For Some Eyes Only: Protecting Online Information Sharing

Iulia Ion[§] Filipe Beato[†] Srdjan Čapkun[§] Bart Preneel[†] Marc Langheinrich[‡]

[§]ETH Zurich
Zurich, Switzerland
{iion,capkuns}@inf.eth.ch

[†]ESAT/ COSIC – KULeuven and IBBT
Leuven, Belgium
{first.lastname}@esat.kuleuven.be

[‡]University of Lugano (USI)
Lugano, Switzerland
langheinrich@acm.com

ABSTRACT

End-users have become accustomed to the ease with which online systems allow them to exchange messages, pictures, and other files with colleagues, friends, and family. This convenience, however, sometimes comes at the expense of having their data be viewed by a number of unauthorized parties, such as hackers, advertisement companies, other users, or governmental agencies. A number of systems have been proposed to protect data shared online; yet these solutions typically just shift trust to another third party server, are platform specific (e.g., work for Facebook only), or fail to hide that confidential communication is taking place. In this paper, we present a novel system that enables users to exchange data over any web-based sharing platform, while both keeping the communicated data confidential and hiding from a casual observer that an exchange of confidential data is taking place. We provide a proof-of-concept implementation of our system in the form of a publicly available Firefox plugin, and demonstrate the viability of our approach through a performance evaluation.

1. INTRODUCTION

Online sharing platforms enable a new communication paradigm. Users disclose intimate thoughts on Facebook, blog about their political views, upload holiday pictures to Google Plus, and publish their current activities on Twitter. According to estimations by the social media blog 'The Social Skinny,' over one billion Facebook posts, 175 million Tweets, and 10 years worth of YouTube videos are being uploaded by users every day [47]. This rise in online sharing activity has prompted increasing privacy and security concerns among consumers [28]. By publishing their private information on a range of public or semipublic platforms, consumers get exposed to unauthorized disclosure of their data. Unauthorized access can happen, for instance, if hackers break into user accounts (e.g. the 2009 Google cyber attack [33]), bugs in the access control enforcement system allow unauthorized users to view user data [53], or platform providers grant advertisement companies access without the user's consent for economical reasons [48].

Several solutions have been proposed to protect user data from unauthorized usage. One solution requires the user to encrypt the data before uploading it to the online sharing platform, and to distribute encryption keys to authorized recipients only. For instance, PGP encryption allows users to protect email communication and attachments. However, this approach breaks the ease of sharing data on dedicated platforms such as online social networking sites, which manage users' network of friends and display the data in the browser.

Other solutions displace the trust users put in platform providers by creating sharing systems owned and hosted by users. Such systems require users to run their own web servers or sharing platforms for hosting their data. For example, Diaspora [27] is a private social network that runs on servers owned and operated users. Such approaches, however, force the user to trade the usability of popular data sharing platforms for better privacy protection. This tradeoff might come at the expense of losing the interaction with potentially less privacy-concerned friends. Even apparently successful systems like Diaspora have a much lower user base than popular sharing systems used today. For instance, Diaspora's user base is only 1.9 million [25]; Twitter has 140 million users [52] and Facebook has over 900 million users [29]. To view protected pictures, user's friends would instead of accessing Facebook need to access the user's personal web server, or start using Diaspora or similar services.

Instead, we desire a technical solution that allows users and their friends to have a similar experience on the platforms they normally use for sharing (e.g., Facebook, Flickr), yet be protected from unauthorized usage of their data. Some systems have been proposed that attempt to protect user data from unauthorized usage, while still allowing consumers to use their platform of choice. However, these solutions either require the existence of a trusted third party server to handle user data and encryption keys, are platform specific (e.g., work for Facebook only), or do not hide the fact that confidential data is being exchanged [10, 41, 42, 50].

In this paper, we propose a system that protects user data on web-based social sharing platforms; our system does not require the user to run a dedicated infrastructure or place his trust in another third party, and hides from unauthorized recipients that confidential communications is taking place. In doing so, we are inspired by what Boyd calls *social steganography* [16]: Boyd found that teenagers sometimes post messages on Facebook that seem innocent to parents (e.g, song lyrics), but carry hidden meaning for friends. Similarly, our system allows users to post innocent looking pictures, files, or status updates that will transparently be replaced with real information for selected recipients in the user's network. Note that while *steganography* typically refers to concealing a message or file within another message or file, our system hides a pointer to the protected data, not the data itself.

As a proof of concept, we implemented our system in the form of a plugin for the Firefox browser. Despite the vastly different nature of websites, the underlying HTML elements used to construct user interfaces for uploading text input, pictures, and other documents are the same on all platforms. Our plugin is thus able to support most web-based data-sharing platforms with the help of platform-specific XML-based definition files that allow it to seamlessly replace dummy postings with hidden values.

Contributions. (1) We propose a system for protecting data on online sharing platforms through strong user-side encryption; (2) we introduce a novel mechanism inspired by social steganography techniques to hide the fact that encrypted communication is being transmitted; and (3) we demonstrate the feasibility of our approach through a proof of concept implementation in the form of a publicly available browser plugin.

Outline. First, Section 2 gives an overview of the main idea, presents our threat model and describes the goals of our system. Section 3 defines the components and protocols of our system, and Section 4 presents the security analysis of our protocol. Then, Section 5 presents our implementation approach and performance evaluation. We discuss solutions that make our approach resistant to data-mining techniques for detecting protected messages in Section 6, review related work in Section 7, and conclude in Section 8.

2. A SHORT OVERVIEW

Consider a user who wishes to upload a protected wall post, status update, or a picture to an online social networking (OSN). While the user wants to take advantage of the communication channel offered by the OSN, he also wants to ensure that only a specific set of authorized recipients can access it, keeping the OSN provider and unauthorized parties oblivious. To this end, the user could just post the encrypted version of the content. However, some OSNs impose length limitations or are not able to display encrypted pictures. Thus, our system stores the encrypted data on a different storage service. It then stores a different cleartext, consisting of fake data on the sharing platform. To enable authorized friends to retrieve the encrypted data, our system stores a pointer to its location on a disjoint Internet mapping service.

More specifically, our system performs the following operations. At first, it replaces the user's real posting (i.e., text or a picture) on the OSN with fake data that looks like another genuine message—either automatically or with the user's help. The user's real data is encrypted for a user-defined set of recipients and stored in a user-selectable, public storage service (e.g., Dropbox, or the user's own server), which returns a URL to the encrypted content. Next, in order to keep the storage location private the system applies a pseudo-random function to the posted fake data (our implementation uses a keyed hash function) and computes a lookup-key. In a final step, this lookup-key is then used to store the (encrypted) URL of the encrypted file in another user-selectable, arbitrary URL lookup service (e.g., TinyURL). On the authorized recipient's side, the system performs the reverse operations while the user visits the OSN page. See also Figure 1 for an overview.

Threat model. We consider an attacker who has control over the communication channels used by the user to store and share data. However, we assume that the attacker does not know the secret keys of the users, and cannot control the user computing environments, such as their browsers and computers, and any device used in the protocol.

Goals. Our system should support sharing most data types directly in the browser on a wide range of web-based online sharing platforms. All content published by a user should be kept confidential by means of cryptographic techniques. However, for a good usability, the operations should be simple and the cryptographic techniques transparent. Only authorized recipients should be able to read and verify the integrity of the protected data. Obviously, once a recipient gets access to the protected content, there is no way of preventing this recipient from redistributing it. However, a casual observer should not be able to infer that hidden information is being transmitted.

3. THE SYSTEM

To unlink innocent-looking data stored on the communication platform from the encrypted data secretly exchanged, our system makes use of the following main services:

Online Sharing Platform ($\mathcal{SP}$) is any online communication platform for storing and sharing digital content (e.g., Facebook, Flickr, Gmail). Such platforms usually require registered login, keep and manage the user's list of contacts, and are regularly accessed by the user's friends.

Storage Service (SS) allows storing user data in the cloud and accessing this data through a browser (e.g., Dropbox, SugarSync). We assume that SS requires users to register before storing data. We assume that each file f stored in SS is accessible through a unique URL, which we denote url_f, and that anybody who knows url_f can retrieve the file without authenticating. Nevertheless, only the account owner can modify and delete stored data.

Hashmap Directory (HD) is a web-based service that stores short strings mapping (*index, value*) pairs, such as URL shortener services TinyURL or Bit.ly. Given an index, it allows anybody to retrieve the value. The service does not accept duplicate indices and places a restriction on the length of both the index and value strings (e.g., 30–140 characters). We consider that stored entries do not expire and cannot be deleted. We also assume that HD accepts any anonymous requests to store and retrieve entries, and does not limit the number of entries one user can make.

