (DOM) of targeted pages, but cannot use variables or functions defined by web pages or by other content scripts [2]. However, by utilizing features in the environment of the browser, we are able to interact with web scripts and implement hooks on typical sources and sinks of web application data.

The ability to run code on a web page is only part of the challenge of collaborative encryption. Editors that are not HTML-backed editors (like the Google Docs framework). often generate their client graphical interface through obfuscated JavaScript. As such, a successful prototype must also identify an appropriate hook through which to intercept communications between the client and the service provider. When these frameworks use the `XMLHttpRequest` Application Program Interface (XHR), however, one may pick out and overwrite the `XMLHttpRequest.send` and `XMLHttpRequest.open` methods to intercept and modify the entire client-provider data stream.

We next describe some of the details involved with this prototype implementation, stressing that our content script hooking exploits a stable feature of the browser (dating back at least to at Chrome 9.0, circa 2011).

1) Content Scripts: Content scripts run JavaScript code within a specific web page context. In Chrome, these scripts may be injected either before the DOM is constructed (`document_start` mode), after the DOM is complete (`document_end` mode), or right after the window's `onload` handler is called (`document_idle` mode) [2]. An enabling feature of these scripts is that, in `document_start` mode, they can insert code before the DOM is constructed. The result is that, by design, the inserted code overshadows corresponding methods that are loaded through the web page.

As a general template, the script modeled in Figure 1 can be injected into a web page, before the DOM is constructed, to overshadow an existing `overrideFunction`. In our prototype example, we combine two payload injections into a Google Docs page to produce an encryption middleware:

- **Outgoing Payload**
  - A script that overwrites `XMLHttpRequest.send`.
- **Incoming Payload**
  - A script that overwrites `XMLHttpRequest.open` and decrypts (with a user-supplied key) initial page content that has been retrieved from the service provider.

2) Outgoing Payload: The outgoing interception payload queries the user for an encryption key and then injects a JavaScript snippet similar to that in Figure 2 into the DOM of the underlying page. When this snippet redefines the `XMLHttpRequest.send` method, the new method is subsequently applied to *all* XMLHttpRequest uses and is executed every time `XMLHttpRequest.send` is called. In effect, the overshadowing method acts as a "man in the middle" and allows direct access to the outgoing data so that it may be viewed and modified before being sent out with the original `XMLHttpRequest.send`.

The `outgoing_data` is sent in an incremental fashion, every time changes (such as keystrokes or formatting modifications) are made within the editing window. In our current prototype, we focus only on modifying keystrokes.

3) Incoming Payload: The incoming payload requests a decryption key from the user, and then initially decrypts the current state of the document from the service backend using the JavaScript snippet resembling Figure 3. This code is based on the observation that, rather than using

```
var code = overrideFunction() {
    ...//Payloads to be injected
};
var script = document.createElement("script");
script.textContent = "(" + code + ")();";
(document.head ||
  document.documentElement).appendChild(script);
```

Fig. 1. Injection template.

```
XMLHttpRequest.prototype.realSend =
    XMLHttpRequest.prototype.send;
var newSend = function(outgoing_data) {
  if (outgoing_data.contain(new_entered_chars)){
    encrypt_algorithm(outgoing_data.
        new_entered_chars, key);
  }
  this.realSend(outgoing_data);
};
XMLHttpRequest.prototype.send = newSend;
```

Fig. 2. Outgoing data interception and modification.

```
  Object.defineProperty(this, "target", {
      get: function(){
          var text = saved_file;
          decrypt(text, key);
          return text;
      },
      set: function(val){
          saved_file = val;
      }});
```

Fig. 3. Decoding data stored on the service.

```
var realOpen = XMLHttpRequest.prototype.open;
var newOpen = function(){
  Object.defineProperty(xhr, "responseText", {
    get: function(){
        if (xhr.readyState===4){
          var incoming_data = xhr.response;
          if (incoming_data.contains(new_inputs))
          decrypt(incoming_data.new_inputs);
          return incoming_data;
        } }
  });
  realOpen.apply(this, arguments);
};
XMLHttpRequest.prototype.open = newOpen;
```

Fig. 4. Decoding incoming user updates.

`XMLHttpRequest.open` to access the existing document, Google Docs loads the document content into a page property, and our redefinition of the getter function of this property allows the Server-stored ciphertext to be intercepted and decrypted into plaintext before being displayed.

Once the initial state has been established, the incoming payload decrypts updates incoming content from the provider in the fashion of Figure 4. Similarly to the overshadowed `XMLHttpRequest.send` in the outgoing payload, this snippet acts as a "man in the middle" to intercept and decrypt incoming data, where `incoming_data` here carries only updates to the document. The only difference with the overshadowed `XMLHttpRequest.send` is that the incoming data is loaded into the property `responseText`, from which the web app loads updates to the editing window. Therefore, once the getter method is redefined, the incoming data can be successfully intercepted, identified, and modified before being returned to the web app for further processing.

IV. Collaborative Encryption

Traditional encryption schemes aim to "confuse and diffuse" a plaintext [20] into a ciphertext, so that a small perturbation in the plaintext produces an unpredictable "avalanche" [9] of changes in the ciphertext. As an overlay for a collaborative system, however, this model has some significant drawbacks.

