HITC: Data Privacy in Online Social Networks with Fine-Grained Access Control

Ahmed Khalil Abdulla
ahabdulla@mail.hbku.edu.qa
Division of Information and Computing Technology
College of Science and Engineering
Hamad Bin Khalifa University
Doha, Qatar

Spiridon Bakiras
sbakiras@hbku.edu.qa
Division of Information and Computing Technology
College of Science and Engineering
Hamad Bin Khalifa University
Doha, Qatar

ABSTRACT

Online Social Networks (OSNs), such as Facebook and Twitter, are popular platforms that enable users to interact and socialize through their networked devices. The social nature of such applications encourages users to share a great amount of personal data with other users and the OSN service providers, including pictures, personal views, location check-ins, etc. Nevertheless, recent data leaks on major online platforms demonstrate the ineffectiveness of the access control mechanisms that are implemented by the service providers, and has led to an increased demand for provably secure privacy controls. To this end, we introduce *Hide In The Crowd* (HITC), a flexible system that leverages encryption-based access control, where users can assign arbitrary decryption privileges to every data object that is posted on the OSN platforms. The decryption privileges can be assigned on the finest granularity level, for example, to a hand-picked group of users. HITC is designed as a browser extension and can be integrated to any existing OSN platform without the need for a third-party server. We describe our prototype implementation of HITC over Twitter and evaluate its performance and scalability.

CCS CONCEPTS

• **Security and privacy → Access control**; **Social network security and privacy**; *Privacy protections*;

KEYWORDS

Data privacy; access control; hidden vector encryption; online social networks

ACM Reference Format:
Ahmed Khalil Abdulla and Spiridon Bakiras. 2019. HITC: Data Privacy in Online Social Networks with Fine-Grained Access Control. In *The 24th ACM Symposium on Access Control Models and Technologies (SACMAT '19), June 3–6, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3322431.3325104

SACMAT '19, June 3–6, 2019, Toronto, ON, Canada

ACM ISBN 978-1-4503-6753-0/19/06…$15.00
https://doi.org/10.1145/3322431.3325104

1 INTRODUCTION

Online Social Networks have fundamentally changed the way people interact with their social contacts. Face-to-face meetings and private phone calls have been replaced by posts, tweets, and instant messages. OSNs come in many different flavors: some focus on personal relationship interactions (Google+), some are tailored to career and professional interactions (LinkedIn), others focus on short posts and comments (Twitter), where others specialize in photo-sharing services (Instagram and Flickr). Their impact on our daily lives has been significant; in Jan 2018, Twitter reported 100 million daily active users [8] while, in Mar 2017, Facebook announced an average of 1.28 billion daily active users [4], by far the most popular platform that comes with a wide range of social applications, including personal interactions, photo-sharing, and instant messaging.

When users subscribe to an OSN, they tend to share a lot of private information [15], such as their personal identity, their friends' and family members' identities, their political and religious views, and even personal photographs. Unfortunately, data privacy is often neglected by the OSN service providers. Some OSNs do offer configurable privacy controls that limit access to shared data, but users might misconfigure these controls due to their complexity or lack of clear instructions [6, 20]. Furthermore, the controls are typically coarse-grained, because they lack flexibility in defining and grouping authorized users and protected information. More importantly, as demonstrated in Facebook's latest data breach [30], existing privacy controls do not restrict third-party applications and developers from accessing users' private information. Finally, OSN service providers have full access over the data stored on their servers, which raises numerous privacy and security concerns. For example, OSNs might share such data with third parties (e.g., advertisers), malicious employees could access them without authorization, and hackers could target them for personal gains.

As a result, the research community has proposed several techniques that employ cryptographic primitives to provide provably secure privacy controls. Nevertheless, some solutions necessitate the implementation of brand new OSN platforms [9, 13, 14], others rely on third-party servers [17, 18, 31], and some are implemented as native applications on the OSN platform [18, 22]. To the best of our knowledge, the current literature lacks a comprehensive approach that (i) implements fine-grained access control over encrypted data, (ii) works seamlessly over existing OSN platforms, (iii) does not require third-party servers, and (iv) hides its activities from the OSN users and service providers (low profile).

In this paper, we introduce *Hide In The Crowd* (HITC), a flexible and user-friendly system that leverages encryption-based access control to assign arbitrary decryption privileges to every data object that is posted on the OSN platforms. HITC employs hidden vector encryption (HVE) [24], which is a ciphertext policy-based access control mechanism. Under HVE, each user generates his/her own master key (one-time) that is subsequently used to generate a unique decryption key for every user with whom they share a link in the underlying social graph. Moreover, during the encryption process (i.e., when posting a new object), the user interactively selects a list of friends that will be granted decryption privileges for that particular data object. To facilitate the deployment of our system over existing OSN platforms, we designed HITC as a Chrome browser extension and utilize steganographic techniques [23] to hide the encrypted data objects within randomly chosen cover images. As such, HITC can operate undetected by other OSN users and even the OSN service provider itself.

Another important feature of our system is that it employs *in-band* mechanisms to perform its basic operations, such as key distribution. Specifically, the users' social relationships information and their HVE-based decryption keys (assigned to them by other users), are all hidden within cover images posted on the OSN platform. The only offline communication required is a one-time exchange of a secret passphrase, during the relationship establishment process between two users. Furthermore, HITC retrieves all the necessary information on-demand, by utilizing the APIs of the underlying OSN platform. Therefore, HITC does not rely on third-party servers, which removes a potential attack vector that can compromise the operation of the system.

The contributions of our work can be summarized as follows:

- We propose HITC, the first system for data privacy in online social networks that operates entirely at the end-devices without any infrastructure support.
- We design our system as a Chrome browser extension that provides a user-friendly environment for enforcing fine-grained access control on shared data. The design is independent of the underlying social network platform.
- We deploy a prototype implementation of HITC on the Twitter platform and evaluate its performance and scalability.

2 PRELIMINARIES

This section briefly describes the cryptographic primitives that are employed in our system, namely hidden vector encryption and image steganography.

2.1 Hidden Vector Encryption

In this work, we utilize the ciphertext policy HVE (CP-HVE) scheme by Phuong et al. [24]. Under this scheme, every ciphertext and decryption key is associated with an access vector of length L that contains letters from an alphabet Σ. The decryption operation is successful if and only if the two vectors match, i.e., the letters in each of the L positions are identical. However, the access vector of the ciphertext (encryption) is allowed to contain wildcard symbols ('$\star$') in at most N positions. A wildcard symbol will match any letter of the alphabet in the access vector of the decryption key. The protocol consists of the following four algorithms:

- **Setup**$(1^k, \Sigma, L, N)$: on input a security parameter 1^k, an alphabet Σ, a vector length L, and a maximum number of allowed wildcards N in the encryption vector, the algorithm outputs a public key $hve.PK$ and a master secret key $hve.MK$.
- **GenDecKey**$(hve.MK, hve.PK, \vec{z})$: on input a master secret key $hve.MK$, a public key $hve.PK$, and a decryption vector $\vec{z}$, the algorithm outputs a decryption key $hve.DK$.
- **Enc**$(hve.PK, \vec{x}, \vec{j}, M)$: on input a public key $hve.PK$, an encryption vector $\vec{x}$, a vector $\vec{j}$ containing the locations of the wildcards in $\vec{x}$, and a message M, the algorithm outputs a ciphertext C.
- **Dec**$(hve.DK, \vec{j}, C)$: on input a decryption key $hve.DK$, a vector $\vec{j}$ containing the locations of the wildcards in the encryption vector, and a ciphertext C, the algorithm outputs a message M.

In HITC, we use the alphabet $\Sigma = \{0, 1\}$ and every user i is assigned a decryption vector that consists of zeroes, except for position i which holds the value of one. Similarly, when a user wants to encrypt a message for users i and j, the encryption vector contains wildcards at positions i and j, while all the remaining values are zero. Appendix A describes the construction of the HVE scheme [24] that leverages pairing-based cryptography.

