2012 IEEE Symposium on Security and Privacy

Hummingbird: Privacy at the time of Twitter

Emiliano De Cristofaro* — *PARC* — *edc@parc.com*

Claudio Soriente† — *ETH Zurich* — *claudio.soriente@inf.ethz.ch*

Gene Tsudik — *UC Irvine* — *gene.tsudik@uci.edu*

Andrew Williams‡ — *UC Irvine* — *andrewmw@uci.edu*

***Abstract*—In the last several years, micro-blogging Online Social Networks (OSNs), such as Twitter, have taken the world by storm, now boasting over 100 million subscribers. As an unparalleled stage for an enormous audience, they offer fast and reliable centralized diffusion of pithy tweets to great multitudes of information-hungry and always-connected followers. At the same time, this information gathering and dissemination paradigm prompts some important privacy concerns about relationships between tweeters, followers and interests of the latter. In this paper, we assess privacy in today's Twitter-like OSNs and describe an architecture and a trial implementation of a privacy-preserving service called Hummingbird. It is essentially a variant of Twitter that protects tweet contents, hashtags and follower interests from the (potentially) prying eyes of the centralized server. We argue that, although inherently limited by Twitter's mission of scalable information-sharing, this degree of privacy is valuable. We demonstrate, via a working prototype, that Hummingbird's additional costs are tolerably low. We also sketch out some viable enhancements that might offer better privacy in the long term.**

I. Introduction

Online Social Networks (OSNs) offer multitudes of people a means to communicate, share interests, and update others about their current activities. Social networking services are progressively replacing more "traditional" one-to-one communication systems, such as email and instant messaging. Alas, as their proliferation increases, so do privacy concerns with regard to the amount and sensitivity of disseminated information.

Popular OSNs, such as Facebook, Twitter, Google+, provide users with customizable "privacy settings", i.e., users can specify other users (or groups) that can access their content. Information is often classified by categories, e.g., personal, text post, photo or video. For each category, the account owner can define a coarse-grained access control list (ACL). This strategy relies on the trustworthiness of OSN providers and on users appropriately controlling access to their data. Therefore, users need to trust the service not only to enforce their ACLs, but also to store and manage all accumulated content.

OSN providers are generally incentivized to safeguard users' content, since doing otherwise might tarnish their reputation and/or result in legal actions. However, user agreements often include clauses that let providers mine user content, e.g., deliver targeted advertising [36] or re-sell information to third-party services. For instance, Facebook's terms of use classify all user contents as "public" by default, raising privacy concerns in the U.S. Federal Trade Commission [25]. Furthermore, content stored at an OSN provider is subject to potential break-ins [24], insider attacks [37], or subpoenas by law enforcement agencies [30] (e.g., during the WikiLeaks investigation [39]). Moreover, privacy risks are exacerbated by the common practice of caching content and storing it off-line (e.g., on tape backups), even after users explicitly delete it. Thus, the threat to user privacy becomes permanent.

Therefore, it appears that a more effective (or at least an alternative) way of addressing privacy in OSNs is by delegating control over content to its owners, i.e., the end-users. Towards this goal, the security research community has already proposed several approaches [8, 31, 38] that allow users to explicitly authorize "friends" to access their data, while hiding content from the provider and other unauthorized entities.

However, the meaning of *relationship*, or *affinity*, among users differs across OSNs. In some, it is not based on any real-life trust. For example, micro-blogging OSNs, such as Twitter and Tumblr, are based on (short) information exchanges among users who might have no common history, no mutual friends and possibly do not trust each other. In such settings, a user publishes content labeled with some "tags" that help others search and retrieve content of interest.

Furthermore, privacy in micro-blogging OSNs is not limited to content. It also applies to potentially sensitive information that users (subscribers or followers) disclose through searches and interests. Specifically, tags used to label and retrieve content might leak personal habits, political views, or even health conditions. This is particularly worrisome considering that authorities are increasingly monitoring and subpoenaing social network content [20]. We therefore claim that privacy mechanisms for micro-blogging OSNs, such as Twitter, should be designed differently from personal affinity-based OSNs, such as Facebook.

*Work done in part at UC Irvine.

†Work done in part at Universidad Politécnica de Madrid, with funding from Spanish Research Council (MICCIN) under project TIN2010-19077, Madrid Research Foundation (CAM) under project S2009/TIC-1692 (co-funded by ERDF & ESF), Juan de la Cierva Fellowship ref. JCI-2010-06161, and European Commission under project MASSIF (FP7-257475).

‡Supported by NSF Scholarship-for-Service (SFS) grant DUE-0723955.

DOI 10.1109/SP.2012.26

A. Motivation

Twitter is clearly the most popular micro-blogging OSN today. It lets users share short messages (tweets) with their "followers" and enables enhanced content search based on keywords (referred to as *hashtags*) embedded in tweets.

Over time, Twitter has become more than just a popular micro-blogging service. Its pervasiveness makes it a perfect means of reaching large numbers of people through their always-on mobile devices. Twitter is also the primary source of information for untold millions who obtain their news, favorite blog posts or security announcements via 140-character tweets. Twitter is used by entities as varied as individuals, news media outlets, government agencies, NGOs, politicians, political parties/movements as well as commercial organizations of all shapes and sizes. As such, it is an appealing all-around source for information-addicted and attention-deficit-afflicted segment of the population.

Users implicitly trust Twitter to store and manage their content, including: tweets, searches, and interests. Thus, Twitter is in possession of complex and valuable information, such as tweeter-follower relationships and hashtag frequencies. As mentioned above, this prompts privacy concerns. User interests and trends expressed by the "Follow" button represent sensitive information. For example, looking for tweets with a hashtag #TeaParty, (rather than, say, #BeerParty), might expose one's political views. A search for #HIVcure might reveal one's medical condition and could be correlated with the same user's other activity, e.g., repeated appearances (obtained from a geolocation service, such as Google Latitude) of the user's smartphone next to a hospital.

Based on its enormous popularity, Twitter has clearly succeeded in its main goal of providing a ubiquitous real-time push-based information sharing platform. However, we believe that it is time to re-examine whether it is reasonable to trust Twitter to store and manage content (tweets) or search criteria, as well as enforce user ACLs.

B. Contributions

This paper proposes Hummingbird: a privacy-enhanced variant of Twitter. Hummingbird retains key features of Twitter while adding several privacy-sensitive ingredients. Its goal is two-fold:

1) Private fine-grained authorization of followers: a tweeter encrypts a tweet and chooses who can access it, e.g., by defining an ACL based on tweet content.
2) Privacy for followers: they subscribe to arbitrary hashtags without leaking their interests to any entity. That is, Alice can follow all #OccupyWS tweets from the New York Times (NYT) such that neither Twitter nor NYT learns her interests.