Consider, for example, several users editing a shared document online. If one user changes an "e" to an "a" somewhere in the document, the cloud back-end propagates only this change to the other users, and not an entirely new copy of the document, in order to limit communication overhead. From the perspective of an overlay, however, if the one letter change completely affects an encryption block, then, in effect, the back-end must update all users with the entire block that was changed upon every edit, and the collaboration is very inefficient.

As such, for our platform we seek a "locally-encodable" encryption scheme that manages two seemingly contradictory demands:

1) Minimize the number of ciphertext characters affected by a small change to the plaintext.
2) Make it difficult to determine a plaintext, or even parts of a plaintext, from a given ciphertext.

The second demand is typical of encryption protocols, and can be defined in a number of ways, most notably based on computational or information-theoretic assumptions. The first demand is specific to our collaborative context, and we next develop several approaches for meeting it.

The overarching basis for our approach will be the classical substitution cipher, as formalized and described in Section IV-A. It is well known that the substitution cipher leaks statistical information about its plaintext and is also not robust to a (chosen or known) plaintext attack. For the issue of statistical leakage, we propose two approaches based on spreading the plaintext alphabet over a larger ciphertext alphabet. In Section IV-B, we consider the approach of apportioning the plaintext alphabet into many equal-sized blocks in the ciphertext alphabet, a practical scheme with a challenging analysis. In Section IV-C, on the other hand, we evaluate apportioning the plaintext into varying-sized blocks in the ciphertext alphabet, resulting in a more complicated implementation with a simpler analysis. Finally, in Section IV-D we consider mitigations for plaintext attacks.

A. *Substitution - a simple approach*

We formalize the first demand of our locally-encodable encryption as Definition 1, based on an encryption function $\mathfrak{E}_k : \Sigma^* \longrightarrow \Sigma^*$ indexed by a key string $k \in \Sigma^*$ (which is the encryption secret shared by collaborating users) and mapping plaintext strings over an alphabet Σ into ciphertext strings over the same alphabet.

Definition 1. *An encryption function $\mathfrak{E}$ is* locally-encodable *if, for some constants $c \geq 1$ and all $k, s, s' \in \Sigma^*$,*

$$\delta\left(\mathfrak{E}_k(s), \mathfrak{E}_k(s')\right) \leq c\, \delta(s, s'),$$

where $\delta(x, y)$ is the Levenshtein edit distance metric [15], denoting the minimum number of insertions, deletions, and/or single character transpositions needed to transform string x into string y.

When c is the ciphertext length, Definition 1 generalizes any deterministic, fixed-length encryption algorithm . Likewise, $c < 1$ is disallowed because it prohibits unique decryption, in that two different plaintexts might map to the same ciphertext.

From the perspective of interfacing cleanly with existing collaborative environments, we desire a position encryption is one that is position independent.

Definition 2. *Encryption $\mathfrak{E}$ is* position-independent *if:*

$$\mathfrak{E}_k(s) = \mathfrak{E}_k(s_0) + \mathfrak{E}_k(s_1) + \mathfrak{E}_k(s_2) + \ldots + \mathfrak{E}_k(s_n),$$

where $+$ denotes concatenation and $s = s_0 + s_1 + \ldots + s_n$.

Position-independent encryptions are useful because they can be calculated without needing to consider the entire text. More precisely, the encryption can be calculated based on individual characters being edited.

Consider, for example, a plaintext $s =$ `philosophical` in a document that is being edited collaboratively, and suppose the string is encrypted with a position-independent

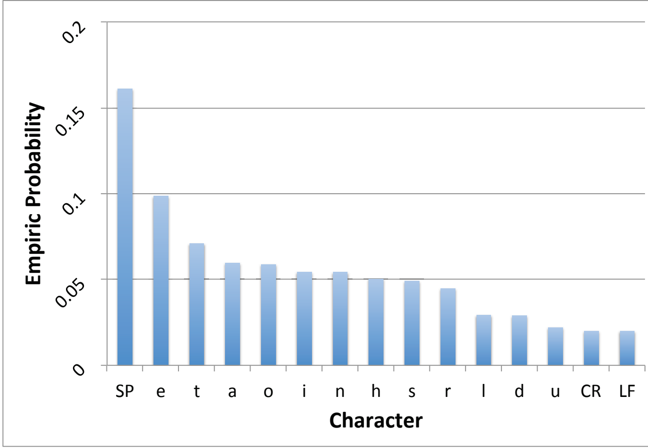


Fig. 5. Histogram of characters in Gutenberg's translation of Mark Twain's "Tom Sawyer". SP, CR, and LF denote a space, carriage-return, and linefeed, respectively.

scheme as $\mathfrak{E}_k(s)$ = KSROLHLKSRXZO on the server. Changing the plaintext by transposing the "c" to a "g" and deleting first "h" corresponds to modifying the ciphertext by transposing $\mathfrak{E}_k(\mathrm{c})$ = X to $\mathfrak{E}_k(\mathrm{g})$ = T and deleting $\mathfrak{E}_k(\mathrm{h})$ = S. The implementational consequence of this is that, when looking at the event messages that are being sent between users, we just need to identify the actual letters being transmitted, and not other metadata, such as their position in the text.