2.2 Image Steganography

Steganography is the art of concealing secret information within non-secret data. Image steganography is one branch of steganography, where secret data is being hidden within the image's pixels. The image used to carry the secret data is commonly called a *cover image* and the resulting image, after the secret data is embedded, is called a *stego image*. One pixel of a grayscale image can have a value from 0 to 255 (8 bits), whereas a color image's pixel has 3 channels (RGB), each with an 8-bit value (i.e., a total of 24 bits). Color images can also have a fourth channel per pixel, called *alpha*, with an additional 8-bit value. This alpha channel controls the transparency level of the pixel in an image. The embedding of secret data within an image can be achieved with different techniques [23]. We highlight below the most common image steganography techniques:

- **Structure-based technique:** This technique exploits specific, usually optional, markers in the JPEG format to embed secret data. For example secret data can be embedded into the *Exchangeable Image File* (EXIF) [7], the Comment markers [1, 2], or after the end of the image (EOI) marker.
- **Spatial domain technique:** Due to the fact that the human eye's perception is not sensitive to slight changes in an image's pixels, this technique exploits the Least Significant Bit (LSB) of the pixel values of the cover image to embed the secret data.
- **Frequency domain technique:** To avoid visual distortion, the frequency domain technique replaces the LSBs in the quantized DCT coefficients whose values are not zero [23].
- **Distortion-resistant schemes:** These schemes are most robust to image processing which lowers the *bit error rate* (BER) by using redundancy and/or marker techniques. For example, YASS [27] uses a redundancy parameter to control

the number of times an information bit is repeated inside an image.

Moreover, image formats differ from one to another. Some formats have a *lossy compression* feature (e.g., JPEG images), where an image permanently loses some of its data when compressed that can never be retrieved. The primary aim of this feature is to decrease the image quality in order to reduce its size. Other image formats have a *lossless compression* feature, where an image retains all its data when compressed that allows it to be fully recovered without any loss after decompression (e.g., PNG images).

3 HITC DESIGN

This section presents in detail the design of HITC. It involves the basic operations of the system and is independent of the underlying OSN platform (e.g., Facebook, Twitter). In the following section we will introduce a proof-of-concept implementation for Twitter.

3.1 Design Goals

We designed HITC with the following goals in mind:

- **Encryption-based privacy:** Protecting the privacy of data shared over OSNs is the ultimate and primary goal of our system. We aim to offer the end-user the ability to protect his/her own shared data, not only from other OSN users but also from the OSN service provider itself.
- **Fine-grained access control:** We aim to provide the end-user with the ultimate flexibility for sharing private data. Unlike existing approaches that define groups where users subscribe, HITC supports dynamic grouping for every object that is posted. The list of friends that can access a single object is decided on-the-fly when the object is posted on the OSN.
- **Utilize existing OSNs:** Given the popularity of existing OSNs, their large user base, and the already established relationships between its users, it is impractical to build a new OSN with stringent privacy controls, as it would be very difficult to convince existing OSN users to move to a new platform. Therefore, we aim to build a system that can operate on top of existing OSNs.
- **Eliminate third parties:** We want to eliminate the need for a third-party server, either trusted or untrusted. As explained previously, third parties introduce another attack vector for adversaries and can be used to compromise the availability of the system.
- **Low profile:** OSN service providers may not allow the sharing of secret information over their platforms, because unencrypted user data is the source of their revenue. As a result, they may be prone to restrict access to users whom they suspect of employing covert communications. Therefore, we want to design a system that is transparent to both unauthorized users and the OSN service providers.
- **Stateless mobility:** Nowadays, users are not logging into OSN services from a single device. For example, a user might login using his personal laptop, the company's workstation, or his mobile device. Thus, HITC aims to support user mobility, where a user can login from any device without the need to transfer any state from another device.

3.2 Threat Model

The threat model defines the adversaries and the possible attacks that can be performed to compromise the privacy guarantees of our system. Specifically, with HITC, we want to protect the privacy of shared data against the following threat actors:

- **OSN service provider:** HITC protects the privacy of data shared over OSNs by means of encryption. Naturally, an OSN service provider might attempt to break the encryption algorithm and retrieve the raw data. Such attempts might be for the sake of collecting user data for analytics purposes (e.g., studying users' behaviors) [19], displaying customized content (e.g., customized advertisements), providing user data to authorities (e.g., law enforcement, governments) [16], or even selling user data and information to third parties [21]. A malicious OSN may also try to launch a man-in-the-middle (MITM) attack, by modifying the stego images that store the public keys of the users. Finally, in this work, we do not address denial of service (DoS) attacks, where the OSN provider deliberately alters all posted images to sabotage HITC's operation.
- **Curious OSN user:** An OSN user who is not granted access to data shared by another user might get curious to see what the user has shared, and try to conduct attacks to gain unauthorized access to such protected data. Similar to the case of the OSN provider, this attack involves the compromise of the underlying encryption algorithm. We allow users to collude with each other and with the OSN provider, by sharing secret information about other users (such as passphrases). However, we do not consider the disclosing of content decryption keys as an attack, because any user can share the content they have access to with others, if they choose to do so (i.e., this is not a weakness of HITC).

3.3 System Architecture

The architecture of our system (Figure 1) consists of three major components that interact with each other. We describe each component below.

(1) **End-user:** The end-user is the OSN user who is registered with an OSN service and has a device that is connected to the Internet.
(2) **Web browser:** A web browser software that is installed on the end-user's device to surf the world wide web (i.e., access the OSN's web page). Moreover, today's browsers can include *extensions* that act as add-ons, which provide extra features to the browser's built-in functionalities. In our system, all the operations (e.g., encryption and decryption) are performed at the client-side by an extension within the browser itself. Additionally, browsers provide different flavors of local storage systems (e.g., local storage, session storage, indexed databases, Web SQL, cookies). We utilize the local storage of the web browser to store data locally in the end-user's device.
(3) **OSN services:** Our system interacts with OSN services that consist of the OSN service provider's servers (e.g., web servers), storage systems, and application layer components (e.g., application programming interfaces, APIs).

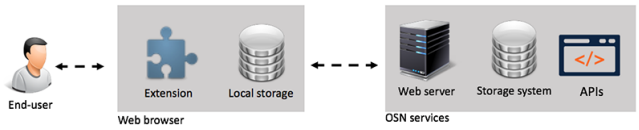


Figure 1: HITC architecture

3.4 Operations

HITC supports six main operations that are independent of the underlying OSN platform. Each operation is discussed in detail in the following paragraphs and, to make the discussion clear and understandable, we summarize the notations used in Table 1.

REGISTER Operation: The REGISTER operation is the user registration phase in HITC. It's main purpose is to initialize the user's secrets: a strong password, a secret passphrase, and a pointer to a posted cover image that stores his RSA public key. Specifically, the following actions are performed:

(1) User u's OSN unique identifier (username), $u.id$, is read. This identifier must be unique for each registered user in the HITC system.
(2) User u manually enters a password, $u.pw$, a secret passphrase, $u.sp$, and a pointer (e.g., a string or a hashtag), $u.ptr$.
(3) The password hash, $H(u.pw)$, is computed using the SHA512 cryptographic hash function and a 2048-bit RSA public/private key pair, $u.rsa.PK$ and $u.rsa.SK$, respectively, is generated automatically using $H(u.pw)$ as an input to the key generation function (PRNG random seed).
(4) User u's RSA public key is then embedded into the cover image CI (selected by the user), using image steganography. The result is a stego image SI: $SI \leftarrow CI.Embed(u.rsa.PK, H(u.sp))$. The stego image is finally posted on the OSN platform and tagged with $u.ptr$, user u's pointer. The hash of $u.sp$ is used as a seed to a PRNG to ensure that data is embedded into the cover image in a random manner, i.e., not in fixed locations.

Note that, the password hash is used as a random seed to a PRNG that generates all the user's keys (both RSA and HVE). As such, it should be a strong password that is hard to brute force. On the other hand, the role of the pointer and secret passphrase is to hide the user's public key inside a random cover image on his social media feed, in order to thwart MITM attacks. Indeed, as we will discuss shortly, RSA public keys are used to distribute HVE decryption keys, so a malicious OSN provider may try to replace a user's public key with their own in order to obtain the decryption keys sent by other users. The uncertainty of the selected CI and the randomness of the embedding pattern of the key inside the CI, make such an attack infeasible (the adversary can not verify whether his guess is correct).