Hummingbird can be viewed as a system composed of several cryptographic protocols that allow users to tweet and follow others' tweets with privacy. We acknowledge, from the outset, that privacy features advocated in this paper would affect today's business model of a micro-blogging OSN. Since, in Hummingbird, the provider does not learn tweet contents, current revenue strategies (e.g., targeted advertising) would become difficult to realize. Consequently, it would be both useful and interesting to explore economical incentives of providing privacy-friendly services (not just in the context of micro-blogging OSNs) over the Internet. However, this topic is beyond the scope of this paper.

To demonstrate Hummingbird's practicality (ease of use and performance overhead), we implemented it as a web site on the server side. On the user side, a Firefox extension is employed to access the server, by making cryptographic operations transparent to the user. Hummingbird imposes minimal overhead on users and virtually no extra overhead on the server; the latter simply matches tweets to corresponding followers.

Organization: The rest of this paper is structured as follows: Section II overviews Twitter and a few cryptographic building blocks used to construct Hummingbird. Section III describes our privacy model, while Section IV details Hummingbird architecture and its protocols. Prototype implementation is described in Section VI. Next, Section VII discusses privacy features of Hummingbird and considers several extensions. Finally, Section VIII surveys related work and the paper concludes in Section IX.

II. Preliminaries

This section provides Twitter background and describes some cryptographic building blocks.

A. Twitter

As the most popular micro-blogging OSN, Twitter (http://www.twitter.com) boasts 100 million active users worldwide, including: reporters, artists, public figures, government agencies, NGOs and commercial entities [46]. Its users communicate via 140-character messages, called *tweets*, using a simple web interface. Posting messages is called *tweeting*. Users may subscribe to other users' tweets; this practice is known as *following*. Basic Twitter terminology includes:

- A user who posts a tweet is a *tweeter*.
- A user who follows others' tweets is a *follower*.
- The centralized entity that maintains profiles and matches tweets to followers is simply *Twitter*.

Tweets are labeled and retrieved (searched) using ***hashtags***, i.e., strings prefixed by a "#" sign. For example, a tweet: "I don't care about #privacy on #Twitter" would match any search for hashtags "#privacy" or "#Twitter". An "@" followed by a user-name is utilized for mentioning, or replying to, other users. Finally, a tweet can be re-published by other users, and shared with one's own followers, via the so-called *re-tweet* feature.

Tweets are *public* by default: any registered user can see (and search) others' public tweets. These are also indexed by third party services – such as Google – and can be accessed by application developers through a dedicated streaming API. All public tweets are also posted on a public website (http://twitter.com/public_timeline), that keeps the tweeting "timeline" and shows twenty most recent messages.

Tweeters can restrict availability of their tweets by making them "private" – accessible only to authorized followers [4]. Tweeters can also revoke such authorizations, using (and trusting) Twitter's *block* feature. Nonetheless, whether a tweet is public or private, Twitter collects all of them and forwards them to intended recipients. Thus, Twitter has access to all information within the system, including: tweets, hashtags, searches, and relationships between tweeters and their followers. Although this practice facilitates dissemination, availability, and mining of tweets, it also intensifies privacy concerns stemming from exposure of information.

B. Cryptography Background

We now overview some cryptographic concepts and tools used in the rest of the paper. For ease of exposition, we omit basic notions and refer to [27, 42] for details on various cryptographic primitives, such as hash functions, number-theoretic assumptions as well as encryption and signature schemes.

Basic Notation. A function $\mathcal{F}(\tau)$ is *negligible* in the security parameter τ if, for every polynomial p, $\mathcal{F}(\tau) < 1/|p(t)|$ for large enough t. Throughout the paper, we use semantically secure symmetric encryption and assume the key space to be τ_1-bit strings, where τ_1 is a (polynomial) function of the security parameter τ. $Enc_k(\cdot)$ and $Dec_k(\cdot)$ denote symmetric-key encryption and decryption under key k, respectively. Finally, $a \leftarrow_R A$ means that variable a is chosen uniformly, at random from set A.

Oblivious PseudoRandom Functions (OPRFs). Informally, a pseudorandom function (PRF) family is a collection of efficiently computable functions such that no efficient algorithm, under some computational assumption, can distinguish, with non-negligible advantage, between a function chosen randomly from this family and one with truly random outputs. A PRF f is a function that takes two inputs: a variable x and a secret function index s, and outputs $f_s(x)$. An Oblivious PRF (OPRF) is a two-party protocol, between sender and receiver, that securely computes $f_s(x)$ on secret index s contributed by sender and input x – by the receiver, such that the former learns nothing from the interaction, and the latter only learns $f_s(x)$. OPRFs have been introduced by Freedman, et al. [26], based on Naor-Reingold PRF [44]. Several OPRF constructions have been suggested since, e.g., [9, 34] based on Boneh-Boyen PRF [12].

Blind RSA Signatures. A blind signature scheme allows one to sign a message such that its content is disguised (blinded) before being signed. The resulting signature can be publicly verified against the original blinded message. Knowledge of the blinding factor allows one to *unblind* a blind signature and obtain a (*message, signature*) pair that cannot be correlated to its original (blinded) counterpart. There have been many interesting research results in the context of blind signatures involving various constructions, security models, assumptions, and computational settings. (See [47] for more details.) In this paper, we focus on RSA Blind Signatures introduced in [16]. Blind RSA Signature Scheme (Blind-RSA) involves a signer (S), a receiver, (R) and the following algorithms:

- Key-Gen(1^τ): On input of the security parameter τ, S generates a safe RSA modulus $N = pq$, where p and q are random distinct τ_2-bit primes, (with τ_2 as a polynomial function of τ), such that $p = 2p' + 1$ and $q = 2q' + 1$ for distinct primes p', q'. Next, a random positive integer $e < \phi(N)$ is selected such that $\gcd(e, \phi(N)) = 1$. Also, d is generated such that $ed = 1 \bmod \phi(N)$. Finally, a Full Domain Hash (FDH) function $H : \{0,1\}^* \rightarrow \mathbb{Z}_N$ is selected. The output consists of the RSA public/private keypair $((N, e), d)$ as well as $H(\cdot)$.
- Blind-Sign(d, x): Given public input $(H(\cdot), N, e)$, $\mathcal{S}$ and $\mathcal{R}$ interact on private input d and x, respectively. The protocol is as follows: $\mathcal{R}$ sends $\mu = H(x) \cdot r^e \bmod N$ (for $r \leftarrow_R \mathbb{Z}_N$) to $\mathcal{S}$, that sends back $\mu' = \mu^d \bmod N$. Finally, $\mathcal{R}$ obtains $\sigma = \mu'/r \bmod N$. It is easy to see that σ is a valid RSA signature on message x under private key d: $\sigma = \mu^d/r = H(x)^d r^{ed} r^{-1} = H(x)^d \bmod N$.
- Verify(σ, x): Signature σ is publicly verified, by checking that $\sigma^e = H(x) \bmod N$.