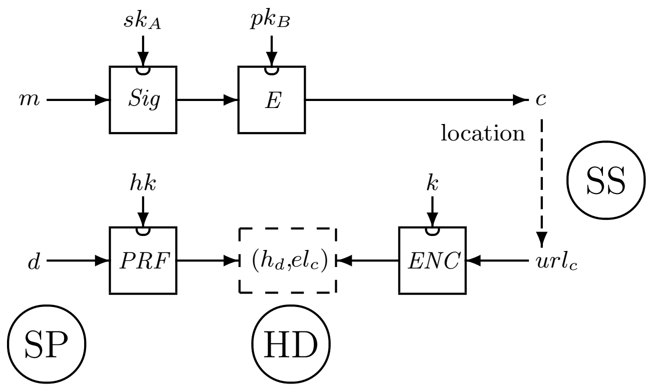


Figure 1: Transmitting a protected message m: After signing and encrypting m, the result c is stored in a publicly accessible URL url_c on storage service SS. Using the fake message d as input to a pseudo-random function, a mapping service HD is then used to store the encrypted link el_c under index h_d.

3.1 Transmitting a Protected Message

In our system, every user U owns a public/private key pair (pk_U, sk_U). Let the users, Alice and Bob who want to exchange protected messages on the platform $\mathcal{SP}$. We assume that Alice and Bob have exchanged and verified their public keys pk_A and pk_B. These keys will be used to encrypt m. We also assume that Alice and Bob have run a key agreement protocol and agreed on two shared keys k_{AB}, a symmetric encryption key, and hk_{AB} a key used to compute a pseudo-random function PRF. We will discuss key management in detail in Section 3.2. In the following, we abbreviate encryption of m with the public key pk_B as $E_{pk_B}(m)$ and decryption using the private key sk_B as $D_{sk_A}(m)$. Furthermore, we abbreviate symmetric encryption using the shared secret key k_{AB} as $ENC_{k_{AB}}(m)$ and decryption as $DEC_{k_{AB}}(m)$.

3.1.1 Send Protected Text

Figure 1 gives an overview of the protocol to exchange a protected message and Figure 2 shows in detail the messages exchanged by the parties. First, to ensure message integrity, Alice signs m with her private key sk_A and obtains $\sigma_m = Sign_{sk_A}(m)$. Then, Alice encrypts (m, σ_m) with Bob's public key pk_B to obtain $c = E_{pk_B}(m, \sigma_m)$.

Next, Alice uploads the ciphertext c to SS. Subsequently, Alice notes the URL url_c under which c can be retrieved. Note that SS allows anyone to access c through the URL url_c without requiring authentication. Next, Alice chooses a dummy text d that looks like a genuine message she wants to transmit to Bob. Alice applies a *pseudo-random function* (PRF) [32] to d using the shared key hk_{AB} and computes $h_d = PRF_{hk_{AB}}(d)$. Note that the result h_d must be an uniformly distributed string, thus a PRF is the appropriate tool to achieve this in the standard model; using random oracles would impose stronger requirements for the building blocks without any benefits. Then she encrypts url_c using a symmetric algorithm to obtain $el_c = ENC_{k_{AB}}(url_c)$. We use symmetric encryption to compute el_c, which produces a ciphertext smaller than public key encryption, because we assume that HD poses a length limitation on the registered content. Finally, Alice registers el_c under the index h_d with HD and publishes the dummy text d on the online sharing platform $\mathcal{SP}$. Since HD only accepts unique index h_d entries, d must not have been used by Alice to communicate with Bob before, under the same key hk_{AB}.

3.1.2 Read Protected Text

To retrieve the protected message m, Bob follows the reverse steps. He first reads the dummy text d published by Alice on the platform $\mathcal{SP}$. Bob tries to see if there is a hidden message m behind d. To verify this, Bob first computes $h_d = PRF_{hk_{AB}}(d)$, and queries HD to see if there is any value registered under h_d. If no value is registered, Bob concludes that d is a genuine, plaintext message from Alice with no protected content behind it. Otherwise Bob receives the encrypted link el_c from HD and decrypts it to obtain $url_c = DEC_{k_{AB}}(el_c)$. Knowing url_c, Bob retrieves c from SS. Next, Bob decrypts c using his private key sk_B to obtain the initial message $(m, \sigma_m) = D_{sk_B}(c)$. Knowing σ_m, Bob verifies the integrity of the message using pk_A. If the integrity verification fails, Bob concludes that Alice is not the actual sender or that an attacker tampered with c. Otherwise, he considers m a message from Alice.

3.1.3 Send Protected Image or File

Assume that instead of the text m, Alice wants to share a secret file. To transmit, for example, a secret image i, Alice follows the same protocol as for text but with a slight variation. First, Alice encrypts i with pk_B and stores the resulting $c = E_{pk_B}(m, \sigma_i)$ in SS. Next, Alice chooses a dummy image d. Because online sharing platforms often perform image processing techniques such as image compression on uploaded pictures, Alice and Bob cannot use a pseudorandom function on the dummy image; doing so might result in different h_d values on the sender and receiver side. For this reason, Alice chooses a random value w_d and hides it in the image d as a secret watermark using a secret derived from hk_{AB} [37]. She then computes $h_d = PRF_{hk_{AB}}(w_d)$. Finally, as for text, Alice registers the encrypted link el_c under the index h_d with HD, and publishes d on $\mathcal{SP}$.

To receive the protected image i, Bob extracts the secret watermark w_d from the dummy image d, computes $h_d = PRF_{hk_{AB}}(w_d)$, and queries HD to retrieve the encrypted link el_c. He then decrypts el_c to obtain $url_c = D_{k_{AB}}(el_c)$ and retrieves c stored on SS at location url_c. Finally, Bob decrypts c using sk_B, obtains the initial image i and verifies the signature σ_i.

3.1.4 Group Communications

Assume that Alice wants to share m with a group of contacts G, not just with Bob. All recipients in G know that Alice is the sender of the protected message based on information provided by the sharing platform, but they should not find out the identity of other recipients in G. In a straightforward solution, Alice could perform the steps described earlier for each recipient U in G, using shared keys k_{AU} and hk_{AU}. However, this solution results in poor performance for large groups of recepients, because Alice must encrypt the message m for each recipient, and then compute and register different (h_d, el_c) values with HD. To obtain better performance, Alice can encrypt the ciphertext only once, using an anonymous broadcast encryption scheme [9, 40].