LOGIN Operation: This operation is performed once, when the user logs into the HITC system for the first time (i.e., on a new device). The following actions are performed:

(1) User u manually enters a password, $u.pw$, a secret passphrase, $u.sp$, and a pointer, $u.ptr$.

Table 1: HITC design notations

Term	Definition
u	User u is a registered user in the OSN
$u.id$	User u's unique identifier
$u.pw$	User u's password
$u.sp$	User u's secret passphrase
$u.ptr$	User u's pointer
$u.s$	A shared random PRNG seed generated by user u
$u.rsa.PK$	User u's RSA public key
$u.rsa.SK$	User u's RSA private key
$u.hve.PK$	User u's HVE public key
$u.hve.MK$	User u's HVE master key
$u.hve.DK[v]$	User v's HVE decryption key generated by user u
$u.relationshipId[v]$	A relationship id $\in \{0, \ldots, L-1\}$ generated by user u and associated with user v.
$u.S[v]$	Sender token generated by user u (the sender) associated with user v (the receiver)
$u.R[v]$	Receiver token generated by user u (the sender) associated with user v (the receiver)
$u.secret$	Data object shared on the OSN platform by user u
$u.secret.AV$	An access vector of L elements constructed by user u to control access to $u.secret$
$SI \leftarrow CI.Embed(d, s)$	Use image steganography to embed data d into cover image CI, where s is a seed to a PRNG function to ensure random data embedding
$d \leftarrow SI.Extract(s)$	Use image steganography to extract data d from a stego image SI, where s is the PRNG seed

(2) The password hash, $H(u.pw)$, is computed using the SHA512 cryptographic hash function and a 2048-bit RSA public/private key pair, $u.rsa.PK$ and $u.rsa.SK$, respectively, is generated automatically using $H(u.pw)$ as an input to the key generation function.
(3) Following $u.ptr$, the stego image that holds $u.rsa.PK$ is retrieved. Then, using $H(u.sp)$ as a PRNG seed, $u.rsa.PK$ is extracted from the stego image and compared to the public key generated in Step (2). If it matches, user u is successfully authenticated and can perform the other system's operations correctly.
(4) With the *hve.Setup* function described in Section 2.1, user u generates his HVE master and public keys, $u.hve.MK$ and $u.hve.PK$, respectively. These keys are deterministically generated using $H(u.pw)$ as a seed to the PRNG function.
(5) In order for user u to decrypt his own HVE-encrypted data posted on the OSN platform, he generates an HVE decryption

key for himself, $u.hve.DK[u]$. For that purpose, we reserve the relationship ID 0 (i.e., $u.relationshipId[u] = 0$), which cannot be assigned to any other user.

(6) A shared random seed $u.s$ is generated by computing the hash of the concatenation of the user's RSA private key and his password, $u.s = H(u.RSA.SK||u.pw)$, using the SHA512 hash function.

(7) All the following data objects are stored locally in user u's device: $H(u.pw)$, $u.rsa.PK$, $u.rsa.SK$, $u.s$, $u.hve.PK$, $u.hve.MK$, and $u.hve.DK[u]$.

In a nutshell, the LOGIN operation generates and stores locally all the keys that are necessary for the correct operation of HITC. In addition, Step (3) acts as an authentication mechanism that verifies the identity of the HITC user. The shared PRNG seed $u.s$ is utilized in the access control phase of our protocol, and is distributed to all users that have established a social relationship (on the HITC system) with user u.

ESTABLISH_RELATIONSHIP Operation: This operation is invoked when user u (the sender) wants to establish a relationship with user v (the receiver). It basically involves user u generating and sharing an HVE decryption key with user v. The following actions are performed:

(1) User u assigns a random relationship ID $u.relationshipId[v] \in \{1, \ldots, L-1\}$ to user v. Next, using $u.relationshipId[v]$, $u.hve.MK$, and $u.hve.PK$ as inputs to the *hve.GenDecKey* function, user u generates an HVE decryption key for user v: $u.hve.DK[v]$.

(2) User u constructs a sender token, $u.S[v]$, which is the encryption of the concatenation of $v.id$ and $u.relationshipId[v]$ with a symmetric cipher (AES), using $H(u.pw)$ as the key.

(3) Users v and u establish an out-of-band communication channel (e.g., face-to-face communication or through a secure messaging app) where user v reveals his secret pointer, $v.ptr$, and secret passphrase, $v.sp$, to user u. Following $v.ptr$, user u retrieves the stego image SI where user v has embedded his own RSA public key, and utilizes $H(v.sp)$ to extract it: $v.RSA.PK \leftarrow SI.Extract(H(v.sp))$.

(4) User u constructs a receiver token, $u.R[v]$, by RSA encrypting the following information with $v.rsa.PK$: $u.relationshipId[v]$, $u.id$, $v.id$, $u.hve.DK[v]$, and $u.s$.

(5) User u embeds the sender token, $u.S[v]$, in the first half of a selected cover image CI, using $H(u.pw)$ as a seed for the random embedding sequence. In addition, he embeds the receiver token, $u.R[v]$, into the second half of the image, using $H(v.rsa.PK)$ as a seed for the random embedding sequence. The resulting stego image, SI, is posted by user u on the OSN platform after being *tagged* with a reference to user v.

This operation presents a malicious OSN provider (or an adversary that has gained unauthorized access to the OSN servers) with an opportunity to intercept a user's HVE decryption key(s) through a MITM attack. Specifically, if an adversary has knowledge of a user's secret pointer and passphrase, they can replace the user's RSA public key with their own, thus getting access to any future HVE key shared with the user. While this attack necessitates an exact guess by the adversary without the possibility of a verification ahead of time, it becomes very feasible if a user's social friend colludes with the adversary. Therefore, to mitigate such attacks, the user should immediately modify the secret passphrase after sharing it via the ESTABLISH_RELATIONSHIP operation. This can be done by invoking the REGISTER operation.

REFRESH_RELATIONSHIPS Operation: This is a data collection operation performed by user u, in order to gather (i) the HITC social relationships information (where user u acts as either a sender or a receiver) and (ii) all the HVE decryption keys assigned to him by other users. The following two tasks are performed:

- **Sender tokens collection:** User u progressively retrieves images shared by himself over the OSN platform (in which other users are tagged) and checks whether a valid sender token, $u.S[v]$, exists, where v is a user with whom user u has already established a relationship. The validity of the sender token is confirmed by the successful decryption of the token using $H(u.pw)$ as the AES key. Once the token is decrypted, user u stores $v.id$ and $u.relationshipId[v]$ locally.
- **Receiver tokens collection:** User u progressively retrieves images shared with him by other users (i.e., u is tagged) over the OSN platform and checks if a valid receiver token, $v.R[u]$, is embedded in the image shared by user v. The validity of the receiver token is confirmed by the successful decryption of the token using his RSA private key $u.rsa.SK$. Once the token is decrypted, user u stores $v.id$, $v.relationshipId[u]$, $v.hve.DK[u]$, and $v.s$ locally.

This operation is invoked when (i) the user logs into HITC from a new device and (ii) after the user participates in an ESTABLISH_RELATIONSHIP operation as a receiver, i.e., to retrieve a new HVE decryption key.

POST_SECRET Operation: This operation implements the access control mechanism of HITC. It is invoked when user u wants to enforce HITC's stringent access control protocol on an arbitrary object, $u.secret$, that is posted on the OSN platform. To achieve that, the following actions are performed:

(1) User u generates a random AES key k and encrypts $u.secret$: $E_k(u.secret)$.

(2) User u constructs $u.secret.AV$, the access vector with L elements, where L is the maximum number of supported friends. For each user v that has a relationship with user u (where v acts as the receiver), user u can either grant or deny access to the private data, by setting the value of the access vector at position $u.relationshipId[v]$ as follows:
   - **Grant access:** Set the value to '$\star$'.
   - **Deny access:** Set the value to '0'.