Blind-RSA based OPRFs. Blind RSA signatures can be used, in the Random Oracle Model (ROM), to realize an OPRF. The actual function is defined as $f_d(x) = H'(H(x)^d)$, where $H(\cdot)$ and $H'(\cdot)$ are modeled as random oracles. The OPRF protocol is simply the Blind-RSA protocol presented above, with d contributed by sender, and x – by receiver. Using this protocol, the function remains a PRF under the *One-More-RSA* assumption [10], even if receiver is malicious, as recently shown in [18].

Blind-DH based OPRFs. In ROM, OPRFs can also be instantiated using a Blind Diffie-Hellman protocol. This construction, presented in [35], relies on the function $f_s(x) = H'(H(x)^s)$ where $H(x)$ maps onto a group where the Computational Diffie-Hellman problem is assumed to be hard, and both $H(\cdot), H'(\cdot)$ are modeled as random oracles. The protocol is similar to Blind-RSA, but is secure under the *One-More-DH* assumption [10]. It runs on public input of two primes p, q (with $q|p-1$). Receiver blinds its input by sending $\mu = H(x)^r \bmod p$, with $r \leftarrow_R \mathbb{Z}_q$. Sender replies with $\mu' = \mu^s \bmod p$ and receiver obtains $f_s(x) = H'(H(x)^s)$ by computing $H'((\mu')^{1/r} \bmod p)$.

Blind-DH vs Blind-RSA. There are a few differences between Blind-DH and Blind-RSA OPRFs. First, sender's computation in Blind-RSA is verifiable, as opposed to Blind-DH. In the former, receiver can verify correctness of sender's computation by checking $H(x) = (\mu'/r)^e \bmod N$, assuming that N is generated correctly. Whereas, to achieve the same in Blind-DH, sender needs to attach a Zero-Knowledge Proof of Knowledge (ZKPK) of the discrete log. Also, usage of Chinese Remainder Theorem (CRT) in signing yields lower computational complexity for Blind-RSA.

III. Defining Privacy in Micro-Blogging OSNs

Defining privacy in a Twitter-like system is a challenging task. Our definition revolves around the server (i.e., Twitter itself) that needs to match tweets to followers while learning as little as possible about both. This would be trivial if tweeters and followers shared secrets [52]. It becomes more difficult when they have no common secrets and do not trust each other.

A. Built-in Limitations

From the outset, we acknowledge that privacy attainable in Twitter-like systems is far from perfect. Ideal privacy in micro-blogging OSNs can be achieved only if no central server exists: all followers receive all tweets and decide, in real-time, which are of interest. Clearly, this would be unscalable and impractical in many respects. Thus, a third-party server becomes necessary.

The main reason for the server's existence is the matching function: it matches incoming tweets to subscriptions and forwards them to corresponding followers. Although we want the server to learn no more information than an adversary observing a secure channel, the very same matching function precludes it.

Similarly, the server learns whenever multiple subscriptions match the same hashtag in a tweet. Preventing this is not practical, considering that a a tweeter's single hashtag might have a very large number of followers. It appears that the only way to conceal the fact that multiple followers are interested in the same hashtag (and tweet) is for the tweeter to generate a distinct encrypted (*tweet, hashtag*) pair for each follower. This would result in a linear expansion (in the number of followers of a hashtag) for each tweet. Also, considering that all such pairs would have to be uploaded at roughly the same time, even this unscalable approach would still let the server learn that – with high probability – the same tweet is of interest to a particular set of followers.

Note that the above is distinct from server's ability to learn whether a given subscription matches the same hashtag in multiple tweets. As we discuss later in the paper (Section VII-C), one could prevent the server from learning this information, while incurring reasonable extra overhead. However, it remains somewhat unclear whether the privacy gain would be worthwhile.

B. Privacy Goals and Security Assumptions

Our privacy goals are commensurate with aforementioned limitations.

- Server: learns minimal information beyond that obtained from performing the matching function. We allow it to learn which, and how many, subscriptions match a hashtag (even if the hashtag is cryptographically transformed). Also, it learns whether two subscriptions for the same tweeter refer to the same hashtag. Furthermore, it learns whenever two tweets by the same tweeter carry the same hashtag. However, as mentioned above, this can be easily remedied; see Section VII-C.
- Tweeter: learns who subscribes to its hashtags but not *which* hashtags have been subscribed to.
- Follower: learns nothing beyond its own subscriptions. Specifically, it learns no information about any other subscribers or any tweets that do not match its subscriptions.

Our privacy goals, coupled with the key desired features of a Twitter-like system, prompt an important assumption that the server must adhere to the **Honest-but-Curious (HbC)** adversarial model. Specifically, although the server faithfully follows all protocol specifications, it may attempt to passively violate our privacy goals. According to our interpretation, the HbC model precludes the server from creating "phantom" users. In other words, the server does not create spurious accounts in order to obtain subscriptions and test whether they match other followers' interests. The justification for this assertion is as follows:

> Suppose that the server creates a phantom user for the purpose of violating privacy of genuine followers. The act of creation itself does not violate the HbC model. However, when a phantom user engages a genuine tweeter in order to obtain a subscription, a protocol transcript results. This transcript testifies to the existence of a spurious user (since the tweeter can keep a copy) and can be later used to demonstrate server misbehavior.

We view this assumption as unavoidable in any Twitter-like OSN. The server provides the most central and the most valuable service to large numbers of users. It thus has a valuable reputation to maintain and any evidence, or even suspicion, of active misbehavior (i.e., anything beyond Honest-but-Curious conduct) would result in a significant loss of trust and a mass exodus of users.

Finally, we emphasize that side-channels (e.g., timing and correlation) attacks are beyond the scope of this paper.

C. Definitions

To quantify privacy loss at the server we define a generic function $\zeta(\cdot,\cdot)$ over an arbitrary set of tweets and subscriptions. We then model the probability that the server, given access to cryptographically transformed tweets and subscriptions, can compute such a function vis-a-vis probability of computing $\zeta(\cdot,\cdot)$ in an ideal world where it interacts with an oracle doing the matching. In particular, let a tweet T be a triple (id^*, M, ht^*) where id is the unique ID of the tweeter, M is the message and ht is the hashtag included in the tweet. (In real-world Twitter, a message can have multiple hashtags; however, we assume a single hashtag in order to simplify the discussion, with no loss of generality). Let a subscription R be a pair (id, ht) where id is the unique ID of the tweeter and ht is the hashtag the tweet should contain in order for a match to occur.

We define the following functions:

$$\begin{aligned} match(T,R) = 1 &\Leftrightarrow T.id^* = R.id \wedge T.ht^* = R.ht \\ match_{tw}(T,T') = 1 &\Leftrightarrow T.id^* = T'.id^* \wedge T.ht^* = T'.ht^* \\ match_{req}(R,R') = 1 &\Leftrightarrow R.id = R'.id \wedge R.ht = R'.ht \end{aligned}$$

Let $\mathcal{T} = \{T_1, \ldots, T_n\}$ and $\mathcal{R} = \{R_1, \ldots, R_m\}$ be the set of tweets and subscriptions up to a given time and let $\hat{\mathcal{T}}, \hat{\mathcal{R}}$ be their cryptographically transformed versions, respectively. Let $\mathcal{O}$ be an oracle that has access to $\mathcal{T}, \mathcal{R}$, and implements functions $match$, $match_{tw}$ and $match_{req}$.