It is not hard to see that the simple *substitution cipher*, which maps strings based on a one-to-one correspondence between input and output character spaces, is an encryption scheme that is both locally-encodable ($c = 1$) and position-independent. One of its well-known drawbacks is that, despite the theoretically large work-factor to break it (95! for printable ASCII characters), the cipher readily yields to classical statistical analysis, since it preserves the character distribution of its source and leaks exact information about where a given string is being changed.

Figure 5 shows a histogram of the characters found in Mark Twain's *The Adventures of Tom Sawyer* [24] as a baseline for subsequent examples. One can clearly see the dominance of characters such as the space (SP) and letter e, which can thus be identified in the substitution-encrypted text.

B. Alphabet Extension

To combat frequency analysis, it is possible to embed the range of usable characters (say, the 95 printable ASCII characters) within the larger Unicode space that is supported by many web applications. For JavaScript engines, it is convenient to use the Unicode characters in the range 0x0020 to 0xD7FF, since many JavaScript engines encode strings as sequences of 16-bit Unicode Transformation Format (UTF-16) code units, where each character is represented by a single code unit.

1) Encryption: One way of achieving this embedding is to divide the available Unicode region into non-intersecting 95-character blocks (corresponding to printable ASCII characters), and assigning to each block a pseudorandom permutation seeded by the block's ID and the encryption key (*i.e.,* the password shared by the various users). To encrypt a printable character, one uniformly randomly picks a 95-character block from the Unicode range, and uses the corresponding permutation to produce a Unicode character.

As an example, consider the extension algorithm encoding a plaintext character b. The user picks a 95-character Unicode block in a random (and not necessarily reproducible) way; in this case, she may choose the second block, with Unicode characters in the range 0x007F-0x00DD. A concatenation of the block ID (2) and a shared password is then used to seed a pseudorandom number generator (PRNG) that produces a permutation of the Unicode characters in the range; there are a number of well-known and efficient methods for producing such a random permutation of k integers, dating back (at least) to Hall and Knuth (see [12, 14, 19] for some implementations). Since our plaintext b is the 66th printable ASCII character, we replace it with the 66th element of our pseudorandom permutation, in this case <<, which is our ciphertext character.

2) Decryption: To decrypt a ciphertext, a second (authorized) user identifies the Unicode block in which the encrypted character is found, and seeds a PRNG with a concatenation of the resulting block ID and the shared password. This PRNG is then used to produce a pseudorandom permutation, the same one produced by the encrypting user, which is inverted to produce the original plaintext character.

In our earlier example, the second user would identify ciphertext << as belonging to Unicode block 0x007F-0x00DD, which has block ID (2). She would concatenate this ID with her shared password to seed a PRNG and produce the permutation found in that block on the figure. The permutation is a one-to-one correspondence between printable ASCII characters and Unicode characters in the range, so it is readily inverted to produce the plaintext b.

3) Unigram analysis: This approach naturally increases the entropy of the resulting ciphertext over simple substitution, as expressed by the following straightforward lemma (we use the notation $\mathcal{M}$ to denote the plaintext alphabet, and $\mathcal{C}$ to denote the ciphertext alphabet).

Lemma 1. *Extending the base alphabet from* $|\mathcal{M}|$ *to* $|\mathcal{C}|$ *characters in this manner increases character entropy by* $\log_2\left(\frac{|\mathcal{C}|}{|\mathcal{M}|}\right)$ *bits.*

The plaintext unigram entropy is given by $H(P) = -\sum_{i\in\mathcal{M}} p_i \log_2(p_i)$. The encoding process uniformly distributes characters among the $\gamma = \frac{|\mathcal{C}|}{|\mathcal{M}|}$ blocks, meaning that the probability of seeing a character ϵ corresponding to i (according to the random permutation of its block) in the ciphertext alphabet is $\Pr(\epsilon) = \frac{p_i}{\gamma}$. Computing the resulting entropy of the ciphertext C produces:

$$H(C) = -\sum_{\epsilon\in\mathcal{C}} \Pr(\epsilon)\log_2\left(\Pr(\epsilon)\right) = -\gamma\sum_{i\in\mathcal{M}} \frac{p_i}{\gamma}\log_2\left(\frac{p_i}{\gamma}\right)$$
$$= H(P) + \log_2(\gamma).$$

In the specific case of graphic Unicode characters consistently accessible via JavaScript (i.e., 0x0020 to 0xD7FF),

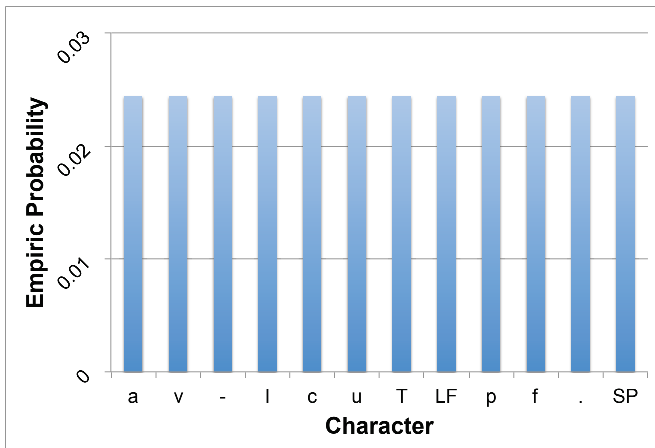


Fig. 6. Histogram of plaintext characters mapping to Unicode characters `0x18DF0 - 0x18E4F` in the *entropy maximization* cipher.

we add roughly 9 bits of entropy to the ciphertext. Extending to $1,112,064$ valid code points of UTF-8 provides approximately 13 bits of extra entropy.