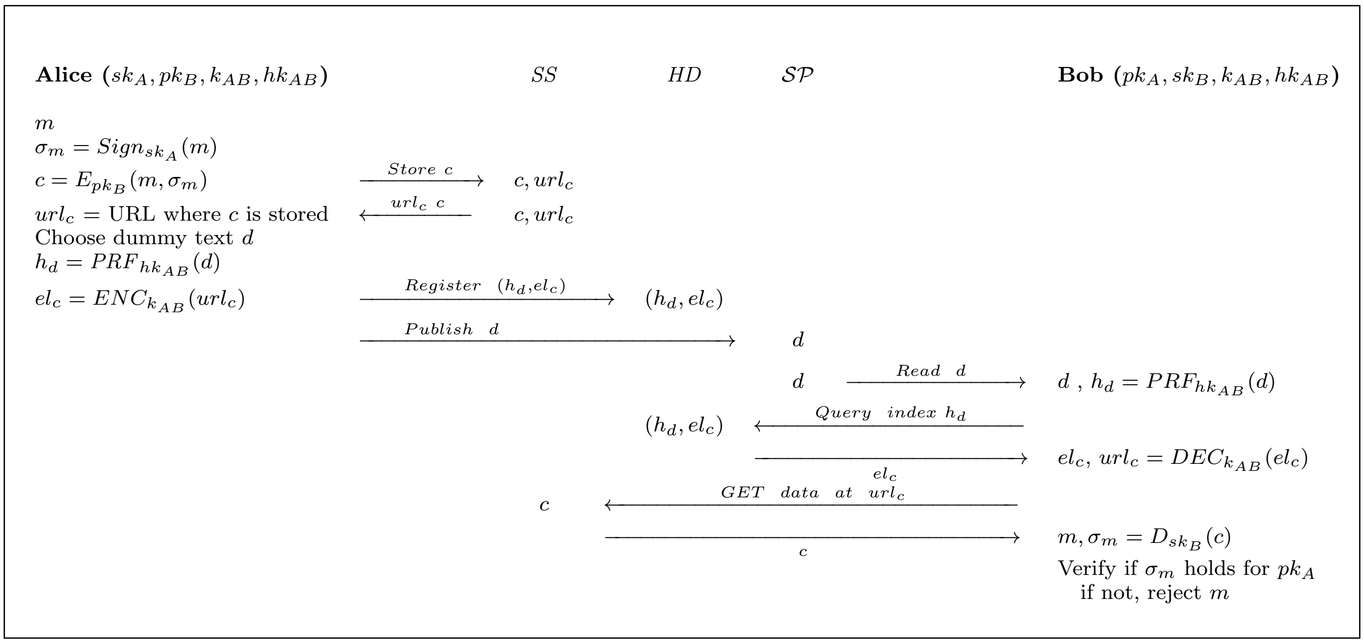

Alice $(sk_A, pk_B, k_{AB}, hk_{AB})$
SS
HD
$\mathcal{SP}$
Bob $(pk_A, sk_B, k_{AB}, hk_{AB})$
m
$\sigma_m = Sign_{sk_A}(m)$
$c = E_{pk_B}(m, \sigma_m)$
$Store\ c$
c, url_c
url_c = URL where c is stored
$url_c\ c$
c, url_c
Choose dummy text d
$h_d = PRF_{hk_{AB}}(d)$
$el_c = ENC_{k_{AB}}(url_c)$
$Register\ (h_d, el_c)$
(h_d, el_c)
$Publish\ d$
d
d
$Read\ d$
$d\ ,\ h_d = PRF_{hk_{AB}}(d)$
(h_d, el_c)
$Query\ index\ h_d$
el_c
$el_c,\ url_c = DEC_{k_{AB}}(el_c)$
c
$GET\ data\ at\ url_c$
c
$m, \sigma_m = D_{sk_B}(c)$
Verify if σ_m holds for pk_A
if not, reject m

Figure 2: Sending a protected message; (pk_B, sk_B) is Bob's public/private key pair; k_{AB} is a symmetric encryption key and hk_{AB} a key for the pseudo-random function PRF, both shared by Alice and Bob.

Depending on the desired trade-off between protocol security guarantees, on the one hand, and system performance and scalability, on the other, the shared keys (k_G, hk_G) used by Alice to transmit protected messages to the group G could be (a) the same for all the contacts in a specific group, or (b) different for each contact. Assume Alice shares different keys (k_{AU}, hk_{AU}) with each contact U. To publish protected content, she must compute different h_{dU} and el_{c_U} values for each recipient U and register each (hd_U, el_{c_U}) pair with HD. Note that all el_{c_U} actually decrypt to the same url_c. Maintaining different keys per contact provides stronger security guarantees (e.g., in case some keys are compromised or leaked), but requires creating more HD entries, which ultimately affects performance and increases network traffic.

3.1.5 Access Right Revocation

Assume that Alice shared m with a group G of users that includes Bob. She now wants to remove Bob's access, but have the data available m to other recipients. Note that it might be impossible for Alice to remove or modify a message m already published on $\mathcal{SP}$ (e.g., emails that have already reached the recipients' inbox, some forum entries, etc). We distinguish between two cases, depending on whether Alice used different shared keys k_{AU} and hk_{AU} with each recipient U in the group G or group keys k_G and hk_G to transmit m. We assume that Alice stored the ciphertext c at the same location url_c for all recipients. A basic solution to revoke Bob's access to m has Alice simply re-encrypt m to the altered recipient list that excludes Bob, and update the ciphertext c stored on SS. Bob is still in possession of the shared keys k_{AB} and hk_{AB} or the group keys k_G. If he still has access to d (or knows d from a previous access), Bob can find out url_c and therefore retrieve c, by simply following the steps for retrieving a protected message under d. Therefore, Bob could still obtain c, and thus prove the existence of a protected message. However, since c is no longer encrypted with his public key pk_B, Bob cannot decrypt c and find out the content of m.

If Alice is not able to delete d or remove Bob's access from $\mathcal{SP}$ and the $(index, value)$ entries in HD are permanent, she must follow the following steps to impede Bob from proving the existence of m. First, Alice must delete c from SS, thus invalidating url_c and store the ciphertext at a new location url'_c. Furthermore, Alice must establish new keys (k'_{AU}, hk'_{AU}) or (k'_G, hk'_G) with the other recipients in G. She should then create new $(h'_d = PRF_{hk'_G}(d),\ el'_c = ENC_{k_G}(url'_c))$ entries on HD. All entries should point to the same location url'_c. If, however, Alice encrypted and stored the ciphertext c in different locations url_{c_U} for each recipient U, she needs only to remove c_B from url_{c_B} since this operation will not affect other recipients in G.

3.2 Key Management

In this section we consider approaches for contacts to perform key establishement, as well as key revocation. Furthermore, we discuss migrating private keys and contact lists across different personal devices.

3.2.1 Key Establishement

Prior to being able to use the system and exchange protected messages, Alice and Bob must first exchange and verify their public keys pk_A and pk_B. Furthermore, Alice and Bob must agree on the shared keys (k_{AB}, hk_{AB}). Then, Alice adds Bob to her contact list and stores the keys pk_B, k_{AB}, and hk_{AB} locally on her machine. To establish shared secret keys, Alice and Bob must follow the following steps:

1. Exchange public keys. Alice and Bob can exchange pk_A and pk_B over a private channel such as email or by publishing them on a public or semi-public platform such as social networking sites. Publishing the keys over a public platform accessed by her contacts and automatically retrieving them from there offers better scalability than performing one-on-one exchange with each of her contacts. By using multiple $\mathcal{SP}$ platforms, one can separate the exchange of cryptographic material from the account used to transmit protected messages, thus hiding clues that encrypted information might be exchanged in the future. Note that this public channel is considered untrusted and might be subject to man-in-the-middle attacks.

2. Verify public keys. Because a malicious attacker might have mounted a man-in-the-middle attack during step 1, Alice and Bob must make use of a trusted out-of-band channel (e.g., QR codes, GSM network, Bluetooth communications) to verify the fingerprints of pk_A and pk_B. We consider this second channel harder (e.g., the SMS network) or impossible (e.g., direct capturing of QR codes) to compromise. To hide the exchange of public keys from a suspicious observer, Alice and Bob could even resort to the out-of-band channel to perform step 1. If we operate under an honest-but-curious attacker, we might choose to skip or postpone this step to a later stage, for instance, until after the start of the protected communication. This is desirable if, Alice and Bob want to start communicating immediately.

3. Generate shared keys. Having exchanged and verified their public keys, Alice and Bob can run a key establishment protocol over the untrusted Internet channel to obtain the shared keys k_{AB} and hk_{AB}. For example a shared secret key could be obtained using a Sigma Protocol of [39], that is an extended version of the authenticated Diffie-Hellman key agreement protocol. As long as the private keys sk_A and sk_B are not compromised, revocation of k_{AB} and hk_{AB} can take place by merely publishing signed (and possibly encrypted) messages over any agreed-upon Internet communication platform. For example, Alice and Bob could perform the Sigma protocol by transmitting messages on Facebook signed with their private keys sk_A and respectively sk_B to ensure that no attacker interfered during the key agreement protocol.

3.2.2 Key Revocation

If the key k_{AB} gets compromised, an attacker could compute h_d and find out el_c. If, additionally, the attacker also knows hk_{AB}, he can then decrypt el_c to obtain url_c. Because from url_c he can obtain c, the attacker is able to prove the transmission of a protected message. However, as long as he does not know the private key sk_B, he cannot decrypt c to find out the the content of m.

If the shared keys k_{AB} and hk_{AB} get compromised, Alice must re-run the key agreement protocol in step 3 with Bob to obtain new (k'_{AB}, hk'_{AB}) values. If Alice's private key sk_A gets compromised, she must inform her contacts, re-run the key exchange and agreement protocol, and re-encrypt previous content with the new key sk'_A.

3.2.3 Key Migration

Assume Alice has generated her key pair (pk_A, sk_A) on her personal laptop, but now wants to be able to view protected messages on her work desktop as well. To be able to decrypt protected messages on another device, Alice must have (pk_A, sk_A) and the shared secret keys (k_{AU}, hk_{AU}) available on the new device. Therefore, she must securely migrate the secret keys to her work desktop. Note that it suffices for Alice to transfer her key pair (pk_A, sk_A) to the new device through an out-of-band channel, and then synchronize her encrypted contact list and shared keys between several devices by posting encrypted and signed messages in SS or another online storage platform. For weaker protection, but arguably more usability, instead of her public/private key pair, Alice could carry only a strong passphrase to the new device through a confidential out-of-band channel. The passphrase could then be used in a key derivation function (KDF) to generate a symmetric key. This key is then used to encrypt Alice's sk_A and the keys (k_{AU}, hk_{AU}) (e.g., using a symmetric authenticated encryption, such as AES in CCM-mode [56]), and then decrypt them on the new device.