Informally, we want the view of the server (that has access to $\hat{\mathcal{T}}, \hat{\mathcal{R}}$) to be the same as that of a server that asks $\mathcal{O}$ to implement aforementioned functions over $\mathcal{T}$ and $\mathcal{R}$. That is, the following probability:

$$|Pr[\zeta(\mathcal{T},\mathcal{R}) \leftarrow A^{\hat{\mathcal{T}},\hat{\mathcal{R}}}] - Pr[\zeta(\mathcal{T},\mathcal{R}) \leftarrow A^{\mathcal{O}(\mathcal{T},\mathcal{R})}]|$$

should be negligible, for any $\zeta(\cdot,\cdot)$ defined over $\mathcal{T}, \mathcal{R}$.

We now provide some definitions that try to capture the privacy loss that Hummingbird must bear in order to efficiently match tweets to subscriptions. Within a tweet, we distinguish between the message (i.e., conveyed information) and its hashtags (i.e., the keywords that the system uses to identify messages).

Tweeter Privacy. An encrypted tweet that includes a hashtag ht should leak no information to any party that has not been authorized by the tweeter to follow it on ht. In other words, only users that have been authorized to follow the tweeter on the hashtag can decrypt the associated message. For its part, the server learns whenever multiple tweets from a given tweeter contain the same hashtag.

Follower Privacy. A request to follow a tweeter on hashtag ht should disclose no information about the hashtag to any party other than the follower. That is, a follower can subscribe to hashtags such that tweeters, the server or any other party learns nothing about follower interests. However, the server learns whenever multiple followers are subscribed to the same hashtag of a given tweeter.

Matching Privacy. The server can compute only functions that can be also computed by interacting with the oracle that implements the match functions.

IV. Private Tweeting in Hummingbird

In this section, we present the Hummingbird architecture and protocols.

A. Architecture

Hummingbird architecture mirrors Twitter's, involving one central server and an arbitrary number of registered users, that can publish and retrieve short text-based messages. Publication and retrieval is based on a set of hashtags (i.e., arbitrary keywords) that are appended to the message or specified in the search criteria. Similar to Twitter, Hummingbird involves three types of entities:

1) ***Tweeters*** post messages accompanied by a set of hashtags that are used by other users to search for those messages. For example, Bob posts a message: "I care about #privacy" where "#privacy" is the associated hashtag.
2) ***Followers*** issue "follow requests" to any tweeter for any hashtag of interest, and, if a request is approved, receive all tweets that match their interest. For instance, Alice who wants to follow Bob's tweets with hashtag "#privacy" would receive the tweet: "I care about #privacy" and all other Bob's tweets that contain the same hashtag.
3) ***Hummingbird Server*** (HS) handles user registration and operates the Hummingbird web site. It is responsible for matching tweets with follow requests and delivering tweets of interest to users.

B. Design Overview

In contrast to Twitter, access to tweets in Hummingbird is restricted to authorized followers, i.e., they are hidden from HS and all non-followers. Also, all follow requests should be subject to approval. Whereas, in Twitter, users can decide to approve all requests automatically. In addition, Hummingbird introduces the concept of ***follow-by-topic***, i.e., followers decide to follow tweeters and specify hashtags of interest. This feature is particularly geared for following high-volume tweeters, as it filters out "background noise" and avoids saddling users with potentially large quantities of unwanted content. For example, a user might decide to follow the New York Times on #politics, thus, not receiving NYT's tweets on, e.g., #cooking, #gossip, etc. Furthermore, follow-by-topic might allow tweeters to charge followers a subscription fee, in order to access premium content. For example, Financial Times could post tweets about stock market trends with hashtag #stockMarket and

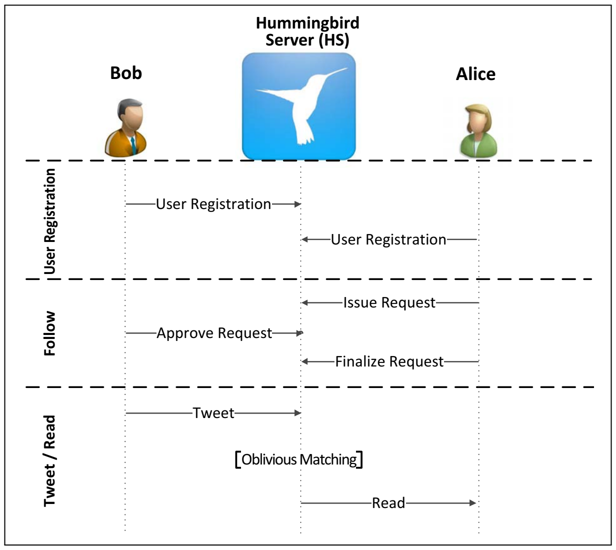

Figure 1: Hummingbird protocols overview.

only authorized followers who pay a subscription fee would receive them.

Key design elements are as follows:

1) Tweeters encrypt their tweets and hashtags.
2) Followers can *privately* follow tweeters on one or more hashtags.
3) HS can *obliviously* match tweets to follow requests.
4) Only authorized (previously subscribed) followers can decrypt tweets of interest.

At the same time, we need to minimize overhead at HS. Ideally, privacy-preserving matching should be as scalable as its non-private counterpart.

Intuition. At the core of Hummingbird architecture is a simple OPRF-based technique. Suppose Bob wants to tweet a message M with a hashtag ht. The idea is to derive an encryption key for a semantically secure cipher (e.g., AES) from $f_s(ht)$ and use it to encrypt M. (Recall that s is Bob's secret.) That is, Bob computes $k = H_1(f_s(ht))$, encrypts $Enc_k(M)$ and sends it to HS. Here, $H_1 : \{0,1\}^* \rightarrow \{0,1\}^{\tau_1}$ is a cryptographic hash function modeled as a random oracle.

To follow Bob's tweets with hashtag ht, another user (Alice) must first engage Bob in an OPRF protocol where she plays the role of the receiver, on input ht, and Bob is the sender. As a result, Alice obtains $f_s(ht)$ and derives k that allows her to decrypt all Bob's tweets containing ht. Based on OPRF security properties, besides guaranteeing tweets' confidentiality, this protocol also prevents Bob from learning Alice's interests, i.e., he only learns that Alice is among his followers but does not learn which hashtags are of interest to her. As described in Section IV-C (protocol **Follow**) below, Alice and Bob do not run the OPRF protocol directly or in real time. Instead, they use HS as a conduit for OPRF protocol messages.