In the sorted empirical entropies of the blocks produced by this extended encoding of "Tom Sawyer". For example, all blocks have a reasonably high entropy within roughly 0.2 bits of the input text's entropy of 4.61 bits, and the overall ciphertext entropy is roughly 13.73, which is about 9 bits more than the input text entropy.

The problem with this scheme becomes evident by examining the histogram of input characters mapped to a specific block, wherein uneven distribution of plaintext characters may carry over to ciphertext characters; for example, one may readily identify that the most common ciphertext characters will be mapped from a space character and the letter "`e`".

4) Greedy Entropy Maximization: Within the alphabet extension scheme of Section IV-B there is flexibility about which Unicode block range to use in producing a ciphertext character. Though a random choice produces a high overall entropy, it might be more advantageous to flatten the histogram of characters mapped into each block.

A joint optimization of entropy across the entire plaintext message is inappropriate for our application, which requires online encryption one character at a time, but a greedy entropy maximization is feasible. In this approach, we maintain the histograms of each Unicode block in memory. When a new character needs to be mapped, we consider its effect on the entropy of each Unicode block and add it to the block for which it most raises the entropy, breaking ties uniformly at random. This has the effect of significantly flattening the distribution of characters.

Indeed, Figure 6 shows that the first 200k of characters from the same text under heuristic produces a flat distribution for a given block , with correspondingly high entropies for each block.

a) Local decodability for $c > 1$: By increasing the constant c in the locally-decodable definition, it is possible to even further reduce the amount of information leaked by an edit, at the expense of significantly increasing the implementational complexity of the system. In this scenario, one user edit results in a constant number of edits in the ciphertext. A consequence of this approach is that the overlay has to be able to produce edit events *de novo*, which is fragile to updates in the underlying web application.

b) Security: Our encryption scheme relies on two elements for its security. First, by greedily maximizing entropy, we end up significantly flattening the distribution of characters mapped into a given block, complicating character-based frequency analysis. Second, the pseudo-random permutation choices per block provide some separability, in that decoding the permutation of one block does not directly lead to the decoding of another block (although it may provide side-information with which to mount an attack).

C. Information Theoretic Optimization

Thus far, we have utilized fixed-length blocks, each encrypting (through substitution) the entire range of usable plaintext characters and a heuristic greedy entropy maximization method to flatten the block-wise unigram distribution. However, variable-length blocks, with lengths adapted to the probability distribution of plaintext characters, can provide even better defense against statistical attacks because the unigram distribution of ciphertext characters can be made arbitrarily close to uniform.

1) Variable-length block algorithm: Let $\mathcal{M}$ denote the plaintext alphabet of size m. Each plaintext character $i \in \mathcal{M}$ is first mapped (independently of other plaintext characters) uniformly at random to a ciphertext character $X_i \in \mathcal{C}_i$, where $\mathcal{C}_1, \ldots, \mathcal{C}_m$ are m disjoint ciphertext sub-alphabets (one for each plaintext character) and $\mathcal{C} := \cup_{i=1}^{m} \mathcal{C}_i$ is the entire ciphertext alphabet. Let $\sigma_{\mathcal{C}}$ denote a permutation on $\mathcal{C}$. Our randomized homophonic substitution cipher can be described by the encryption function which $i \longrightarrow \sigma_{\mathcal{C}}(X_i)$, which is invertible with knowledge of $\sigma_{\mathcal{C}}$. Here, the permutation represents a shared secret key which is available to both the encryption and decryption algorithms, but not the attacker.

In order to simplify the exposition, we shall assume that the plaintext stochastic process $Z_1, \ldots, Z_n$ is first-order stationary, meaning that the unigram (i.e., marginal) distribution of individual plaintext characters is the same at all positions within the plaintext sequence. While this assumption may not hold exactly in practice, it is a fairly weak technical assumption to make since it still allows the process to be non-stationarity (of higher orders) and also have strong temporal dependencies (memory). Moreover, it can be made to hold to any desired degree by encrypting a suitably long sequence of consecutive plaintext characters at once as a group and permuting the *sequential ordering* of characters within the group using another shared secret key. For simplicity, however, we will assume that the first-order stationarity condition holds without such grouping and sequential ordering permutation.