   Note that, user u always sets $u.secret.AV[0] =$ '$\star$', in order for him to decrypt his own encrypted data.

(3) User u invokes the $hve.Enc$ function to encrypt the secret key k: $C \leftarrow hve.Enc(u.hve.PK, u.secret.AV, k)$.

(4) User u constructs an object $SECRET$, by concatenating C, $E_k(u.secret)$, and $u.secret.AV$.

(5) User u embeds $SECRET$ into a randomly selected cover image, CI, and shares the resulting stego version of the image, SI, over the OSN platform. The shared PRNG seed, $u.s$,

is used to generate the random data embedding sequence: $SI \leftarrow CI.Embed(SECRET, u.s)$.

To summarize, access control is enforced by AES encrypting the shared object and giving access to the underlying key through HVE decryption. Also note that we do not need to post the entire access vector $u.secret.AV$ on the OSN platform, since the number of friends τ granted decryption privileges would typically be $\tau \ll L$. Therefore, it suffices to post only the relationship IDs of the users that have been granted access. Finally, we should point out that, due to the randomness of the embedding sequence, it is very difficult for the OSN provider to identify HITC-related content. On the other hand, since all of user u's friends have access to the shared seed, $u.s$, they can immediately recognize HITC's stego images by identifying the access vector. As a result, if any of user u's friends discloses $u.s$ to the OSN provider, it would enable it to identify (i) the stego images and (ii) the relationship IDs associated with the hidden content (but not the actual users). Nevertheless, we do not view this as a serious weakness, because we assume that users would only distribute HVE decryption keys to trusted social friends.

EXTRACT_SECRET Operation: This operation is performed on a stego image, SI, in order to retrieve the embedded secret, $v.secret$. It involves the following actions:

(1) User u invokes the $SI.Extract$ function to retrieve the embedded $SECRET$ object from a stego image SI shared by user v: $SECRET \leftarrow SI.Extract(v.s)$.
(2) Once $SECRET$ is successfully retrieved, user u scans the access vector, $v.secret.AV$, to see if he has been granted access, i.e., whether $v.relationshipId[u]$ appears on the access list. If not, user u aborts the operation.
(3) If user u is granted access to the secret, he invokes the $hve.Dec$ function to decrypt the HVE ciphertext C (with $v.hve.DK[u]$ as the key) and retrieve the AES key k.
(4) User u extracts $v.secret$ by decrypting $E_k(v.secret)$.

4 IMPLEMENTATION

In this section, we present our prototype implementation of HITC on the Twitter platform.

4.1 Implementation Overview

We built a fully functional HITC implementation in the form of a Chrome browser extension and applied it on the Twitter platform as a proof-of-concept. Our source code is publicly available on GitHub [5]. The reason for choosing Twitter is twofold. First, Twitter offers a very coarse-grained access control to its users; the user can either make his account publicly available or completely private. Therefore, we want to give users the ability to enforce fine-grained access control over their tweets. Second, Twitter allows the posting of PNG images that are not modified when uploaded. As a result, we do not need to employ error correcting codes in the stego images.

We implemented the browser extension with the client-side JavaScript language. The reason for implementing the system as a browser extension is the simplicity and ease of use when it comes to the average user. Furthermore, browser extensions depend on the browser software itself, which is already installed on the end-user's device and is used to navigate through the OSN website. Therefore, HITC does not require any additional software or configuration on behalf of the user. For storing HITC related data (keys, passphrases, relationships, etc.), we utilized the local storage of the Chrome browser.

The image steganography technique adopted in our implementation is the *spatial domain* technique. For each pixel within a colored image, there are three channels: red, green, and blue. Each channel corresponds to an 8-bit value, where we embedded one bit of secret data into one of 4 LSBs, chosen randomly. For the cryptographic protocols, we employed the following cryptographic libraries: *crypto-js* [32] for SHA512 hashing and 256-bit AES encryption, and *cryptico* [29] for 2048-bit RSA encryption. We implemented our own CP-HVE library by leveraging the *mcl* pairing-based cryptographic library [26].

4.2 Operations Implementation

We now introduce and discuss our implementation, starting from a user registering with HITC until he communicates (e.g., shares and reads) private data objects over the Twitter platform.

Twitter authentication and authorization: Before proceeding with HITC's basic operations, the extension needs to get permission from the user to access his Twitter account. To this end, once the user clicks on the extension icon next to the browser's address bar, a page is displayed asking the user to authenticate and authorize the extension to have access permission to the user's Twitter account, as shown in Figure 2. This step is performed in order to verify that the user is authorized to access this specific Twitter account, thus avoiding impersonation and identity-stealing attacks. Furthermore, it gives HITC the authority to make Twitter API calls on behalf of the user.

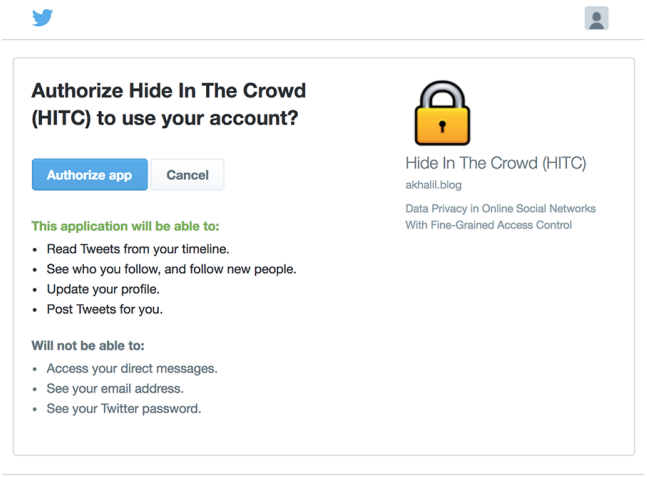


Figure 2: HITC asking for Twitter authorization

User registration: The user provides a *strong* password that he must remember for future logins (e.g., after logging out or to login from a different device), a secret passphrase, a hashtag, and a cover image from his local device, as illustrated in Figure 3. Once this information is provided, the REGISTER operation is invoked that

leverages the Twitter API to post the stego image containing the user's RSA public key. The post is marked with the provided hashtag, which serves as a pointer that is later shared with other users, in order for them to retrieve the RSA public key of the user during the relationship establishment operation.

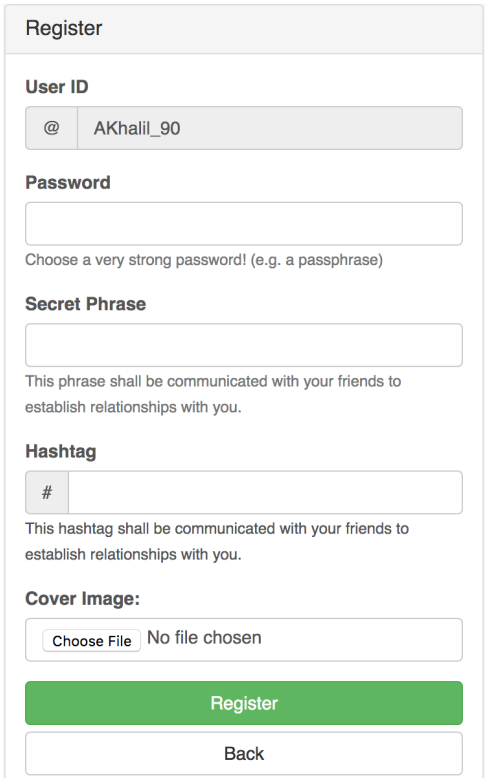


Figure 3: HITC registration popup screen

User login: An already registered user can login to HITC by providing his password, secret passphrase, and hashtag that were used in the registration process. The login popup screen where this information is entered is very similar to the one in Figure 3. Next, the extension utilizes the hashtag to pull the tweet that includes the image storing the user's RSA public key. Following the successful authentication of the user (as explained in the LOGIN operation) the rest of the data objects (keys, random values) are generated and stored locally.