Once Alice establishes a follower relationship with Bob, HS must also efficiently and *obliviously* match Bob's tweets to Alice's interests. For this reason, we need a secure tweet labeling mechanism.

To label a tweet, Bob uses a PRF, on input an arbitrary hashtag ht, to compute a cryptographic token t, i.e., $t = H_2(f_s(ht))$ where H_2 is another cryptographic hash function, modeled as a random oracle: $H_2 : \{0,1\}^* \rightarrow \{0,1\}^{\tau_3}$, with τ_3 polynomial function of the security parameter τ. This token is communicated to HS along with the encrypted tweet.

As discussed above, on the follower side, Alice must obtain $f_s(ht)$ beforehand, as a result of an OPRF protocol with Bob. She then computes the same token t, and uploads it to HS. Due to OPRF security properties, t reveals no information about the corresponding hashtag. HS only learns that Alice is one of Bob's followers. From this point on, HS obviously and efficiently matches Bob's tweets to Alice's interests. Upon receiving an encrypted tweet and an accompanying token from Bob, HS searches for the latter among all tokens previously deposited by Bob's followers. As a result, HS only learns that a tweet by Bob matches a follow request by Alice.

OPRF choice. Although Hummingbird does not restrict the underlying OPRF instantiation, we selected the OPRF construct based on Blind-RSA signatures (in ROM) since it offers lowest computation and communication complexities. One side-benefit of using the Blind-RSA-based OPRF is that it allows us to use standard RSA public key certificates. At the same time, the Hummingbird architecture is not dependent on Blind-RSA based OPRF; it can be seamlessly replaced with any other OPRF construction, e.g., see Section VII-H.

C. Protocols

Figure 1 overviews protocols involved in a simple scenario with only two users (tweeter Bob and follower Alice). The actual protocols are described below.

User Registration. To join Hummingbird, a user registers, and creates an account, with HS. This involves obtaining username/password credentials and creating a public Hummingbird profile. Also, at this phase, each user creates an RSA keypair, e.g., $[(N_b, e_b), (d_b)]$ for Bob. The public key becomes part of the user's profile.

Follow. This is a three-step protocol that authorizes a user (Alice) to follow another user (Bob). For simplicity's sake, we assume that Alice is interested only in one hashtag ht and later describe how to support multiple hashtags. The protocol is shown in Figure 2. As mentioned above, it builds on Blind-RSA based OPRFs and includes the following three steps:

1) ***Issue Request:*** Alice sends Bob a request to follow his tweets on an arbitrary hashtag ht. This corresponds to

the first round of Blind-Sign protocol in Section II-B. The request is routed through HS; thus, Bob does not need to be on-line.

2) ***Approve Request:*** Whenever Bob logs in to Hummingbird, HS prompts him with Alice's request. If Bob approves the request, he blindly signs it and the resulting signature is stored by HS for later delivery to Alice. (Note that Alice might not be on-line at this time.) This corresponds to the second round of the Blind-Sign protocol in Section II-B.
3) ***Finalize Request:*** Upon next login, Alice receives Bob's approved request. She then finalizes it as the last round of Blind-Sign protocol (i.e., by unblinding Bob's signature) and computing the outer hash to obtain the final OPRF value, i.e., a token, corresponding to Bob's hashtag ht. Finally, Alice deposits the resulting token with HS.

After successful completion of the above protocol, Alice is authorized to obtain all Bob's tweets (labeled with hashtag ht). Our present architecture does not handle revocation of follow requests: although it is not difficult to incorporate it, there does not seem to be an elegant way to do it without relying on HS. To withdraw Alice's authorization, Bob could either generate a new set of RSA parameters, or replace hash function $H(\cdot)$. Unfortunately, either way, all outstanding subscriptions would have to be renewed, i.e., Bob needs to re-run the Follow protocol with all non-revoked followers. This is possible since Bob knows their identities.

Tweet/Read. We now discuss how to privately send/read tweets in Hummingbird. For now, we assume that Bob attaches only a single hashtag to his tweet, and later show how to support multiple hashtags. The protocol is illustrated in Figure 3.

1) ***Tweet:*** Assume that Bob tweets a message M, associated with hashtag ht^*. He derives an encryption key from a PRF evaluation of ht^*, encrypts M, and uploads it to HS. PRF evaluation of ht^* is also used by Bob to compute a cryptographic token that is attached to the encrypted message. This token is later used by HS to match tweets with followers. Once again, we use the OPRF construction based on Blind-RSA signatures.
2) ***Oblivious Matching:*** Suppose Alice follows Bob on ht and $ht = ht^*$. This means that she has deposited at HS, a cryptographic token matching the one uploaded by Bob upon tweeting (derived by the OPRF execution as per protocol in Figure 2). As a result, HS adds Bob's encrypted tweet to Alice's profile.
3) ***Read:*** Upon login, Alice receives Bob's encrypted tweet; she reconstructs the decryption key (again, derived by the OPRF execution) and reads the tweet upon decryption.

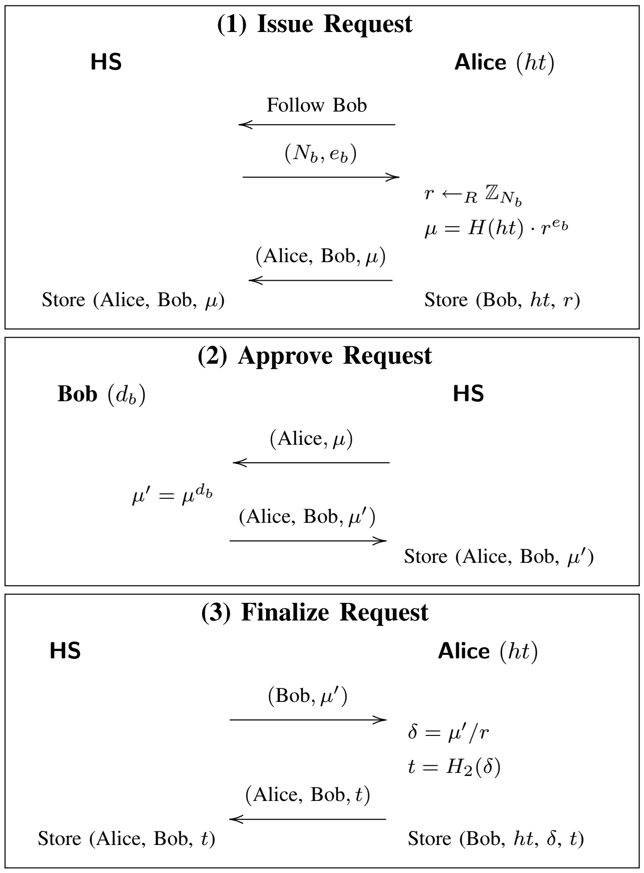

Figure 2: The three-step protocol corresponding to *follow* a user in Hummingbird. It executes on common input H, H_2. All computation is assumed mod N_b.

V. Security

In this section, we discuss security properties of Hummingbird.