2) Unigram distribution: Since the encryption process operates in a character-wise and statistically time-invariant manner, the ciphertext character process is also first-order stationary. If the unigram (first-order) probability mass func-

tion (pmf) of the plaintext is $\Pr(Z_1 = i) = p_i$, $i \in \mathcal{M}$, then the unigram (first-order) pmf of the ciphertext is given by $q_j := \Pr(\sigma_{\mathcal{C}}(X_{Z_1}) = j) = \frac{p_i}{|\mathcal{C}_i|}$, for all $i \in \mathcal{M}$ and $j \in \sigma_{\mathcal{C}}(\mathcal{C}_i)$.

$$\forall i \in \mathcal{M} \text{ and all } j \in \sigma_{\mathcal{C}}(\mathcal{C}_i), q_j := \Pr(\sigma_{\mathcal{C}}(X_{Z_1}) = j) = \frac{p_i}{|\mathcal{C}_i|}.$$

This is because in order to get ciphertext character j, the plaintext character i that corresponds to it must be generated (this happens with probability p_i) and then the particular ciphertext character within the bin $\mathcal{C}_i$ from which j arises (under permutation $\sigma_{\mathcal{C}}$) must be picked (this happens with probability $1/|\mathcal{C}_i|$).

Proposition 1. *If the unigram pmf over plaintext characters and the ciphertext sub-alphabet sizes are such that for all $i \in \mathcal{M}$, $|\mathcal{C}_i| = p_i \cdot |\mathcal{C}|$, then the unigram distribution of ciphertext characters is exactly uniform over the ciphertext alphabet, i.e., $q_j = 1/|\mathcal{C}|$ for all $j \in \mathcal{C}$.*

If the plaintext unigram probabilities p_i are all rational numbers, then the ciphertext unigram probabilities q_j can be made *exactly* uniform over the ciphertext alphabet $\mathcal{C}$ using a sufficiently large, but finite, $|\mathcal{C}|$. In practice, the plaintext unigram probabilities would be estimated empirically as normalized character counts (frequencies) in some corpus of documents. The estimated probabilities would therefore be rational numbers. If, on the other hand, even one p_i is irrational, exact uniformity cannot be attained with any finite $|\mathcal{C}|$. However, one can always approximate any irrational fraction with a rational one with a sufficiently large denominator. Thus, the ciphertext unigram distribution can be made as close to uniform as desired by making $|\mathcal{C}|$ sufficiently large. In practice, if $p_i \cdot |\mathcal{C}|, i \in \mathcal{M}$ are not integers, we would drop the fractional parts and distribute (in some manner) any remaining characters in the ciphertext alphabet among the plaintext alphabet characters.

In different scenarios, the plaintext distribution may be known to both the users and the attacker, or only to the various users, or to none. Similarly, the *sizes* of the ciphertext sub-alphabets $|\mathcal{C}_1|, \ldots, |\mathcal{C}_m|$ may be known to both the users and the attacker or only the users. However, the overall ciphertext alphabet $\mathcal{C}$ will be known to both the users and the attacker. In what follows, we assume that the users know everything and with the exception of the secret key, the attacker also knows everything.

If the unigram ciphertext distribution q_j can be made exactly uniform, then no statistical test based only on observed ciphertext (ciphertext-only attack) will be able to tease the plaintext characters apart (with confidence better than a random guess) using only a unigram frequency analysis. On the other hand if the q_j's are not exactly uniform, and they are all distinct for different j and known to the attacker, then as n becomes very large, a ciphertext-only attack may be able break the cipher with overwhelming probability. However, the closer that the q_j's are to being uniform, the longer that the attacker will have to wait to gather enough ciphertext characters before being able to break the cipher with sufficient confidence.

3) Unigram sample complexity analysis: In order to gain quantitative insight into how long the ciphertext needs to be before it can be broken with some desired degree of confidence via unigram analysis and how this minimum length increases with increasing ciphertext alphabet size $|\mathcal{C}|$, we consider the following simpler task for the attacker: in a binary plaintext alphabet $\mathcal{M} = \{0, 1\}$, decide whether a particular ciphertext character j corresponds to the plaintext character 0 or the plaintext character 1. Let n_j denote the number of ciphertext characters that equal j in a message of length n. For analytical tractability, here we will assume that the plaintext process is stationary and memoryless, i.e., it is a sequence of independent and identically distributed (iid) characters. Then n_j will have a binomial distribution for n trials with success probability equal to q_0, if j corresponds to plaintext character 0, and success probability q_1, if j corresponds to plaintext character 1. The attacker's task of deciding 0 or 1 based on n_j and knowledge of n, q_0, q_1 is a simple Bayesian binary hypothesis testing problem that has been extensively studied in the literature.

Indeed, for sufficiently large n, the error probability $P_e(n)$ of the optimum plaintext decoding (i.e., based on the Maximum A posteriori Probability [MAP] rule) approximates $e^{-nD(r||q_0)}$, where $D(u||v)$ denotes the Kullback-Leibler (KL) divergence [8, Section 11.9]. We have the following result whose proof may be found in Section 11.9 of [8].