4. SECURITY ANALYSIS

We analyze the resilience of our system against a number of attacks. We show that none of the services used by our system can find out the content of m or provide its existence, whether they work independently or collaborate. Furthermore, we show that an attacker cannot carry out impersonation attacks and discuss possible approaches to defend our system from traffic analysis attacks.

Sharing Platform. Because Alice published d on $\mathcal{SP}$, $\mathcal{SP}$ knows d. However, $\mathcal{SP}$ does not know the key hk_{AB}. Therefore, it cannot compute $h_d = PRF_{hk_{AB}}(d)$. Consequently, $\mathcal{SP}$ cannot query HD to find out the value el_c registered under the index h_d. Hence, $\mathcal{SP}$ cannot find out m or find out whether there is a hidden message m behind d. However, $\mathcal{SP}$ can erase or alter d, which would result in a denial of service for the communication between Alice and Bob. Under suspicion, $\mathcal{SP}$ might be able to replace d with a previous message transmitted by Alice d', which could hide a protected message m'. In this case, Bob would verify m' as a message originating from Alice, but $\mathcal{SP}$ could not know or choose the value of m'.

Hashmap Directory. HD knows h_d and el_c, but does not know k_{AB}, therefore it cannot decrypt el_c to obtain $url_c = DEC_{k_{AB}}(el_c)$, the location of the ciphertext c. Also, although HD knows h_d, it cannot extract the value of the fake message d (this follows immediately from the properties of a pseudo-random function, $h_d = PRF_{hk_{AB}}(d)$). Furthermore, HD cannot tell if h_d corresponds to any given d because it does not know the key hk_{AB} in the case of text or the key variable for extracting the secret watermark in the case of images. As a result, HD cannot identify m and c corresponding to (h_d, el_c).

Storage Service. SS has access to the encrypted data c, and possibly all other ciphertext stored by Alice. However, since SS does not know the private key sk_U of any authorized recipient, it cannot decrypt c to find out $m = D_{sk_U}(c)$. Furthermore, even though SS knows url_c, it cannot compute $el_c = DEC_{k_{AB}}(url_c)$ because it does not know k_{AB}. Given an entry el_c on HD, SS cannot verify if el_c decrypts url_c. Therefore, SS cannot find out if c is linked to a message d stored by Alice on another platform $\mathcal{SP}$. However, SS can tamper with c, remove it or replace it with a different ciphertext c', previously generated and posted by Alice.

Collusion. We consider that in special circumstances $\mathcal{SP}$, HD, and SS might collude and share user information among themselves or with other parties for profit or legal obligations. We show that, although any attacker with access to d stored on $\mathcal{SP}$ can tell that Alice and Bob are communicating, he cannot tell that there is a hidden message behind d. An attacker cannot link d published on $\mathcal{SP}$ to the ciphertext c stored in SS because he cannot compute h_d.

However, $\mathcal{SP}$, HD, and SS might keep logs, record users' IP addresses and requests, which could be used by an attacker to match requests made by the same user. By matching IP addresses or the timing of the requests, an attacker could conclude that a message d on $\mathcal{SP}$, a ciphertext c on SS and an entry (h_d, el_c) originate from the same user. Thus the attacker might be able to infer the existence of a protected message transmitted with d, though he cannot find out the content of m. We acknowledge that a highly motivated attacker (e.g., governments) might be able to link information across all protocol parties ($\mathcal{SP}/SS/HD$). In practice, however, such attacks may be non-trivial for commercial and/or political reasons, as these parties may be competitors or located in different countries. To avoid such attacks, one could hide IP addresses by running our system on top of Tor [3] and defend against timing attacks by introducing random noise and delay in user requests.

Impersonation attacks. An attacker with read and write access to all messages posted and received by Alice on $\mathcal{SP}$ (e.g., $\mathcal{SP}$ themselves, a hacker who got a hold of Alice's account, or a governmental agency who requests access from $\mathcal{SP}$) could try to create fake messages d and convince Bob that a hidden message m actually comes from Alice. Since the attacker does not know the secret k_{AB} (which is only know by Alice and Bob), he cannot create a valid d, h_d pair. Given a d published by the attacker on Alice's behalf, Bob will query HD for h_d and conclude that there is no hidden message when nothing is returned. If, however, HD is malicious and colludes with the attacker, it could fake an entry by returning a chosen value el_c for any h_d submitted by Bob. However, since HD does not know k_{AB}, it cannot compute a valid el_c that decrypts to an url_c, but could mount a replay attack if in possession of a previous value el'_c posted by Alice for Bob. If SS also colludes and returns a given c' for any request el'_c originating from Bob, the attacker has the chance to deliver a chosen ciphertext c' to Bob. Knowing Bob's public key pk_B, the attacker could compute $E_{pk_B}(m')$. However, the attacker cannot trick Bob to believe this message comes from Alice because he cannot generate a valid signature $\sigma = Sign_{sk_A}(m')$.

5. IMPLEMENTATION

We implement our system as a Firefox plugin, basing our implementation on the Scramble! open source project [11]. Part of our plugin is implemented in Java, therefore the user must have Java Applet support enabled to run our plugin. We use AES-CCM for symmetric (authenticated) encryption, HMAC-SHA-256 as pseudorandom function, and the OpenPGP standard [17] for broadcast encryption. We make use of Dropbox as a Storage Service (SS) and TinyURL as the Hashmap Directory (HD). Ultimately, the user could select from a list of available storage platforms. We use TinyURL because it allows the user to choose a custom short URL to map to. TinyURL could be interchanged with similar URL shortening services, publishing services or online blogs that can store a public list of index-value pairs.

During the first installation for user U, the plugin creates a new Dropbox account with a random username; subsequently it generates an OpenPGP public/private key pair (pk_U, sk_U) and the shared keys k_G and hk_G for his group of friends G. All encrypted user data is later stored in the `Public` folder of the Dropbox account and is accessible through a public URL.

5.1 Sharing Protected Text

The user invokes the plugin while the mouse cursor is inside the input area, e.g., by a mouse right click menu. The plugin extracts the message m typed in by the user in the input field and manipulates the HTML page to replace m with a chosen fake text d. In our implementation, the user must enter d. We discuss approaches on how to automatically generate good fake messages in Section 6.

Support for any Text Input Fields: Websites are becoming richer and more complex, making use of complex Javascript calls and HTML code. As a result, text entry is no longer restricted to just a few HTML elements such as `<input type='text'>` and `<textarea>`. For example, in Gmail, the input area for composing email messages is in fact an editable `<html>` element within an `<iframe>`. Our plugin can handle special text input types. It identifies the HMTL node containing the entered user input through the `document.popupNode` Firefox API call. It then obtains the inserted text from its `.value` attribute or `.innerHTML`, depending on the HTML node type.

Support for Rich Text Formatting: Web-based sharing platforms increasingly encourage users to edit and annotate documents, and write HTML rich emails and blog entries. As a consequence, separating user-generated content from the page source is becoming more challenging. Our plugin aims to protect user data without loss of website functionality. For example, the email reply together with the initial secret email are tightly coupled with Gmail's specific HTML webpage email header. When clicking the "Send" button, it is crucial to avoid reposting the initial secret message m (which is being displayed on the page, but is unknown to Gmail who only knows d). To achieve this, in our implementation the whole message thread, including the Gmail reply headers and the tags for rich HTML formatting are encrypted, and replaced with a new dummy text.

5.2 Sharing Protected Images

Unlike text input, which can be implemented through a variety of means, file upload in the browser takes place exclusively through an `<input type='file'>` HTML element. When a webpage is loaded, the plugin identifies all `file input` elements and registers `change` event listeners for all of them. Consequently, when the user selects a file to upload, the plugin gets notified first and prompts the user whether to protect the file. Note that this file protection mechanism can be applied to any file type. In our implementation, the plugin retrieves images from different Flickr pages, given a start URL and XPath-based webpage parsing and navigation rules. In a real setting, one might want to be careful about copyright issues.