Tweeters/Followers Privacy. We start by noting that the OPRF-based protocol implementing *following-by-topic* hides the content of hashtags requested by Alice from Bob, HS, or any eavesdropper. Similarly, Bob's tweets (and hashtags) reveal no information about tweet content to any party but authorized followers. Indeed, this follows directly from the security of our OPRF instantiation, based on Blind-RSA signatures, under the One-More-RSA assumption [10].

Matching Privacy. Next, we consider security of oblivious matching, as performed by HS. To do so, we need to show that HS matches tweets to followers while learning nothing about cleartext tweets and hashtags. However, as discussed in Section III, we do not attempt to prevent HS from learning tweeter/follower relations, distributions of (encrypted) tweets and hashtags, as well learning whether multiple follow requests are based on the same hashtag. (Though we sketch out a way to avoid the latter in Section

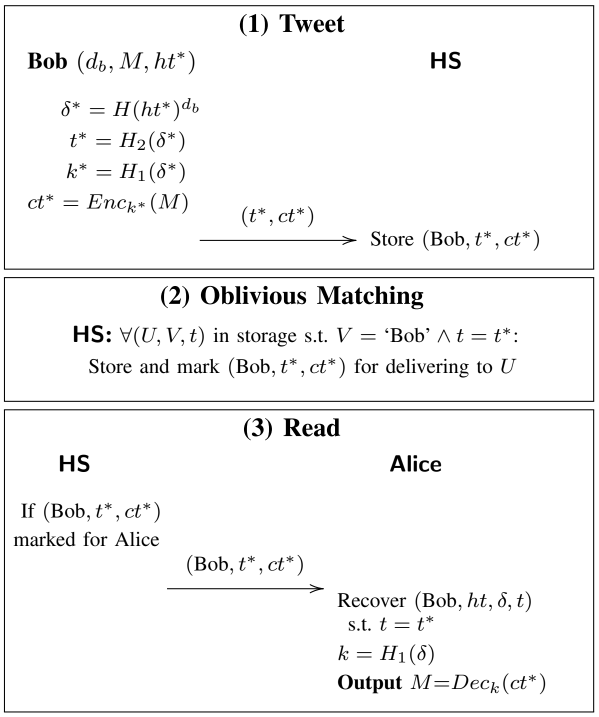
(1) Tweet
Bob (d_b, M, ht^*)
HS
$\delta^* = H(ht^*)^{d_b}$
$t^* = H_2(\delta^*)$
$k^* = H_1(\delta^*)$
$ct^* = Enc_{k^*}(M)$
(t^*, ct^*)
Store (Bob, t^*, ct^*)
(2) Oblivious Matching
HS: $\forall (U, V, t)$ in storage s.t. $V =$ 'Bob' $\wedge\, t = t^*$:
Store and mark (Bob, t^*, ct^*) for delivering to U
(3) Read
HS
Alice
If (Bob, t^*, ct^*) marked for Alice
(Bob, t^*, ct^*)
Recover (Bob, ht, δ, t) s.t. $t = t^*$
$k = H_1(\delta)$
Output $M = Dec_k(ct^*)$

Figure 3: Tweeting and reading in Hummingbird. It executes on common input H, H_1, H_2. All computation is mod N_b.

VII-C.)

Let $\mathcal{T} = \{m_1, ht_1^*, \ldots, m_n, ht_n^*\}$ be a set of tweets and $\mathcal{R} = \{ht_1, \ldots, ht_l\}$ a set of follow requests.[1] Then, we define the corresponding sets that HS receives as $\hat{\mathcal{T}} = \{ct_1^*, t_1^* \ldots, ct_n^*, t_n^*\}$ and $\hat{\mathcal{R}} = \{t_1, \ldots, t_l\}$. We want to demonstrate that, with overwhelming probability, any function $\zeta(\mathcal{T}, \mathcal{R})$ computable by an adversary A on input $(\hat{\mathcal{T}}, \hat{\mathcal{R}})$, can be also computed by a PPT algorithm A^* that has only access to an oracle matching tweets and requests. Specifically, we need to show that:

$$|Pr[\zeta(\mathcal{T}, \mathcal{R}) \leftarrow A^{(\hat{\mathcal{T}}, \hat{\mathcal{R}})}] - Pr[\zeta(\mathcal{T}, \mathcal{R}) \leftarrow A^{\mathcal{O}(\mathcal{T}, \mathcal{R})}]|$$

is negligible in the security parameter, with $\mathcal{O}$ implementing the following functions:

$$match(i, j) = 1 \Leftrightarrow \exists\, ht_i^* \in \mathcal{T}, ht_j \in \mathcal{R} : ht_i^* = ht_j$$
$$match_{tw}(i, j) = 1 \Leftrightarrow \exists\, ht_i^* \in \mathcal{T}, ht_j^* \in \mathcal{T} : ht_i^* = ht_j^* \wedge i \neq j$$
$$match_{req}(i, j) = 1 \Leftrightarrow \exists\, ht_i \in \mathcal{R}, ht_j \in \mathcal{R} : ht_i = ht_j \wedge i \neq j$$

If this occurs, then there exists a PPT algorithm A^* that can construct $\bar{\mathcal{T}} = \{\gamma_1^*, \upsilon_1^*, \ldots, \gamma_n^*, \upsilon_n^*\}, \bar{\mathcal{R}} = \{\upsilon_1, \ldots, \upsilon_l\}$, indistinguishable from $\hat{\mathcal{T}}, \hat{\mathcal{R}}$. Thus, A^* can simulate A with input $\bar{\mathcal{T}}, \bar{\mathcal{R}}$ to obtain the same function output. A^* builds $\bar{\mathcal{T}}, \bar{\mathcal{R}}$ as follows:

[1]We assume, w.l.o.g., that there is only one tweeter, thus, removing identities from the sets of tweets and follow requests.

- Assume there are no requests, $\bar{\mathcal{R}} = \emptyset$. The i-th tweet/token pair γ_i^*, υ_i^* of $\bar{\mathcal{T}}$ is set as follows:
  $\gamma_i^* \leftarrow_R \mathbb{Z}_{N_b}$
  $$\upsilon_i^* : \begin{cases} \upsilon_i^* = \upsilon_j^* \text{ if } \exists\, j : \mathcal{O}.match_{tw}(j, i) = 1 \wedge j < i \\ \upsilon_i^* \leftarrow_R \{0,1\}^{\tau_3} \text{ otherwise} \end{cases}$$
- Assume there are no tweets, thus, $\bar{\mathcal{T}} = \emptyset$. The i-th token υ_i of $\bar{\mathcal{R}}$ is set as:
  $$\upsilon_i : \begin{cases} \upsilon_i = \upsilon_j \text{ if } \exists\, j : \mathcal{O}.match_{req}(j, i) = 1 \wedge j < i \\ \upsilon_i \leftarrow_R \{0,1\}^{\tau_3} \text{ otherwise} \end{cases}$$
- Assume there are both tweets and follow requests, i.e., $\bar{\mathcal{T}} \neq \emptyset, \bar{\mathcal{T}} \neq \emptyset$. The i-th tweet/token pair γ_i^*, υ_i^* of $\bar{\mathcal{T}}$ is set as in the first bullet. The i-th token υ_i of $\bar{\mathcal{R}}$ is set as:
  $$\upsilon_i : \begin{cases} \upsilon_i = \upsilon_j \text{ if } \exists\, j : \mathcal{O}.match(j, i) = 1 \wedge j < i \\ \upsilon_i = \upsilon_j \text{ if } \exists\, j : \mathcal{O}.match_{req}(j, i) = 1 \wedge j < i \\ \upsilon_i \leftarrow_R \{0,1\}^{\tau_3} \text{ otherwise} \end{cases}$$