Proposition 2. *[8] If $\mathcal{M} = \{0, 1\}$ and the plaintext process is stationary and memoryless, then for each ciphertext character $j \in \mathcal{C}$, the error probability $P_e(n)$ of the optimum plaintext decoding rule (*i.e., *, the MAP rule) goes to zero exponentially fast with the ciphertext size n:*

$$\lim_{n\to\infty} -\frac{1}{n} \ln P_e(n) = D(r||q_0) = D(r||q_1)$$

where

$$r(q_0, q_1) := \frac{\ln(1-q_0) - \ln(1-q_1)}{\ln(1-q_0) - \ln(1-q_1) + \ln q_1 - \ln q_0} \in [0, 1]$$

is a probability and $D(u||v) := u \ln(u/v) + (1-u)\ln((1-u)/(1-v))$ denotes the Kullback-Leibler (KL) divergence between the binary probability distributions $(u, (1-u))$ and $(v, (1-v))$.[1]

Thus for all n sufficiently large, $P_e(n) \simeq e^{-nD(r||q_0)}$. In order to achieve a target decoding error probability of ϵ or less, we require a message of length $n \geq n_{|\mathcal{C}|} = (\ln(1/\epsilon))/D(r(q_0, q_1)||q_0)$ characters. If there was no ciphertext alphabet expansion, i.e., $|\mathcal{C}| = |\mathcal{M}| = 2$, then the minimum number of samples needed to attain a decoding error probability of ϵ is given by $n_{|\mathcal{M}|} = (\ln(1/\epsilon))/D(r(p_0, p_1)||p_0)$. Therefore, for each ϵ, we need $\frac{n_{|\mathcal{C}|}}{n_{|\mathcal{M}|}}$ times more ciphertext samples compared to the case when there is no alphabet expansion.

[1] To be technically precise, $D(u||v)$ is finite if $u = 0$ (resp. 1) whenever $v = 0$ (resp. 1) and is infinity otherwise. Also $0 \ln(0)$ is treated as zero.

Theorem 1. *If $|\mathcal{M}| = 2$ and the plaintext process is iid, then the ratio of the length of ciphertext needed to break the cipher (via unigram analysis) with ciphertext alphabet expansion to the length needed without alphabet expansion is given by:*

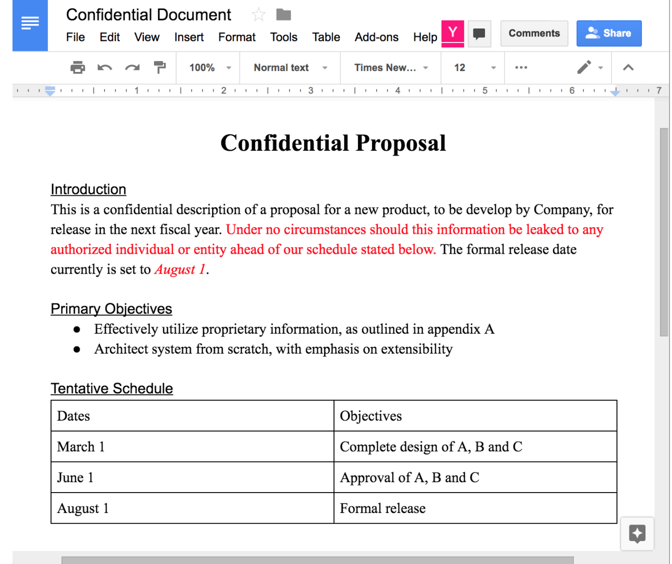
$$\frac{n_{|\mathcal{C}|}}{n_{|\mathcal{M}|}} = \frac{D(r(p_0, p_1)||p_0)}{D(r(q_0, q_1)||q_0)}.$$

4) Example: Consider as a toy example the non-uniform plaintext unigram distribution given by $p_0 = 0.1$ and $p_1 = 1 - p_0 = 0.9$, and the ciphertext alphabet size is $|\mathcal{C}| = 512$ (giving a ciphertext to plaintext size ratio equal to that of UTF-16 Unicode to ASCII). Then taking $|\mathcal{C}_0| = \lceil 0.1 \times 512 \rceil = 52$ and $|\mathcal{C}_1| = 512 - |\mathcal{C}_0| = 460$, we get $q_0 = 0.1/52 = 1.9231 \times 10^{-3}$ and $q_1 = 0.9/460 = 1.9565 \times 10^{-3}$ which is more uniform. Of course, in this particular example if $|\mathcal{C}|$ was a multiple of 10, then the distribution would be exactly uniform, i.e., $q_0 = q_1 = 1/|\mathcal{C}|$ and the cipher will be unbreakable even via multigram analysis (for an iid plaintext process). Continuing, we have $r(p_0, p_1) = 0.5, D(r(p_0, p_1)||p_0) = 0.5108$ and $r(q_0, q_1) = 1.9398 \times 10^{-3}, D(r(q_0, q_1)||q_0) = 7.2221 \times 10^{-10}$. This makes $\frac{n_{|\mathcal{C}|}}{n_{|\mathcal{M}|}} = 7.0731 \times 10^6$, i.e., the ciphertext length needed to break a single character (at any confidence level) with a 256-fold ciphertext alphabet expansion is about 7 million times that needed to break a single character (to the same confidence level) without ciphertext alphabet expansion. Specifically, for $\epsilon = 0.01$ (99% confidence), $n_{|\mathcal{M}|} = 9$ and $n_{|\mathcal{X}|} = 63$ million. We would like to emphasize that these numbers are just for the toy example where the plaintext alphabet has only two characters and the unigram distribution of the two characters is highly non-uniform. These numbers can be expected to be much more larger in practice because typical plaintext alphabet sizes are much larger than 2 (95 for ASCII) and the unigram plaintext distribution is much less skewed.