Relationship establishment: After a successful login, the main popup screen of the HITC extension includes two buttons for managing relationships. With the first one, the user can establish a relationship with user v, by entering user v's unique identifier (i.e., Twitter screen name) and secret passphrase, the hashtag linked to user v's RSA public key, and a random cover image. The ESTABLISH_RELATIONSHIP operation is then invoked, as described in Section 3.4, that pulls the image tweeted by user v and extracts user v's public key, in order to encrypt the receiver token. After that, both the sender and receiver tokens are constructed and then embedded into the cover image. The resulting stego image gets tweeted, and user v is explicitly *mentioned* in that tweet.

Refresh relationships: The second button on the main extension screen allows the user to invoke the REFRESH_RELATIONSHIPS operation. Once started, the operation searches (using Twitter's API) for all tweeted images by the logged user (to collect sender tokens), and all tweeted images by other users where the logged user is mentioned (to collect receiver tokens). The extracted relationship information is stored locally in the browser.

Sharing secret: For a logged in HITC user, as illustrated in Figure 4(a), an extra "Secret Tweet" button is added to the original Twitter page. Once clicked, a Twitter-like box is displayed, as shown in Figure 4(b). From this box, the user can choose the type of data object to share: a *text*, an *image*, or a *file*. Once the user selects the actual data object he wants to share, another box is displayed that allows the user to choose the social friends that are granted access to this object. This model box is depicted in Figure 4(c). As each data object is transformed into its binary representation before being embedded into the cover image, HITC can support the sharing of private files with arbitrary format (e.g., .pdf, .exe), even though Twitter and almost all other OSNs, do not allow this. We view this as an inherited added feature for HITC users.

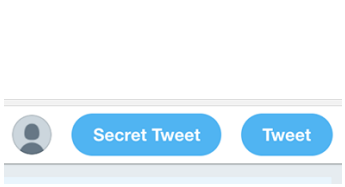
Extracting secret: Extraction of secrets is a seamless process that is performed while surfing the Twitter pages. The extension continuously reads the HTML markup of the pages and identifies tweets that are posted by users that have sender relationship with the logged in user (i.e., the logged in user is the receiver in this relationship). Any image in these tweets is investigated for steganography, by performing the EXTRACT_SECRET operation. The extension marks such images with an informational message that is being displayed beneath the image. The informational messages cover the following cases: (i) a secret is not found, (ii) a secret is successfully extracted, or (iii) no access is granted to that secret. These messages are illustrated in Figure 4(d). On the successful extraction of a secret, the user can view the secret data by hovering the mouse over the stego image. We choose the mouse hovering action to reduce the risk of *shoulder surfing* attacks. In the case where the protected data object is an arbitrary file, once the image is hovered and clicked, the protected file is automatically downloaded on the end-user's device.

5 EVALUATION

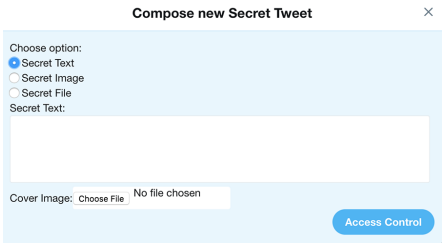
This section presents the results of our performance evaluation that measures the computational overhead of HITC.

5.1 Operations Performance Measurement

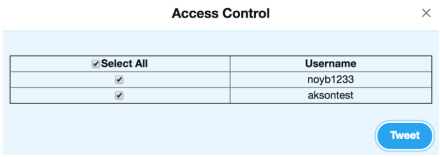
In our experiments, we focus on evaluating the cost of the various cryptographic primitives involved in each of the system's operations, as discussed in Section 3.4. In all measurements, we used a 1200×1200 cover image (the largest accepted dimension of an image without invoking re-sizing or cropping operations by Twitter), a 300×300 secret image, a 2048-bit RSA key pair, and the following parameters for the HVE functions: elliptic curves on 256-bit prime order groups (see Appendix A), $L = 1000$ (maximum number of friends), $N = 20$ (maximum number of friends with access to a



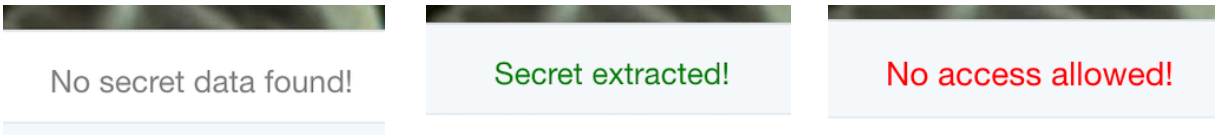
(a) "Secret Tweet" button is displayed on Twitter page



(b) Twitter-like model box to share a private data object



(c) Access vector construction



(d) Informational messages displayed beneath the shared image

Figure 4: HITC's access control interface

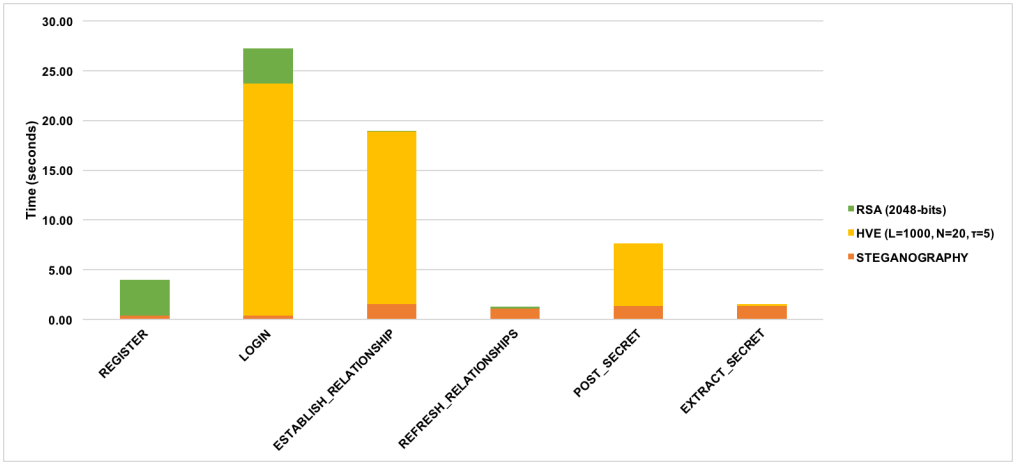


Figure 5: Response time of HITC operations

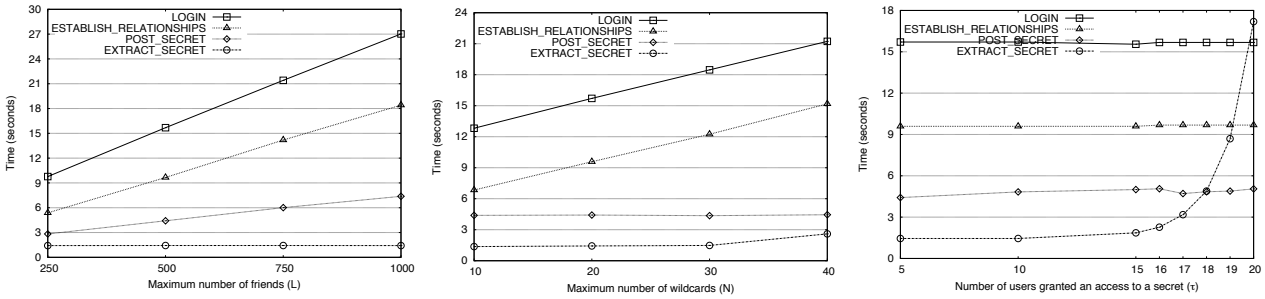
single secret), and $\tau = 5$ (number of friends granted access to a single secret). We ran HITC on a laptop equipped with a 2.7GHz Intel Core i5 processor and 8GB of RAM.

We intentionally left out the measurements for posting/retrieving images on/from the OSN, as these processes depend on many variables, including internet connection, network latency, caching, etc. Moreover, we do not measure the overhead of hashing (SHA512) or symmetric key (256-bit AES) operations, because their impact is negligible compared to public key and hidden vector encryption operations. The results are summarized in Figure 5 and are discussed in the following paragraphs.

REGISTER: The time to generate the RSA keys is 3.6 seconds, while the embedding of the data into the cover image requires less than 0.5 seconds. Even though RSA key generation is an expensive operation, it is performed very rarely and does not affect the user's browsing experience.