In all above cases, $(\bar{\mathcal{T}}, \bar{\mathcal{R}})$ is indistinguishable from $(\hat{\mathcal{T}}, \hat{\mathcal{R}})$. If not, then we could use A, A^* to distinguish random strings from pseudo-random ones.

VI. System Prototype

We implemented Hummingbird as a working research prototype. It is available at http://sprout.ics.uci.edu/hummingbird.

In this section, we demonstrate that: (1) by using efficient cryptographic mechanisms, Hummingbird offers a privacy-preserving Twitter-like messaging service, (2) the resulting implementation introduces no overhead on the central service (HS) (thus raising no scalability concerns), and (3) performance of Hummingbird are suitable to real-world deployment.

A. Server-side

In the description of the implementation, we distinguish between server- and client-side components, as shown in Figure 4. Hummingbird's server-side component corresponds to HS, introduced in Section IV. It consists of three parts: (1) database, (2) JSP classes, and (3) Java back-end. We describe them below.

Database. Hummingbird employs a central database to store and access user accounts, encrypted tweets, follow requests, and profiles.

JSP Front-end. The visual component of Hummingbird is realized through JSP pages. They allow users to seamlessly interact with a back-end engine, via the web browser. Main web functionalities include: registration, login, issuing/accepting/finalizing a request to follow, tweeting, reading streaming tweets, and accessing user profiles.

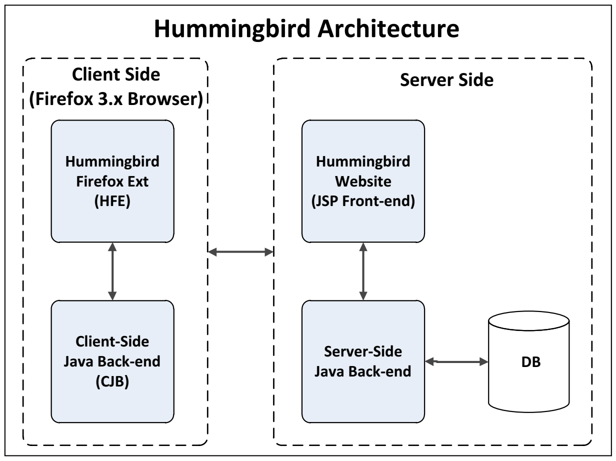
Hummingbird Architecture
Client Side
(Firefox 3.x Browser)
Hummingbird Firefox Ext (HFE)
Client-Side Java Back-end (CJB)
Server Side
Hummingbird Website (JSP Front-end)
Server-Side Java Back-end
DB

Figure 4: Server-side and Client-side components of the Hummingbird prototype.

Java Back-end. Hummingbird functionality is realized by a Java back-end running on HS. The back-end is deployed in Apache Tomcat. The software includes many modules and we omit their detailed description. The back-end mainly handles access to the database, populates web pages, and performs efficient matching of tweets to followers using off-the-shelf database querying mechanisms.

B. Client-side

Users interface with the system via the Hummingbird web site. We implement each of the operations in Hummingbird as a web transaction, and users perform them from their web browser. However, several client-side cryptographic operations need to be performed outside the browser: to the best of our knowledge, there is no straightforward browser support for complex public-key operations such as those needed in OPRF computation.

To this end, we introduce, on the client-side, a small Java back-end, used to perform cryptographic operations. Then, we design a Firefox extension (HFE) to store users' keys and to *automatically* invoke appropriate Java code for each corresponding action. Its latest version is compatible with Firefox 3.x.x and is available from http://sprout.ics.uci.edu/hummingbird.

Client-side Java Back-end (CJB). Hummingbird users are responsible for generating their RSA keys, encrypt/decrypt tweets according to the technique presented in Section IV, and perform OPRF computations during follow request/approval. In our system, these cryptographic operations are implemented by a small Java back-end, CJB, included in the HFE presented below. CJB relies on the Java Bouncy Castle Crypto library.

Hummingbird Firefox Extension (HFE). As mentioned above, HFE interfaces the web browser to the client-side Java back-end, that is included as part of the extension package. The extension code connects to it using Java Live-Connect [43]. Once installed, HFE is completely transparent to the user. HFE is used for:

Key management. During user registration, HFE automatically invokes RSA key generation code from CJB, stores (and optionally password-protects) public/private key in the extension folder, and lets browser report public key to HS.

Following. For each of the three steps involved in requesting to follow a tweeter, the user is guided by Hummingbird web site, however, CJB code needs to be executed to realize corresponding cryptographic operations. This is done, automatically, by HFE.

Tweet. When a user tweets on Hummingbird, HFE—transparently to her—intercepts message and hashtag and invokes the CJB code to encrypt the message and generate the appropriate cryptographic token.

Read. Followers receive, from HS, tweets matching their interest, however, these are encrypted (recall that matching is performed obliviously at HS). Nevertheless, HFE automatically decrypts them (using CJB code) and replace web page content with corresponding cleartext.

C. Performance

Following its architecture, the Hummingbird prototype guarantees privacy of tweeters and followers at minimal costs for both and at virtually no cost for the server. In particular, cryptographic overhead incurred by tweeters and followers is negligible and arguably not perceivable by end-users. Table I summarizes the overhead incurred by each operation in Hummingbird. On the user side, experiments were conduced on a 2011 Macbook Pro with a 2.3 GHz Intel Core i5 CPU. Whereas, HS was running on an Intel Harpertown platform with a 2.5GHz Xeon CPU (HS). Performance analysis is discussed below.

Follow. Following a user requires executing the three-step protocol of Figure 2. (Once again, since messages are routed through HS, this protocol is asynchronous and does not require users to be simultaneously online.) It requires an OPRF invocation, thus, two modular multiplications, one (short) exponentiation, two hash evaluations, one RSA signature and one modular inversion in the RSA setting. Therefore, computational overhead is clearly dominated by tweeter's computation of one RSA signature per each requested hashtag. According to our experiments, an RSA signature with a 1024-bit modulus takes less than $1ms$ using Java and Chinese Remainder Theorem, while modular multiplications take less than $0.01ms$. Thus, we conclude that the user-side cryptographic overhead stemming from the *Follow* protocol is dominated by the time to perform the web transaction itself. Total communication overhead amounts to 2 integers in the RSA group for each hashtag of interest.