D. Plaintext Attacks

A known or chosen plaintext (with a corresponding ciphertext) significantly reduces the complexity of breaking a substitution cipher by providing some of the plaintext-ciphertext substitutions that form the encryption key; language and context may be used to infer the remaining substitutions. Utilizing a polyalphabetic cipher, as described in this work, improves the resilience of the cipher, since less information is revealed with each substitution. In other words, if the letter character `a` is mapped uniformly at random to one of ten ciphertext characters, then revealing one of these plaintext-ciphertext connections only reveals one tenth of the occurrences of `a`. This mapping can be modified in some coarse manner, say based on the month in which the text is produced or the name of the original author of the work, in order to limit the usability of known plaintexts.

The cipher can be made even more resilience by encrypting one plaintext character with more than one ciphertext character, and, indeed, this does not break our collaborative encryption model or our prototype implementation, although



Confidential Proposal

Introduction

This is a confidential description of a proposal for a new product, to be develop by Company, for release in the next fiscal year. Under no circumstances should this information be leaked to any authorized individual or entity ahead of our schedule stated below. The formal release date currently is set to *August 1*.

Primary Objectives

- Effectively utilize proprietary information, as outlined in appendix A
- Architect system from scratch, with emphasis on extensibility

Tentative Schedule

Dates	Objectives
March 1	Complete design of A, B and C
June 1	Approval of A, B and C
August 1	Formal release

Fig. 7. A collaborator running our browser extension (with correct password) edits a Google Doc. The app functions normally with its collaborative features. However, data specifying the document's text contents is intercepted and modified in transit to/from Google's servers according to our encryption. Only users utilizing the correct password can view the original contents.

it is possible that location-sensitive processing could suffer. Though character-level encryption is inherently weaker than block- or stream-based encryption, we stress that the proposed approach provides a measurable increase in privacy, where currently none exists, without requiring server-side or browser rewriting, and that the encryption can be strengthened further at the expense of efficiency.

V. Prototype

We demonstrate our proposed encryption framework through the Google Docs platform on the Google Chrome. Google Docs is a collaborative document editing service provided by Google. Two or more users can edit a document's state (*i.e.,* text, formatting, and figures) together in real-time using only a modern web browser. Data resides on Google's servers, and any collaborative edits pass through Google's infrastructure before being forwarded to other collaborators.

A. Mechanism

For Google Docs, document data is structured as a series of event messages, each of which has an associated opcode and a set of fields specific to that opcode. The client-side app parses these messages, which specify document contents, layout, and formatting, to render a document for the user.

With our architecture we were able to intercept both sources of information using the techniques in Section III-B. First our implementation involves hooking the `XMLHttpRequest` prototype for all frames originating from Google's relevant servers, providing access to the incoming and outgoing XHR data streams. By examining the effects on document state of messages with particular opcodes, we found that events with opcode `is` specify one or more character insertions. By filtering for `is` events, we captured collaborative edits of a document's text contents.

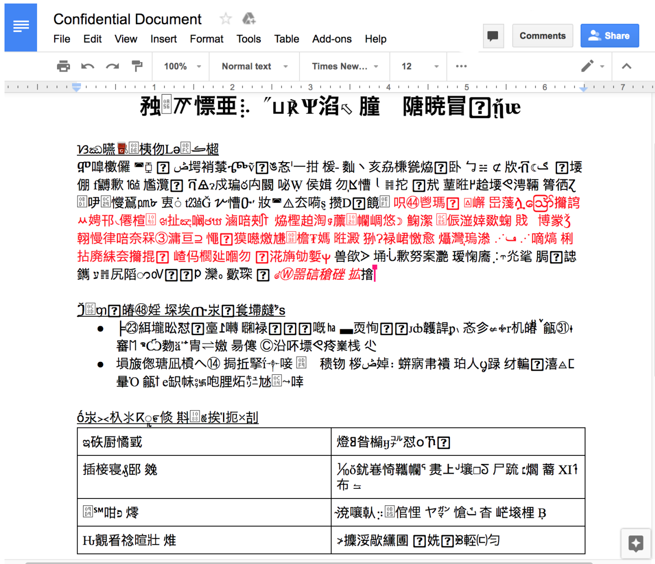


Fig. 8. An undesired third party (or the provider) views the Google Docs document as a guest without a correct private key. Since the document has been configured to allow public viewing, Google Docs permits access. However, in the absence of the extension and a valid password, document text is indecipherable to the third party.

We then exploited the `document_start` feature to gain control of the `DOCS_modelChunk` variable at app load-time and inspected the series of event messages it contained. We found that these messages captured the cumulative document state of all previous collaborative edits. The messages followed the same format as those of the XHR stream, but state changes were combined across messages where possible. Filtering for `is` events again provided enough specificity to capture all document state information concerning text contents, because messages specifying formatting state such as text size, color, and layout have an opcode other than `is`.

B. Performance Evaluation

We focused our evaluation of the project demonstration on users' experience in both qualitative and quantitative way.