For watermarking images, we use the DCT-watermark library [31]. Unlike steganography, good image watermarks are resistant to typical image compression, some cropping and scaling techniques. To ensure that only intended recipients can retrieve the watermark from the image, we use a secret derived from the encryption key k to embed and extract the watermark. We noticed that success rate is dependent on the used images. Based on our experiments, the DCT-watermark library needs on average two tries to successfully embed a 20-digit long watermark on a random Flickr picture (success rate 54%, N=595 pictures, t_{avg}=0.3s). We are not currently aware of other libraries who might perform better. However, a better chosen pool of pictures might lead to better success rates. Finally, the plugin automatically

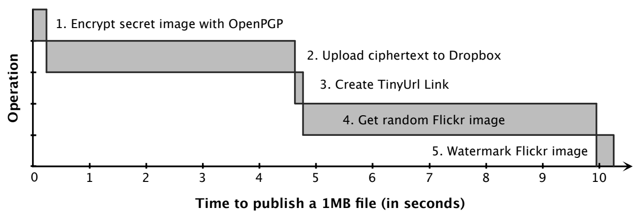


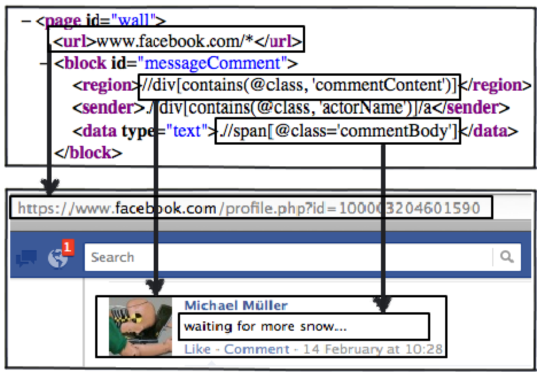
Figure 3: Steps needed to publish a protected file. For optimization, steps could be run in parallel or be precomputed.

updates the file selection in the `<input>` field to the watermarked image. For security reasons, websites and Javascript code, including Firefox plugins, are by default not allowed to change the value of a file HTML `<input>` element. To this end, we sign our plugin with a trusted certificate, and request higher security privileges needed.

Unlike text, we display secret images through a pop-up window. For security reasons, images stored locally cannot be embedded in a webpage hosted remotely by simply manipulating the value of the `src` attribute on an HTML `<img>` tag (see the strict origin security policy [2]). This ultimately helps raise more user awareness on data protection levels. Alternativelly, decrypted images could be hosted and retrieved from a local web server and displayed in line.

5.3 Extensible Page Parsing Rules

For our implementation to be online sharing platform independent, we make use of simple XML rules that define where and on which pages the browser should expect hidden data. Adding support for one more communication platform comes down to adding the XML specification files. To specify the page structure on a generic form, we make use of XPath queries [58]. XPath is a language used to navigate through elements and attributes in an XML document which uses path expressions to select nodes or node-sets. We use XPath queries to identify (sender, message) pairs on a page. Figure 4 shows an example. The `region` query is used to restrict the search on the page to a single section containing published messages. The execution of the `sender` and `message` subqueries is then restricted to the identified region. Next, the identified sender is matched against contacts from the address book, which can contain email addresses, nicknames and user IDs. To show the universal applicability of our solution, we defined parsing rules for any type of communication over Gmail, Facebook and Twitter. Such XPath-based rules need to be updated if web interfaces change. For the full specifications, see the Appendix.



```
- <page id="wall">
    <url>www.facebook.com/*</url>
  - <block id="messageComment">
      <region>//div[contains(@class, 'commentContent')]</region>
      <sender>.//div[contains(@class, 'actorName')]/a</sender>
      <data type="text">.//span[@class='commentBody']</data>
    </block>
```

https://www.facebook.com/profile.php?id=100003204601590

Michael Müller
waiting for more snow...
Like · Comment · 14 February at 10:28

Figure 4: The plugin identifies the dummy messages candidates based on webpage-specific XPath parsing rules.

5.4 Key Management and Distribution

There are two main approaches for end-users to distribute public keys: (1) rely on a mutually trusted certification authority, or (2) manually verify the authenticity by checking the fingerprints of the public keys through an out-of-band channel [36]. However, most everyday users do not have mutually trusted certification authorities (CA) [36]. Furthermore, obtaining certificates from certification authorities is too difficult, expensive and time consuming. It takes even for power users 30 minutes to 4 hours to obtain a certificate from a public CA that performs little to no verification [34]. Therefore, our system perform key exchange and verification through trusted, out-of-band channels.

The plugin makes cryptographic operations, including key generation, management and distribution, transparent to users, thus avoiding the pitfalls of previous systems [57]. The plugin distributes public keys by publishing them on users' Facebook profiles embedded in a QR code image. Our plugin can be easily extended to distribute keys and run key agreement and cryptographic communication protocols over any other platform, by adding external JAR files that are automatically loaded at runtime.

To support key management and verification, we implemented an Android mobile application with two out-of-band key verification methods: SMS and phone-to-phone QR code scanning. The mobile application holds the user's public/private key pair (which is transferred from the computer through a QR code). Furthermore, the plugin can encrypt and sign verified contact keys and upload them to the Dropbox account from where they are synchronized with the browser plugin on other user devices. Finally, the power of the presented solution would not be complete if limited to PCs only. All the functionality of the plugin can be easily ported to a mobile platform, for example in the form of stand-alone mobile applications for different platforms. (Unfortunately, plugin development for Firefox on the Android platform currently lacks Java Applet support.)

5.5 Performance Evaluation

A smooth user experience is essential in the success of any security solution; otherwise users will sacrifice security for usability. To calculate its performance, we run the plugin on a MacBook Pro laptop with an Intel Core i5 2.4GHz processor and 4GB of memory over a wireless network. The plugin has a memory consumption of 70MB. We measured the time needed to retrieve and display hidden messages on a Facebook page from the time the page is loaded in the browser. Note that only messages with senders in the contact list are candidates for protected communication. Processing a Facebook page with one hidden message (out of two candidates) took on average 0.9s, (N=10, $stdev$=0.2s). Displaying a page with 10 hidden messages (out of 11 candidates) took 6s (N=10, $stdev$=0.6s). On average, retrieving a hidden text message took 0.5s (N=25, $stdev$=0.07s), and processing a message that does not hide any communication took 0.06s (N=25, $stdev$=0.004s). Posting a hidden message took on average 0.67s (N=10, $stdev$=0.1s). Therefore, two users talking over a protected chat message system would experience a delay of approx. 1 second. For our plugin, the time to display a page increases linearly with the number of hidden messages.

Figure 3 displays the time needed to execute each step of our implementation, in order to securely send a 1MB file to 100 contacts who share group shared keys k_G and hk_G. We present here only the extra security steps that must be preformed by our plugin, in comparison to the normal browser experience. The computation intensive tasks, file encryption (1) and image watermarking (5), take very little time compared to network operations, uploading the encrypted file to Dropbox (2) and retrieving a random image from Flickr (4). Note that given the OpenPGP symmetric type of encryption, the encrypted file has approximatively the same size as the initial file. Uploading an encrypted 1MB file to Dropbox took on average 4.4s ($stdev$=0.6s, N=20). Figure 5 shows that the time needed to encrypt a file increases linearly with the number of contacts and file size, but remains relatively low, below 2 seconds for a 100MB file and 500 contacts. Finding and saving a Flickr image took on average 5.2s, of which 3.8s were needed to download and parse the starting webpage. Once the URL was identified, saving an image locally took on average only 0.9s (N=50). Creating a TinyURL mapping the secret watermark to the encrypted Dropbox link took only 0.1s (N=20, $stdev$=0.01s).

While executing all steps sequentially could account for slow browser response time and ultimately poor usability, implementation optimizations can make the process seem instantaneous. For example, a pool of Flickr pictures could be retrieved and stored locally beforehand. Since for files and images the TinyURL index h_d is not a secret derived from a dummy text chosen by the user, but rather from a randomly generated string, even image watermarking could be pre-executed. Similarly, uploading the encrypted file could happen in parallel to other operations and finish after the upload of the watermarked image.

6. SEMANTICS AND MINING ATTACKS

Suspicion of using our system could cause trouble to users in countries with totalitarian regimes, or simply refusal of service from platform providers with business models exclu-

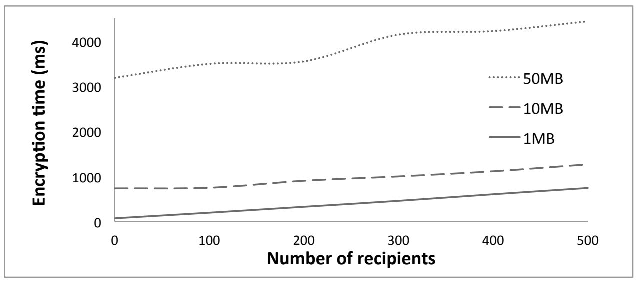


Figure 5: Encryption time increases slowly, remaining bellow one second for a 1MB file shared with 500 contacts.

sively based on targeted advertising. It should therefore be impossible for single or colluding services to tell which users communicate using covert messages. As discussed in Section 3, our solution provides complete data confidentiality against any attacker having access to one or all of $\mathcal{SP}$, HD, SS. However, as data mining and profiling techniques are becoming more advanced and possibly even used by oppressive governments to identify activists, it is crucial to ensure that users cannot be singled out based on semantic analysis of dummy messages and steganographic data. In this section we make a few initial considerations on how dummy messages could be automatically generated.