LOGIN: Overall, the LOGIN operation is the most time consuming operation in HITC, with a total cost of 27.3 seconds. Nevertheless, this is not a major concern, because the LOGIN operation is performed only once per device. The HVE-related functions, *hve.Setup* and *hve.GenDecKey*, are the most expensive ones, contributing 23.3 seconds to this process. On the other hand, the time to generate the RSA keys and the time to process the image steganography function are equivalent to the ones reported above.

ESTABLISH_RELATIONSHIP: The relationship establishment cost is dominated by the cost of generating an HVE decryption key, which takes approximately 17.4 seconds. The RSA encryption of the receiver token, the AES encryption of the sender token, and the embedding of data into the cover image take a total of 1.6 seconds. Again, the high computational cost of this operation is not a serious concern, because it is only invoked when a new relationship is established under HITC (only close social friends would typically



(a) Response time vs. L (b) Response time vs. N (c) Response time vs. τ

Figure 6: Response time for varying HVE parameters

get an HVE decryption key). Note that, the steganography-related functions are more expensive compared to the ones reported in the LOGIN operation. This is due to the size of the HVE decryption keys that is significantly larger than RSA keys (Appendix A).

REFRESH_RELATIONSHIPS: The total average time required for this operation is 1.3 seconds and is dominated by the extraction of embedded data in the stego images. The time reported in Figure 5 corresponds to the average time required to process two relationships: one where the user acts as a sender and the other as a receiver. This is an operation that is not executed frequently and does not affect the normal operation of HITC.

POST_SECRET: The HVE encryption function when posting a secret takes an average of 6.3 seconds, while embedding the private data into the cover image necessitates 1.4 seconds of processing time. Even though encryption is a time consuming function, the POST_SECRET operation is not the bottleneck of HITC's performance. Indeed, social media content follows the *write once*, *read many* model where, in our case, every user's tweet appears in the Twitter feed of all his followers. As such, the most critical operation that dictates the user's experience with HITC is EXTRACT_SECRET, which is discussed next.

EXTRACT_SECRET: This is the most frequently executed operation in HITC, which runs continuously as the user scrolls through his Twitter feed. Fortunately, it is also very efficient; data extraction from the stego image takes 1.4 seconds, and the HVE decryption of the AES key costs only 0.1 seconds. Note that the data extraction time reported here corresponds to a "hit," i.e., the stego image contains decipherable data. In the general (and most frequent) case, the EXTRACT_SECRET operation would terminate early if (i) there is no secret data embedded into the image or (ii) the access vector does not include the user's ID.

5.2 HVE Performance Measurement

Given the large overhead of HVE-related functions (Figure 5), we next assess the performance of HITC's operations when modifying the underlying HVE parameters: L, N, and τ. The results are summarized in Figure 6, where the response time is measured on a live system, i.e., it includes the cost of the steganographic operations as well. First, Figure 6(a) illustrates the response time as a function of the maximum number of friends, L ($N = 20, \tau = 5$). The LOGIN, ESTABLISH_RELATIONSHIP, and POST_SECRET operations have a cost that grows linearly with L, because the corresponding HVE functions *hve.Setup*, *hve.GenDecKey*, and *hve.Enc*, respectively, are all linear in L (Appendix A). On the other hand, the EXTRACT_SECRET operation is not affected, since the *hve.Dec* function is independent of L.

Figure 6(b) shows the response time as a function of N, the maximum number of wildcards allowed in the encryption vector ($L = 1000, \tau = 5$). As shown in Appendix A, the only function that is affected by N is *hve.GenDecKey*, i.e., the function that generates decryption keys. As such, the POST_SECRET and EXTRACT_SECRET operations remain mostly unaffected. The reason why the cost of the LOGIN operation also grows linearly with N, is because it includes the generation of an HVE decryption key (for the user to decrypt his own posts).

Finally, Figure 6(c) depicts the response time of the four operations as a function of τ, the number of wildcards in the encryption vector ($L = 1000, N = 20$). As evident in Appendix A, only the *hve.Dec* function depends on τ and, as such, all other operations exhibit constant cost. It is worth noting that the exponential growth of the cost is due to the computation of Viete's formulas, where $a_{\tau-k}$ necessitates the enumeration of all k-subsets out of τ elements. Nevertheless, the decryption function is very efficient for groups of up to 15 friends.

6 DISCUSSION

In this section, we discuss the limitations of HITC and highlight some research directions that we plan to investigate in the future.

6.1 Key Revocation

HITC does not support key revocation. We do not consider it an essential function, because access control is performed on the finest granularity level (no grouping of members). As a result, there is no need to explicitly revoke a user's key; it suffices to exclude that user from the access control vectors of future shared objects. The downside of this approach, is that a revoked user can still access all the data for which he was given decryption privileges in the past. To avoid that, the straightforward (but not practical) solution is to re-encrypt every object that the user had access to. Nevertheless,

we do not view this as a serious privacy concern, because users can easily collect and store private data while authorized, and retrieve them locally even after their keys are revoked.

6.2 Embedded Data Survival

In our implementation, we chose to embed secret data into PNG images, which is a lossless compression format. However, some existing OSNs, such as Facebook, only support JPEG images on their platforms. In addition to JPEG's lossy compression format, these OSNs further manipulate the uploaded images and may destroy the embedded data (e.g., Twitter manipulates uploaded JPEG images). To that end, in our future work, we aim to employ a more robust image steganography technique (by employing Reed-Solomon codes) that is able to survive the image processing performed by the OSN service providers.

6.3 Secret Data Capacity

In the evaluation reported on Section 5, we randomly embedded one bit of secret data into one of the 4 LSBs of a pixel's channel value, i.e., the overall secret data capacity is 3 bits/pixel. As a result, the maximum amount of data that we can embed into a 1200×1200 cover image is 520KB. To increase the secret data capacity, we could utilize more bits per channel, but this may ultimately introduce noticeable changes in the image itself. Such changes may violate our low-profile requirement, and make HITC visible to the OSN users and service providers.

6.4 Access Vector Visibility

The HVE protocol that we employed necessitates the attachment of the cleartext access vector associated with each ciphertext. Due to the random embedding sequence with a shared secret seed, the OSN service provider is unable to identify and read the access vector and is, thus, oblivious to the underlying data sharing. However, all users that have a receiver relationship with the sender of the protected data can read the contents of the access vector. An inherited limitation is that any user with access to the shared secret seed of another user u can read the access vector constructed by user u and identify the number (and IDs) of users who are given access to a specific object. Nevertheless, this is not a serious privacy concern, because user IDs can not be mapped directly to OSN users (recall that relationship IDs are assigned randomly). Furthermore, the access vector itself does not leak any information regarding the secret object.

6.5 APIs Availability

Our implementation relies on Twitter APIs to perform image search and post new images. To implement HITC in an OSN other than Twitter, only minimal changes are required, in order to utilize the other OSN's APIs. Moreover, the process of posting images, tagging, and reading them needs to be independently implemented for each OSN (e.g., write on a user's wall for Facebook). However, all the changes are introduced in the implementation part only, without affecting the design presented in Section 3. On the other hand, using the OSN's APIs can be considered a drawback if the OSN decides to disable them or dramatically limit their usage. Nevertheless, disabling APIs is unlikely, because OSNs are encouraging third-party developers to integrate their applications and services with the OSNs' platforms, in order to have a larger exposure and reach a larger user base.

6.6 DoS Attacks

One of our main goals is to prevent malicious OSNs from observing and disrupting our system's activities. Due to our random embedding sequence, it may be difficult for an OSN to identify whether an image includes hidden data or not. However, an OSN with knowledge of HITC's protocols will realize that the secret data is embedded into one of the 4 LSBs in each pixel's channels. The OSN may then launch a preemptive DoS attack, by flipping all 4 LSBs in each channel for every image uploaded by its users (regardless of whether they actually use HITC). Obviously, such an attack would sabotage HITC's operation, but we do not believe it is a viable solution on behalf of the OSN provider. Indeed, such aggressive pixel modifications would severely affect the quality of the uploaded images, thus frustrating the user base and possibly forcing them to leave the platform.