Qualitatively, our prototype extension does not modify formatting (bold type, font size, line spacing, etc.) as well as the Google Docs GUI. This enables users to edit and collaborate as if the encryption layer does not exist. The only discrepancy is spelling corrections (which are handled server-side) are disabled as servers only store ciphertext. If the user mistypes a word, the word would not have an red underline or any suggestion for correction.

On the quantitative side, we measured the delays our extension would cause due to encryption, recording the time between the start and the end of overridden XMLHttpRequest call. The resulting delay is linearly proportional to the number of characters being encrypted, fitting the equation $t = 0.0017n + 6.0876$, where n is the number of character edits, and time t is measured in milliseconds and measures both outgoing and incoming data stream. Since large numbers of character edits happen only during the process of copy-and-paste and loading of the page, most users will experience an average of $6ms$ delay during their active editing. Because the delay is not significant, we conclude that such delay does not affect the overall users' experience.

VI. Conclusions

We have presented a client side encryption system for real time collaborative editing web app. The system consists of an encryption interface as well as a novel variant of the polyalphabetic substitution cipher, designed to seamlessly encrypt and decrypt data without interfering with app functionality or the users' experience within the web app. In this way, the users' data privacy is preserved both during transmission and at rest with the provider.

We have implemented a prototype of our system for the Google Docs collaborative word processing web application within the framework of the Google Chrome web browser. We believe that our choice of prototyping framework serves as a reasonable template for other major browsers and web apps, and that our design can be straightforwardly extended to any real-time collaborative editor which uses the standard `XHR` interface for client-server communication, including such well-used products as the Google productivity suite (Docs, Slides, Sheets), Conceptboard, MeetingWords, Collabedit, Codepen[1, 3, 5].

For potential extensions to this work, minor modifications need to be made in order for the extension to work with different browsers. Moreover, an automated process of recognizing formatting of event messages can be developed in order to avoid manual analysis for different web apps. Finally, with respect to encryption, a motivated provider could identify and blacklist features of our approach, such as the utilization of a broad range of the Unicode spectrum.

Acknowledgments

The authors acknowledge John Navon and John Moore for work on an earlier versions of this work, and Manuel Egele for early proofreading and feedback. This work was also supported, in part, by the National Science Foundation under Grant No. CCF-1563753.

References

[1] Conceptboard - visual project collaboration made easy. http://conceptboard.com/.
[2] Content Scripts. https://developer.chrome.com/extensions/content_scripts.
[3] Meetingwords: Collaborative text editing. http://meetingwords.com.
[4] Mozilla development network: Content scripts. https://developer.chrome.com/extensions/content_scripts.
[5] Online text editor - collabedit. http://collabedit.com.
[6] Tor: Anonymity online. https://www.torproject.org/.
[7] Xmlhttprequest living standard. https://xhr.spec.whatwg.org/.
[8] T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012.
[9] H. Feistel. Cryptography and computer privacy. *Scientific american*, 228:15–23, 1973.
[10] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.
[11] S. Halevi and V. Shoup. Helib-an implementation of homomorphic encryption, 2014.
[12] M. Hall and D. E. Knuth. Combinatorial analysis and computers. *American Mathematical Monthly*, pages 21–28, 1965.

[13] B. Lau, S. Chung, C. Song, Y. Jang, W. Lee, and A. Boldyreva. Mimesis Aegis: A Mimicry Privacy Shield–A System's Approach to Data Privacy on Public Cloud. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 33–48. USENIX Association, 2014.
[14] D. H. Lehmer. Teaching combinatorial tricks to a computer. In *Proc. Sympos. Appl. Math. Comb. Anal.*, volume 10, pages 179–193, 1960.
[15] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.
[16] B. Ramsdell. S/mime version 3 message specification. 1999.
[17] S. Ruoti, K. Seamons, and D. Zappala. Layering Security at Global Control Points to Secure Unmodified Software. In *2017 IEEE Secure Development Conference*, pages 42–49. IEEE, 2017.
[18] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons, 2007.
[19] R. Sedgewick. Permutation generation methods. *ACM Computing Surveys (CSUR)*, 9(2):137–164, 1977.
[20] C. E. Shannon. Communication theory of secrecy systems*. *Bell system technical journal*, 28(4):656–715, 1949.
[21] S. Sheng, L. Broderick, C. A. Koranda, and J. J. Hyland. Why Johnny still can t encrypt: evaluating the usability of email encryption software. In *Symposium On Usable Privacy and Security*, 2006.
[22] A. Sinkov and T. Feil. *Elementary cryptanalysis*, volume 22. MAA, 2009.
[23] D. Stehlé and R. Steinfeld. Faster fully homomorphic encryption. In *ASIACRYPT 2010*, pages 377–394. Springer, 2010.
[24] M. Twain. The adventures of tom sawyer. https://www.gutenberg.org/ebooks/74.txt.utf-8.
[25] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in cryptology–EUROCRYPT 2010*, pages 24–43. Springer, 2010.
[26] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of pgp 5.0. In *Usenix Security*, volume 1999, 1999.
[27] P. R. Zimmermann and P. R. Zimmermann. *The official PGP user's guide*, volume 265. MIT press Cambridge, 1995.