It should be hard or impossible for an outside observer (e.g., platform provider, governmental agency) to tell with high probability, through mass-targeted or user-targeted mining, that users are protecting their communication. Automatically generated messages must, therefore, be consistent with past user behavior. Ultimately, to ensure less detectability, users can compose the dummy messages themselves. For good usability, however, the plugin should make a good suggestion. Furthermore, it is likely that users are not good at coming up with diverse dummy messages either.

Resistance to machine detection. Previous work on detecting automated posts on Twitter and social networks mainly focused on spam and used simple detection techniques that would not work against our solution. For example, Benvenuto et al. [12] use behavior attributes, such as average hash tags per tweet, number of tweets received, account age, number of followers per number of followees, and fraction of tweets with URLs. Zhang et al. [59] analyze timestamps and observe that humans post messages at random times of the day, whereas bots post at specific minutes of the hour. Since in our system the user would always be the one initiating the posting, such techniques would not be effective to identify dummy messages. Automatic publishing of noise messages, however, should take into account such considerations.

Consistent user behavior. Constantinides et al. [19] have shown that user behavior on social networks follows well defined trends. The authors found that user profiles cluster into four main categories, depending on their usage patterns on Facebook: *Beginner*, *Habitual*, *Outstanding*, and *Expert*. The authors then quantify the likelihood for a certain type of user to engage in a certain activity. For instance, searching for people online, sending private messages and updating profiles are popular among all users, while reporting about products used and commenting about advertising are mostly done by *Expert* users. A plugin user might be easily identified if, for instance, all the public posts on his Facebook wall are about sharing current activities, while all the steganographic messages are TinyURL links (e.g., Scramble! [11]) or sentences from Wikipedia (e.g., FaceCloak [42]).

A solution based on topic models. To make sure dummy text messages are consistent with the previous topics in users' communications, one could use text document analysis techniques, such as constructing and applying topic models [15]. In particular, we propose using the Latent Dirichlet Allocation (LDA), a generative model in which any specific document is viewed as a mixture of topics. Each topic is characterized by a distribution of words. LDAs can be used to learn and define topics from an existing set of documents or communication. Other models consider correlation [14] and hierarchy between topics [18]. Previous work already applied topic models to social networks [45] and images [55].

Possible implementation. To generate valid dummy text messages, one might want to restrict the social network activities identified by Constantinides et al. [19] to those that can be used to transmit a social steganographic message by our plugin: discuss what people do, communicate news or issues, share mood, share links about interesting web sites, report about current activities, and report about brands or products. Based on such predefined types of communication and their expected frequencies, a heuristic could be defined that, for each posting attempt, takes as input a pseudorandom number and determines what kind of dummy text message to generate. In addition, to disguise the usage of steganographic messages the plugin could insert noise communication.

The Machine Learning for Language Toolkit [44] a Java-based software for document classification and topic modeling could be easily integrated with our plugin and used to analyze past user communication. Dummy messages consistent with identified topics could be generated through different means. For instance, the set of keywords in a predicted topic could be used to retrieve sentences from the web through a Google search. Steganographic replies could be generated with the use of online Turing test chats (e.g., Touring Hub [51]), or aggregated among several users and outsourced as tasks to human workers on Mechanical Turk [1].

Good usability. As described above, generating semantically correct sentences, which must be consistent topic-wise with user profiles, is not a trivial task for a computer. The problem becomes even more challenging on long steganographic communication threads, which might be visible to other users, not just to the intended recipients (e.g., public posts on Facebook). Furthermore, the existence of steganographic messages poses usability challenges. More research is needed to find out how well users cope with friends replying to fake posts. Finally, adequate interfaces should help users keep track of the two worlds: the steganographic messages and the hidden message thread.

7. RELATED WORK

Some solutions have been proposed to increase privacy on different online sharing platforms like exclusively on social networks or webmail platforms. However, these solutions either: (1) do not hide that communication is confidential [11, 8], (2) protect only certain kind of data (e.g., profile information or private messages) [42], (3) require the existence of dedicated infrastructure or a trusted third-party, or (4) are restricted to a specific platform. Other solutions propose novel, privacy-friendly architectures meant to replace existing platforms [4, 21, 22, 24, 35]. Instead, our solution enables consumers to keep using current system while protecting their privacy.

While a large body of work has explored privacy-preserving data sharing in an outsourcing scenario (e.g. [6, 38]), these settings differ significantly from ours. They assume the existence of a set of private databases that should be opened up for queries from others without giving complete access to the raw data—a common scenario in business relationships. In contrast, we present a scheme for directly sharing information across a covert channel, without other parties being aware of this fact.

Many research solutions and commercial products have been proposed to encrypt messages and files exchanged over webmail platforms [11, 23, 26, 30, 41, 43]. However, these solutions do not hide that communications are encrypted. Scramble! [11] uses cryptographic mechanisms to enforce access control rules and ensure confidentiality of sensitive information. Our plugin inherits the concept of using OpenPGP for group communication from Scramble!. It can replace encrypted text with a TinyURL link that points to a server storing the ciphertext, but the link and content are public. Furthermore, Scramble! does not offer any support for files, which is a central contribution of our system.

In the context of social networks, Conti et al. [20] propose Virtual Private Social Networks to protect some static user profile information on social networks: name, picture, and current city. The authors do not consider messages and files. The user posts fake information online and distributes his real data unencrypted in an XML file to his friends over email. This data is then matched using regular expressions and automatically replaced by the browser, similar to how our plugin offers extensibility through XPath queries. Yeung et al. [7] also take a decentralized approach. Each user has a trusted server which stores his data, has knowledge of social network specific functionality and applications (e.g., photo tagging, personal wall), and enforces access control rules based on cross-platform specifications. When trying to access data on different platforms, the user's friends are redirected to the trusted server which must handle the access control rules.

FaceCloak [42] is similar to our approach, but limited to Facebook profile data and messages. The secret information is encrypted and stored on a dedicated server, while fake information (e.g., random sentences from Wikipedia) is posted on Facebook. Just like FaceCloak, our system substitutes real information with dummy one. In FaceCloak, the fake information and a key that only the sender and all his contacts know serve to compute the index under which the third-party server stores the ciphertext. In contrast, our solution works solely with currently available services on the Internet and does not require dedicated infrastructure. In addition, our scheme supports per-group communications and file exchange on any platform. Furthermore, by separating the storage service from the Hashmap Directory, we protect against fake data creation in case steganography keys are leaked. This can happen if a malicious user leaks the shared group steganography keys, but the encryption keys of other users (i.e., their private keys) remain uncompromised. If one has control over the FaceCloak server and access to the user's index key, one could swap ciphertexts and create successful plaintext swapping.

StegoWeb [13] is implemented as a browser bookmarklet, i.e., as a simple program that can be executed by clicking on a bookmark in the browser. The user must rely on a trusted third-party server to perform the encryption and data steganalysis. StegoWeb does not use public key cryptography. Instead, for each piece of data shared with each recipient, both the sender and the receiver must enter a shared passphrase. By supporting public key encryption, our solution offers more security and better scalability. Oren and Wool [46] propose a system which increases webmail privacy by hiding the email content with text steganography and then splitting the output into two parts. The user must send the two parts over two different email accounts. This solution requires no key distribution, but protects only against a weak attacker who has access to one of the two webmail servers.

Secretwit [49] uses text and image steganography to transmit secret Twitter messages (e.g., by appending whitespaces at the end of the tweet). The size of messages that can be transmitted purely through steganography is limited, because the size of the hidden data must be much smaller than that of the carrier message. Therefore, an approach purely based on steganography causes a serious limitation on the type and the volume of data users can transmit. For example, picture sharing platforms often compress and resize images, which could not hide a high-quality picture. By transmitting only a pointer to the location of the secret data instead of the data itself, our system poses no limitation on the size and type of protected information. For text, we provide the user with complete freedom on how to compose the dummy messages, thus making it less likely to being identified as unusual communication. Furthermore, our approach is robust against basic image manipulation techniques applied by online sharing platforms.

Adkinson-Orellana et al. [5] encrypt documents stored in Google Docs and enable simultaneous editing of encrypted documents among a group of people by intercepting the HTTP requests for AJAX calls and encrypting/decrypting transmitted document content. This approach requires knowledge of a platform-specific AJAX protocol and does not hide the nature of encrypted communication. However, by dealing with document editing, this work is orthogonal to ours and could be extended to provide a viable solution for group editing of protected data in our system.