6.7 Survival Against Steganalysis

Another defense against HITC is for the OSN service provider to apply steganography detection techniques on the uploaded images, and ban all images that are flagged as suspicious. To this end, we tested 10 random images (shown in Figure 7) that are produced by our system, against two PNG image steganalysis tools: *StegExpose* [11] and *zsteg* [3]. The StegExpose tool inspects the input image and flags it as suspicious or not. On the other hand, the zsteg tool tries to extract data embedded into an input image using different patterns. The results showed that both tools failed to mark the inspected images as suspicious or extract any embedded data. As a result, we are confident that HITC can remain undetected by the OSN service providers.

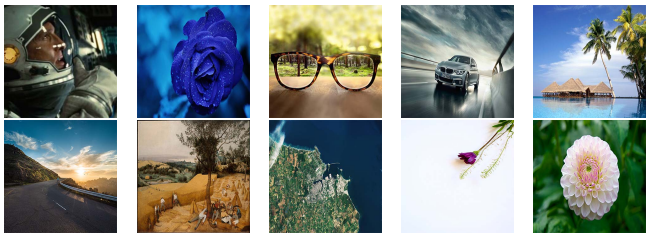


Figure 7: Random stego images tested by steganalysis tools

7 RELATED WORK

Persona [9] is an OSN that replicates Facebook's functionalities, but adds encryption-based access control, by combining both public-key and attribute-based encryption (ABE) [10], where users are able to enforce their own policies. Moreover, to avoid centralized storage at the OSN site, Persona uses decentralized persistent storage which enables its users to store their data with intermediaries without the need to trust those intermediaries.

EASiER [18] is similar to Persona, in that leverages ABE to provide encryption-based access control. Its novelty is that it provides

an efficient key revocation procedure, which is the major limitation in Persona. Revocation necessitates a trusted proxy that enforces the revocation constraints by participating in the decryption process. In particular, the proxy is initially assigned a secret key with revocation details by the data owner. Then, before performing the decryption process, a user would share part of the encrypted data with the proxy. The proxy would then use its key to extract the necessary information that only non-revoked users can use to successfully complete the decryption operation.

Sun et al. [28] introduced a private OSN with an efficient revocation mechanism for restricting access to personal data that reside within an untrusted storage service. For sharing private data with other OSN users, a privacy-preserving construction is performed that allows the owner of the data to implement access control over the shared data. The data owner acts as a group manager by classifying contacts according to their role (e.g., co-workers, friends) and grants contacts a membership to those groups. The proposed scheme provides the data owner with a convenient way to cope with membership changes without the need to rebuild the group or re-key group members. It also includes an efficient revocation mechanism against changing group membership that has minimal impact on data privacy.

Hummingbird [14] is a privacy-preserving Twitter-like microblogging OSN. It adds several cryptographic protocols, such as blind RSA signatures [12], that allow users to post tweets and follow users with privacy. Specifically, Hummingbird introduces the following privacy features: (i) it allows the tweeter to apply fine-grained access control on his tweets and (ii) it offers privacy to followers, i.e., followers can subscribe to hashtags without sharing their interests with any entity.

Safebook [13] is a decentralized OSN for privacy-preserving applications. It exploits real-life trust and integrates several privacy and security mechanisms that provide data storage and data management functions to its users. The authors also examine the architecture of existing social network services, in order to assess and analyze their security and privacy threats.

NOYB [17], which stands for *none of your business*, is a scheme that provides privacy in existing OSNs. It is based on the observation that existing OSNs are not verifying the data shared over their platform for authenticity. If users can map shared fake data back into real data, they can still use the OSN service while limiting the correct mapping operation to authorized users only. NOYB partitions private data into multiple *atoms* and then replaces each atom with a "fake" atom, using external dictionaries (for that, it relies on distributed store-lookup infrastructures [25]). A proof-of-concept NOYB was implemented for Facebook as a Firefox web browser extension that interprets and modifies Facebook pages.

Lockr [31] is a system that enhances the level of privacy in both centralized and decentralized content sharing networks. It offers the following privacy benefits to existing OSN users: (i) separation of the content shared over an OSN's platform from all other functionalities that an OSN service provider offers, (ii) employment of digitally signed social relationships that can not be re-used by the OSN to access social data, and (iii) encryption of messages using social relationship keys. Lockr has been implemented for both Flicker and BitTorrent.

flyByNight [22] is a system that mitigates the privacy risks of Facebook. It was implemented as a native Facebook application with the aim of encrypting and decrypting secret data at the client-side. flyByNight benefits from Facebook in managing users' social relationships and relies on Facebook's interface for key management. It supports both one-to-one and one-to-many communications and ensures that cleartext data and private keys are never stored on Facebook's servers.

Ning et al. [23] proposed a framework for private communications in OSNs by establishing covert channels in the existing OSNs' infrastructure. The idea is to use steganography to embed secret messages in cover images before posting them on the OSN platform. The authors made three key contributions in this field. First, they analyzed the effects of image processing done by photo-sharing OSNs on the embedded secret messages. Second, they proposed a simple change to the traditional image steganography techniques, which ensures that secret messages survive the image processing introduced by the OSN service providers. Finally, they proposed a bootstrapping protocol for in-band communication in order to share encryption keys and perform key exchange.

8 CONCLUSIONS

In this paper, we introduced the design and implementation of Hide in the Crowd (HITC), a system that incorporates encryption-based access control to existing OSN platforms. HITC offers many advantages compared to previous approaches. First, it supports dynamic group formation, by assigning decryption privileges to individual users in an interactive manner, when a new object is posted on the OSN platform. Second, HITC employs in-band mechanisms for key exchange, with minimum out-of-band communication for relationship establishment, and uses image steganography to hide keys and shared objects within the OSN servers. As such, it does not rely on third parties to operate successfully. Third, it supports stateless mobility and can be installed on different devices without the need to transfer any piece of information. Finally, HITC is implemented as a browser extension which, in combination with the underlying steganographic techniques, makes HITC invisible to unauthorized OSN users and the OSN service provider itself. To illustrate the feasibility of our design, we implemented a proof-of-concept system that provides fine-grained access control over the Twitter platform. The results of our evaluation show that HITC is scalable and has a minimal impact on the user's experience.

REFERENCES

[1] 2014. Camouflage. http://camouflage.unfiction.com/.
[2] 2014. JpegX Software. http://www.freewarefiles.com/Jpegx_program_19392.html.
[3] 2014. zsteg. https://github.com/zed-0xff/zsteg.
[4] 2017. Facebook Stats. https://newsroom.fb.com/company-info/.
[5] Ahmed Khalil Abdulla. 2019. HITC source code. https://github.com/AKhalil90/HITC-Hide-In-The-Crowd/.
[6] Alessandro Acquisti and Ralph Gross. 2006. Imagined Communities: Awareness, Information Sharing, and Privacy on the Facebook. In *Proc. Workshop on Privacy Enhancing Technologies (PET)*. 36–58.
[7] Paul Alvarez. 2004. Using Extended File Information (EXIF) File Headers in Digital Evidence Analysis. *IJDE* 2, 3 (2004).
[8] Salman Aslam. 2019. Twitter by the Numbers: Stats, Demographics and Fun Facts. https://www.omnicoreagency.com/twitter-statistics/.
[9] Randolph Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee, and Daniel Starin. 2009. Persona: an online social network with user-defined privacy. In

Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM). 135–146.
[10] John Bethencourt, Amit Sahai, and Brent Waters. 2007. Ciphertext-Policy Attribute-Based Encryption. In *Proc. IEEE Symposium on Security and Privacy (S&P)*. 321–334.
[11] Benedikt Boehm. 2014. StegExpose - A Tool for Detecting LSB Steganography. *CoRR* abs/1410.6656 (2014). http://arxiv.org/abs/1410.6656
[12] David Chaum. 1982. Blind Signatures for Untraceable Payments. In *Proc. CRYPTO*. 199–203.
[13] Leucio Antonio Cutillo, Refik Molva, and Thorsten Strufe. 2009. Safebook: a privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine* 47, 12 (2009), 94–101.
[14] Emiliano De Cristofaro, Claudio Soriente, Gene Tsudik, and Andrew Williams. 2012. Hummingbird: Privacy at the Time of Twitter. In *Proc. IEEE Symposium on Security and Privacy (S&P)*. 285–299.
[15] Ralph Gross and Alessandro Acquisti. 2005. Information revelation and privacy in online social networks. In *Proc. ACM Workshop on Privacy in the Electronic Society (WPES)*. 71–80.
[16] Joshua Gruenspecht. 2011. "Reasonable" grand jury subpoenas: asking for information in the age of big data. *Harvard Journal of Law Technology* 24, 2 (2011).
[17] Saikat Guha, Kevin Tang, and Paul Francis. 2008. NOYB: privacy in online social networks. In *Proc. ACM Workshop on Online Social Networks (WOSN)*. 49–54.
[18] Sonia Jahid, Prateek Mittal, and Nikita Borisov. 2011. EASiER: encryption-based access control in social networks with efficient revocation. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 411–415.
[19] Long Jin, Yang Chen, Tianyi Wang, Pan Hui, and Athanasios V. Vasilakos. 2013. Understanding user behavior in online social networks: a survey. *IEEE Communications Magazine* 51, 9 (2013).
[20] Balachander Krishnamurthy and Craig E. Wills. 2008. Characterizing privacy in online social networks. In *Proc. ACM Workshop on Online Social Networks (WOSN)*. 37–42.
[21] Balachander Krishnamurthy and Craig E. Wills. 2009. On the leakage of personally identifiable information via online social networks. In *Proc. ACM Workshop on Online Social Networks (WOSN)*. 7–12.
[22] Matthew M. Lucas and Nikita Borisov. 2009. flyByNight: mitigating the privacy risks of social networking. In *Proc. Symposium on Usable Privacy and Security (SOUPS)*.
[23] Jianxia Ning, Indrajeet Singh, Harsha V. Madhyastha, Srikanth V. Krishnamurthy, Guohong Cao, and Prasant Mohapatra. 2014. Secret message sharing using online social media. In *Proc. IEEE Conference on Communications and Network Security (CNS)*. 319–327.
[24] Tran Viet Xuan Phuong, Guomin Yang, and Willy Susilo. 2014. Efficient Hidden Vector Encryption with Constant-Size Ciphertext. In *Proc. European Symposium on Research in Computer Security (ESORICS)*. 472–487.
[25] Sean C. Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. 2005. OpenDHT: a public DHT service and its uses. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 73–84.
[26] Mitsunari Shigeo. 2018. a portable and fast pairing-based cryptography library. https://github.com/herumi/mcl.
[27] Kaushal Solanki, Anindya Sarkar, and B. S. Manjunath. 2007. YASS: Yet Another Steganographic Scheme That Resists Blind Steganalysis. In *Proc. International Workshop on Information Hiding (IH)*. 16–31.
[28] Jinyuan Sun, Xiaoyan Zhu, and Yuguang Fang. 2010. A Privacy-Preserving Scheme for Online Social Networks with Efficient Revocation. In *Proc. IEEE International Conference on Computer Communications (INFOCOM)*. 2516–2524.
[29] Rye Terrell. 2012. An easy-to-use encryption system utilizing RSA and AES for javascript. https://github.com/wwwtyro/cryptico.
[30] New York Times. 2018. Zuckerberg, Facing Facebook's Worst Crisis Yet, Pledges Better Privacy. https://www.nytimes.com/2018/03/21/technology/facebook-zuckerberg-data-privacy.html.
[31] Amin Tootoonchian, Stefan Saroiu, Yashar Ganjali, and Alec Wolman. 2009. Lockr: better privacy for social networks. In *Proc. ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*. 169–180.
[32] Evan Vosberg. 2018. JavaScript library of crypto standards. https://github.com/brix/crypto-js.

A CP-HVE CONSTRUCTION

In this work, we utilize the CP-HVE construction by Phuong et al. [24] that is based on bilinear maps on prime order groups. Specifically, given two groups $\mathbb{G}$, $\mathbb{G}_T$ of the same prime order q, a bilinear map $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ satisfies the following properties:

(1) It is *computable*, i.e., given $u, v \in \mathbb{G}$, there is a polynomial time algorithm for computing $e(u, v) \in \mathbb{G}_T$.
(2) It is *bilinear*, i.e., for any $u, v \in \mathbb{G}$ and $a, b \in \mathbb{Z}_q$, $e(u^a, v^b) = e(u, v)^{ab}$.
(3) It is *non-degenerate*, i.e., if g is a generator of $\mathbb{G}$ then $e(g, g)$ is a generator of $\mathbb{G}_T$.

Based on the groups $\mathbb{G}$ and $\mathbb{G}_T$, and the bilinear map e, the four algorithms of the CP-HVE protocol are implemented as shown below. Note that the security of the protocol is based on the Decisional L-Bilinear Diffie-Hellman Exponent (L-BDHE) assumption.

Setup$(1^k, \Sigma, L, N)$: on input a security parameter 1^k, an alphabet Σ, a vector length L, and a maximum number of allowed wildcards N in the encryption vector, the algorithm outputs a public key PK and a master secret key MK.

- Choose random elements $V, H_0, H_1, \ldots, H_{L-1} \in_R \mathbb{G}$.
- Choose random generators $g, w, f \in_R \mathbb{G}$.
- Compute $Y = e(g, w)$.
- The public key is $PK = \langle Y, V, (H_0, \ldots, H_{L-1}), g, f, q, \mathbb{G}, \mathbb{G}_T, e \rangle$.
- The master secret key is $MK = w$.

GenDecKey$(MK, PK, \vec{z})$: on input a master secret key MK, a public key PK, and a decryption vector $\vec{z}$, the algorithm outputs a decryption key DK.

- Choose random elements $r, r_1 \in_R \mathbb{Z}_q$.
- Compute the decryption key DK as:

$$K_1 = g^r, K_2 = g^{r_1}, \begin{pmatrix} K_{3,0} = w(\prod_{i=0}^{L-1}(H_i^{z_i} V)^r) f^{r_1} \\ K_{3,1} = \prod_{i=0}^{L-1}(H_i^{z_i} V)^{(i+1)r} \\ \cdots \\ K_{3,N} = \prod_{i=0}^{L-1}(H_i^{z_i} V)^{(i+1)^N r} \end{pmatrix}$$

Enc$(PK, \vec{x}, \vec{j}, M)$: on input a public key PK, an encryption vector $\vec{x}$, a vector $\vec{j}$ containing the locations of the wildcards in $\vec{x}$, and a message M, the algorithm outputs a ciphertext C.

- Let $\tau \leq N$ be the number of wildcards in $\vec{x}$ that occur at positions $\vec{j} = (j_1, j_2, \ldots, j_\tau)$.
- Using Viete's formulas below, compute $t = a_0$.

$$a_{\tau-k} = (-1)^k \sum_{1 \leq i_1 < i_2 \ldots < i_k \leq \tau} j_{i_1} j_{i_2} \cdots j_{i_k}, \text{ for } 0 \leq k \leq \tau$$

- Choose random $s \in_R \mathbb{Z}_q$.
- Compute $C_0 = MY^s, C_1 = g^{\frac{s}{t}}, C_2 = f^s$, and $C_3 = \prod_{i=0}^{L-1}(VH_i^{x_i})^{\frac{\prod_{k=1}^{\tau}(i+1-j_k)s}{t}}$
- Output ciphertext $C = \langle C_0, C_1, C_2, C_3, \vec{j} \rangle$.

Dec$(DK, \vec{j}, C)$: on input a decryption key DK, a vector $\vec{j}$ containing the locations of the wildcards in the encryption vector, and a ciphertext C, the algorithm outputs a message M.

- Using Viete's formulas above, compute

$$M = \frac{e(K_1, C_3) \cdot e(K_2, C_2)}{e(\prod_{k=0}^{\tau} K_{3,k}^{a_k}, C_1)} \cdot C_0$$