8. CONCLUSIONS

While users are lured into storing their personal data online and sharing it with others over an ever wider range of free services (e.g., webmail, social networks, photo sharing platforms), they have little control over which third parties can access their data (e.g., hackers, advertisement companies, governmental agencies). In this paper, we propose a system that allows users to protect data they share online, and hide the fact that confidential information is being exchanged from unauthorized recipients. Our system does not rely on dedicated infrastructure or trusted servers. While the secret data is encrypted and stored in the cloud, dummy data that looks like genuine files or user communication is uploaded on the sharing platform. Users share secret keys which they use to discover the hidden data behind the dummy one. They find out the encrypted location of the secret data through a public indexing service like TinyURL. Our system accounts for easy specification of location of expected hidden messages on HTML paged through simple XML rules based on XPath parsing queries. We provide a proof of concept implementation of our system in the form of a Firefox plugin that focusses on protecting text messages and images.

Our solution does not hide the exchange of communication between two parties, thus leaving such transactional data open to law enforcement agencies. However, an attacker who does not have access to the users' secret keys cannot detect the exchange of confidential communication. Our system preserves most website functionality including text and image display. However, because data is encrypted and not actually stored on the online sharing platform, our solution does cause a loss of functionality on certain types of systems, e.g., Google Spreadsheets. For most online sharing platform functionality though, techniques such as encrypted search promise to be a viable solution [54]. Further research should investigate techniques to impede websites from sniffing the data while being entered by the user in the browser. One possible solution is to have the plugin disable Javascript which the user types in the message. Furthermore, a future study should evaluate the detectability of image watermarks in different watermarking algorithms. If an attacker can identify users who post watermarked pictures, he might be able to narrow down the consumers who use our system.

While our proof-of-concept implementation deals only with plaintext and image exchange, our solution can be similarly used to implement protected exchange of any data type, including video files and data documents. Further work could implement sharing protected video files through platforms like YouTube. Based on research by Boyd [16], we believe users can cope well with the usage of steganographic messages. However, further research is needed to test the usability of our plugin and devise adequate user interfaces to help them distinguish between regular messages and protected communication. Most importantly, further research should look into having users specify recipient rules for each website or page. The browser should learn users' preferences in terms of when, which data should be protected for which recipients, and automatically apply those data protection policies. Dummy pictures used for watermarking could come from a local folder with user's personal pictures, be automatic repostings of Facebook pictures in which the user was tagged from his friends' profiles, or general photos from public websites (e.g., search queries on Flickr, Google Image Search), possibly corrupted to be harder to match against originals. Finally, future work should investigate techniques to generate sound dummy messages and data.

9. REFERENCES

[1] MTurk. https://www.mturk.com/.

[2] Strict origin policy. http://kb.mozillazine.org/Security.fileuri.strict_origin_policy. Accessed on August 31, 2012.

[3] Tor. http://www.torproject.org.

[4] 2peer. http://2peer.com. Accessed on Sept. 3, 2012.

[5] L. Adkinson-Orellana, D. A. Rodriguez-Silva, F. J. Gonzalez-Castano, and D. Gonzalez-Martinez. Sharing secure documents in the cloud—a secure layer for Google Docs. In *Proc. of CLOSER 2011*.

[6] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *Proc. of ACM SIGMOD*, pages 86–97, June 2003.

[7] C. M. Au Yeung, I. Liccardi, K. Lu, O. Seneviratne, and T. Berners-Lee. Decentralization: The future of online social networking. In *Proc. W3C Workshop on the Future of Social Networking*, January 2009.

[8] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: an online social network with user-defined privacy. *SIGCOMM Computing Communication Review*, 39(4):135–146, 2009.

[9] A. Barth, D. Boneh, and B. Waters. Privacy in encrypted content distribution using private broadcast encryption. In *Proc. of Financial Cryptography and Data Security*, pages 52–64, Feb. 2006.

[10] F. Beato, M. Kohlweiss, and K. Wouters. Enforcing access control in social networks. In *Proc. of HotPets 2009*, pages 10–21, August 2009.

[11] F. Beato, M. Kohlweiss, and K. Wouters. Scramble! your social network data. In *Proc. of Privacy Enhancing Technologies*, Waterloo, Canada, July 2011.

[12] F. Benevenuto, G. Magno, T. Rodrigues, and V. Almeida. Detecting spammers on Twitter. In *Proc. of the 7th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference*, Redmond, 2010.

[13] T. Besenyei, A. M. Foldes, G. G. Gulyas, and S. Imre. StegoWeb: Towards the ideal private web content publishing tool. In *Proc. of SECURWARE 2011*, pages 109–114, August 2011.

[14] D. Blei and J. Lafferty. A correlated topic model of science. *Annals of Applied Statistics*, 1:17–35, 2007.

[15] D. M. Blei. Probabilistic topic models. *Communications of the ACM*, 55:77–84, 2012.

[16] D. Boyd and A. Marwick. Social steganography: Privacy in networked publics. In *International Communication Association*, Boston, MA, May 2011.

[17] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), November 2007.

[18] J. Chang and D. Blei. Hierarchical relational models for document networks. *Annals of Applied Statistics*, 4(1):124–150, 2010.

[19] E. Constantinides, M. del Carmen Alarcón del Amo, and C. L. Romero. Profiles of social networking sites users in the netherlands. In *Proc. of HTSF 2010*.

[20] M. Conti, A. Hasani, and B. Crispo. Virtual private social networks. In *Proc. of ACM CODASPY 2011*.

[21] E. D. Cristofaro, C. Soriente, G. Tsudik, and A. Williams. Hummingbird: Privacy at the time of twitter. In *IEEE Security and Privacy*, 2012.

[22] L. A. Cutillo, R. Molva, and T. Strufe. Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine*, 47(12):94–101, 2009.

[23] G. D'Angelo, F. Vitali, and S. Zacchiroli. Content cloaking: Preserving privacy with Google Docs and other Web applications. In *Proc. of SAC 2010*, pages 826–830.

[24] Diaspora. https://joindiaspora.com/. Accessed on Sept. 3, 2012.

[25] How many users are in the Diaspora network? Data as of Sept. 6, 2012. https://diasp.eu/stats.html. Accessed on Sept. 6, 2012.

[26] DocCloak. http://www.gwebs.com/doccloak.html. Accessed on Sept. 3, 2012.

[27] J. Dwyer. Four nerds and a cry to arms against Facebook. May 11, 2010. http://www.nytimes.com/2010/05/12/nyregion/12about.html. Accessed on Sept. 3, 2012.

[28] Facebook and your privacy: Who sees the data you share on the biggest social network? Consumer Reports magazine, June 2012. http://www.consumerreports.org/cro/magazine/2012/06/facebook-your-privacy/index.htm. Accessed on Sept. 6, 2012.

[29] Facebook Newsroom—Key Facts. http://newsroom.fb.com/content/default.aspx?NewsAreaId=22. Accessed on Sept. 3, 2012.

[30] FireGPG. http://getfiregpg.org. Accessed on Sept. 3, 2012.

[31] C. Gaffga. DCT-watermark: Robust watermarks for color JPEG in java. https://code.google.com/p/dct-watermark/. Accessed on Sept 3., 2012.

[32] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, August 1986.

[33] A new approach to China. Google Official Blog, http://googleblog.blogspot.com/2010/01/new-approach-to-china.html, January 13, 2010.

[34] P. Gutmann. Plug-and-play PKI: a PKI your mother can use. In *Proc. of USENIX Security 2003*, pages 4–4.

[35] S. Jahid, S. Nilizadeh, P. Mittal, N. Borisov, and A. Kapadia. DECENT: a decentralized architecture for enforcing privacy in online social networks. In *Proc. of SESOC 2012*, 2012.

[36] A. Kapadia. A case (study) for usability in secure email communication. *Security Privacy, IEEE*, 5(2):80–84, 2007.

[37] S. Katzenbeisser and F. A. Petitcolas, editors. *Information Hiding Techniques for Steganography and Digital Watermarking*. Artech House, Inc., Norwood, MA, USA, 2000.

[38] L. Kissner and D. Song. Privacy-preserving set operations. In *Proc. of CRYPTO 2005*, pages 241–257.

[39] H. Krawczyk. SIGMA: The 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE-protocols. In *Proc. of CRYPTO 2003*.

[40] B. Libert, K. G. Paterson, and E. A. Quaglia.

Anonymous Broadcast Encryption: Adaptive Security and Efficient Constructions in the Standard Model. In *Proc. of PKC 2012*.

[41] M. M. Lucas and N. Borisov. FlyByNight: mitigating the privacy risks of social networking. In *Proc. of ACM WPES*, October 2008.

[42] W. Luo, Q. Xie, and U. Hengartner. FaceCloak: An architecture for user privacy on social networking sites. In *Proc. of ICCSE 2009*, pages 26–33, August 2009.

[43] MailCloak. `http://www.gwebs.com/mailcloak.html`. Accessed on Sept. 3, 2012.

[44] MALLET. `http://mallet.cs.umass.edu/`. Accessed on August 31, 2012.

[45] A. McCallum, X. Wang, and A. Corrada-Emmanuel. Topic and role discovery in social networks with experiments on enron and academic email. *Journal of Artificial Intelligence Research*, 30:249–272, 2007.

[46] Y. Oren and A. Wool. Perfect privacy for webmail with secret sharing. `http://www.eng.tau.ac.il/~yos/spemail/OrenWool-SPEmail.pdf`. Accessed on Sept. 6, 2012, Feb. 2009.

[47] C. Pring. 100 more social media statistics for 2012. Web Blog "the social skinny". Feb. 13, 2012. `http://thesocialskinny.com/100-more-social-media-statistics-for-2012/`. Accessed on Sept. 6, 2012.

[48] C. Riederer, V. Erramilli, A. Chaintreau, B. Krishnamurthy, and P. Rodriguez. For sale : your data: by : you. In *Proc. of ACM HotNets-X 2011*, pages 13:1–13:6.

[49] SecreTwit. `http://code.google.com/p/secretwit/`. Accessed on Sept. 6, 2012.

[50] A. Tootoonchian, K. K. Gollu, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: social access control for Web 2.0. In *Proc. of WOSN 2008*, August 2008.

[51] Touring Hub. `http://testing.turinghub.com/`. Accessed on August 31, 2012.

[52] Twitter turns six. Twitter Blog. March 21, 2012. `http://blog.twitter.com/2012/03/twitter-turns-six.html`. Accessed on Sept. 6, 2012.

[53] J. E. Vascellaro. Google discloses privacy glitch. WJS Blogs, March 8, 2009. `http://blogs.wsj.com/digits/2009/03/08/1214/`. Accessed on Sept. 6, 2012.

[54] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou. Secure ranked keyword search over encrypted cloud data. In *Proc. of ICDCS 2010*, June 2010.

[55] Y. Wang and G. Mori. A discriminative latent model of image region and object tag correspondence. In *Proc. of NIPS 2010*.

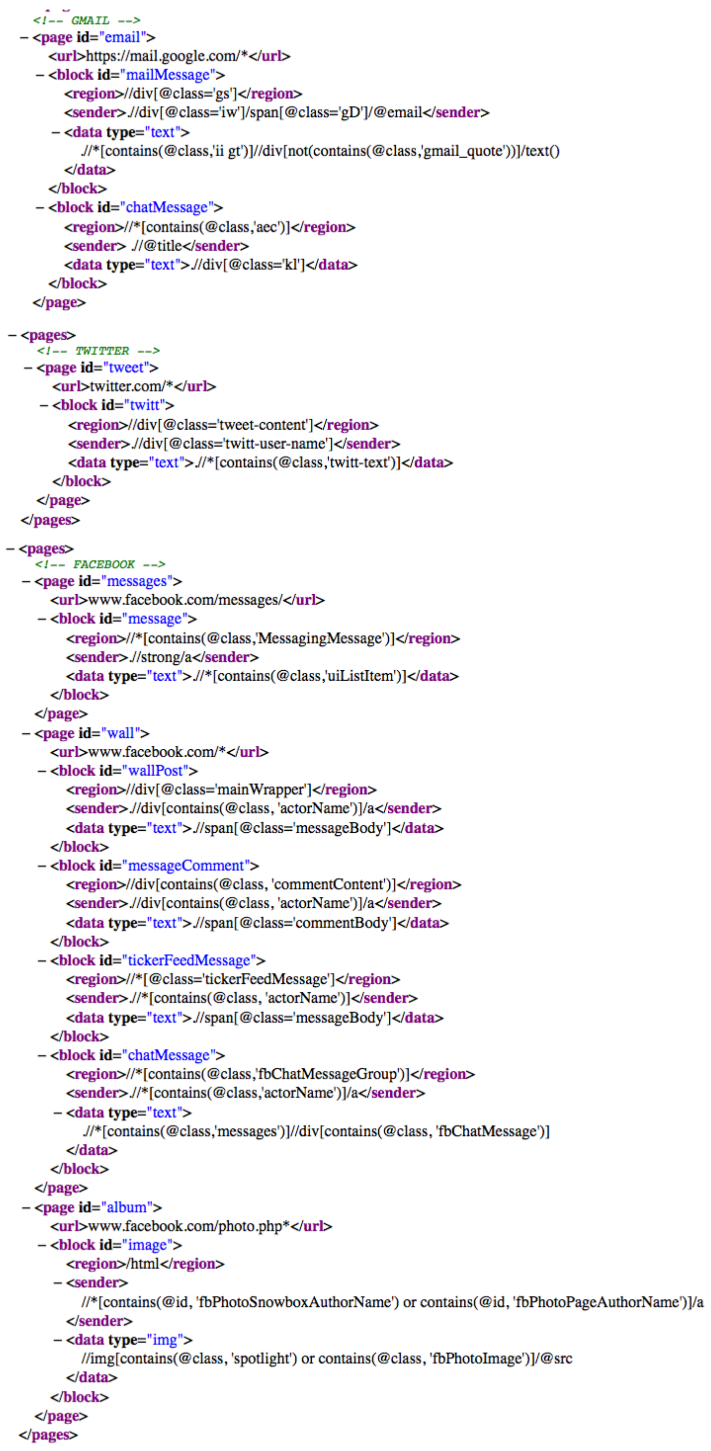
[56] D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). RFC 3610, Sept. 2003.

[57] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. In *Proc. of USENIX Security 1999*, pages 169–184.

[58] XPath. `http://www.w3schools.com/xpath/`. Accessed on Sept. 6, 2012.

[59] C. M. Zhang and V. Paxson. Detecting and analyzing automated activity on Twitter. In *Proc. of PAM 2011*.

APPENDIX



```
    <!-- GMAIL -->
  - <page id="email">
      <url>https://mail.google.com/*</url>
    - <block id="mailMessage">
        <region>//div[@class='gs']</region>
        <sender>.//div[@class='iw']/span[@class='gD']/@email</sender>
      - <data type="text">
          .//*[contains(@class,'ii gt')]//div[not(contains(@class,'gmail_quote'))]/text()
        </data>
      </block>
    - <block id="chatMessage">
        <region>//*[contains(@class,'aec')]</region>
        <sender> .//@title</sender>
        <data type="text">.//div[@class='kl']</data>
      </block>
    </page>

- <pages>
    <!-- TWITTER -->
  - <page id="tweet">
      <url>twitter.com/*</url>
    - <block id="twitt">
        <region>//div[@class='tweet-content']</region>
        <sender>.//div[@class='twitt-user-name']</sender>
        <data type="text">.//*[contains(@class,'twitt-text')]</data>
      </block>
    </page>
  </pages>

- <pages>
    <!-- FACEBOOK -->
  - <page id="messages">
      <url>www.facebook.com/messages/</url>
    - <block id="message">
        <region>//*[contains(@class,'MessagingMessage')]</region>
        <sender>.//strong/a</sender>
        <data type="text">.//*[contains(@class,'uiListItem')]</data>
      </block>
    </page>
  - <page id="wall">
      <url>www.facebook.com/*</url>
    - <block id="wallPost">
        <region>//div[@class='mainWrapper']</region>
        <sender>.//div[contains(@class, 'actorName')]/a</sender>
        <data type="text">.//span[@class='messageBody']</data>
      </block>
    - <block id="messageComment">
        <region>//div[contains(@class, 'commentContent')]</region>
        <sender>.//div[contains(@class, 'actorName')]/a</sender>
        <data type="text">.//span[@class='commentBody']</data>
      </block>
    - <block id="tickerFeedMessage">
        <region>//*[@class='tickerFeedMessage']</region>
        <sender>.//*[contains(@class, 'actorName')]</sender>
        <data type="text">.//span[@class='messageBody']</data>
      </block>
    - <block id="chatMessage">
        <region>//*[contains(@class,'fbChatMessageGroup')]</region>
        <sender>.//*[contains(@class,'actorName')]/a</sender>
      - <data type="text">
          .//*[contains(@class,'messages')]//div[contains(@class, 'fbChatMessage')]
        </data>
      </block>
    </page>
  - <page id="album">
      <url>www.facebook.com/photo.php*</url>
    - <block id="image">
        <region>/html</region>
      - <sender>
          //*[contains(@id, 'fbPhotoSnowboxAuthorName') or contains(@id, 'fbPhotoPageAuthorName')]/a
        </sender>
      - <data type="img">
          //img[contains(@class, 'spotlight') or contains(@class, 'fbPhotoImage')]/@src
        </data>
      </block>
    </page>
  </pages>
```

Figure 6: XPath-based webpage parsing rules for Gmail, Twitter, and Facebook