Tweeting. This operation requires one hash and one RSA signature for each hashtag associated with the tweet and the

Operation	Computation Overhead		Communication Overhead
	HS	User	
Request to Follow	None	1 exp, 1 mult, 1 hash	1024 bits
Approve Request	None	1 exp	1024 bits
Finalize Request	None	1 mult, 1 inv, 1 hash	1024 bits
Tweet	None	1 exp, 1 hash 1 AES enc	160 bits
Read	None	1 AES dec	160 bits
Match	O(log(n))	–	–

Table I: Overhead of all Hummingbird operations *per-hashtag*.
[Notation: n denotes the total number of *follow requests*; AES enc/dec' use 128-bit symmetric keys; 'hash' designates SHA-1 invocations; 'exp', 'mult', and 'inv' denote, resp., modular exponentiations, multiplications, and inverses of 1024-bit integers.]

computation of one symmetric key encryption (e.g., AES).[2] Hence, computation overhead is once again negligible compared to that required to complete the web transaction. Communication overhead only amounts to one output of a cryptographic hash function (e.g., SHA1) for each hashtag. Also, reading a tweet only requires one AES decryption, which can be considered negligible.

Server Overhead. HS is not saddled with any cryptographic operations. Besides storing/relaying messages, HS simply matches tweets to followers by matching pseudorandom values. For any incoming pair (*encrypted tweet, cryptographic token*), HS only needs to lookup the token against those uploaded by corresponding followers (and, if needed, forward the ciphertext). In our implementation, tokens are outputs of SHA1 hash function, computed over OPRF evaluations on hashtags. Thus, complexity of the matching function depends only on the efficiency of the lookup algorithm. Although we ignore details of the Twitter "search" algorithm for matching tweets to subscription, we consider that matching hashtags in Twitter is conceptually similar to matching tokens in HS. In other words, if we were to implement a non-private baseline that matches tweets to followers *in the clear*, matching operation would actually incur comparable complexity. Also, HS lookup performance can be enhanced using, for example, binary search techniques, replication and load-balancing.

VII. Discussion and Extensions

To the best of our knowledge, Hummingbird is one of the first attempts to construct a privacy-enhanced micro-blogging OSN architecture. Tweeters control who can access their tweets while enforcing fine-grained access control. That is, a tweeter authorizes followers to read only tweets with specific hashtags. We believe this offers benefits to both tweeters and followers. For example, Alice can subscribe to CNN tweets with the hashtag #CNNTonight and avoid receiving all other CNN tweets that are not of interest. Furthermore, a tweeter's control over followers enables new revenue possibilities. For example, Financial Times might require followers to pay a subscription fee for accessing premium and time-critical content, e.g., stock market forecasts.

Moreover, followers can follow arbitrary tweets without disclosing their interests. For example, if Alice subscribes to tweets with hashtag #Caucus from NYT, no one learns her subject of interest.

A. Information Disclosure

As argued in Section III, operational requirements of HS impose a (minimal) privacy leakage. Any time a follow request is issued, HS learns who are the involved parties; later, when the request is finalized, HS also learns that the request has been accepted. In other words, HS always knows who follows whom and can build a full graph of tweeter-follower relations. Also, HS learns whenever: (1) a tweet matches a follow request, (2) two follow requests for the same tweeter are based on the same hashtag, and (3) two tweets by the same tweeter contain the same hashtag. However, (3) is avoidable, as discussed in Section VII-C below.

B. User Anonymity

Our focus is mainly on privacy; user anonymity is largely beyond the scope of this paper. In Hummingbird, all user identities are assumed to be known to everyone. However, pseudonymous or anonymization techniques could be used to protect identities of both followers and tweeters.

As far as followers, pseudonymity is perhaps the best achievable degree of privacy, since HS forwards all tweets matching a follower's interests to that follower. Thus, even if a follower's real identity is unknown, at the very least, there must be some persistent identifier or an account.

The same does not hold for tweeters: they can actually attain real anonymity, however, not without system modifications. In current OSNs, followers and tweeters are one and the same: they are all *users*. Suppose that we decouple the two roles and create follower-users and tweeter-users. The former would still create accounts much like they do today and would continue depositing tokens corresponding to hashtags of interest. However, tweeter-users would have

[2]Signatures for recurring hashtags can be cached so that each new tweet would only require one encryption operation.

no accounts at all. Instead, each time Bob has a tweet to post, he would connect to HS via some anonymous means (e.g., Tor) and communicate the tweet, same as in Hummingbird. This way, whenever Bob produces two tweets with different hashtags, HS would be unable to link them to the same author. If we employ this approach in tandem with the mechanism described below (for unlinking same-hashtag tweets) privacy would increase even further. Finally, we note that anonymity of tweeter-users would necessitate a change in our Follow protocol: instead of using HS as a store-and-forward conduit, Alice and Bob would need to run this protocol directly, away from HS.

We conclude that many aspects of achieving anonymity in a Twitter-like OSNs remain open and more work is clearly needed. In its current form, Hummingbird does not provide anonymity nor does it hide relationships between tweeters and followers. Greater privacy would be achieved if the central server could be oblivious of user identities and tweeter-follower relationships.

C. Unlinking Same-Hashtag Tweets

We sketch out a simple method for preventing HS from learning whether multiple tweets by the same tweeter contain the same hashtag.

The OPRF-based Follow protocol between Alice and Bob remains the same. But, instead of the token t corresponding to ht, Alice now deposits a temporary token: $t_{seq} = H_3(seq, t)$ at HS, where seq is the current sequence identifier and $H_3(\cdot)$ is yet another suitable cryptographic hash function. For his part, Bob now labels its tweets in the same manner. Periodicity of seq can be arbitrary: time-based (e.g., a week, a day or an hour) or volume-based, e.g., 100 or $1,000$ tweets. (The former makes more sense as it would not require fine-grained synchronization. Whenever the epoch changes – due to either time or volume – both tweeters and followers update seq, recompute t_{seq} and each follower deposits it at HS. Although increased frequency of re-depositing tokens at HS might be considered undesirable, we note that users tend to be connected most of the time and bandwidth involved in re-depositing is rather low; linear in the number of tokens for each follower.

Clearly, within the same seq epoch, all tweets by the same tweeter with the same ht would remain linkable by HS. However, consider two tweets from different epochs containing the same hashtag: they are labeled with $t_{seq} = H_3(seq, t)$ and $t_{seq'} = H_3(seq', t)$, respectively, where $seq \neq seq'$. Linking them is computationally infeasible due to the properties of $H_3(\cdot)$. On the other hand, as pointed out in Section III, the privacy gain obtained from this modification is of dubious value. The main reason is that HS retains its central role and sees all tokens updates by all followers. This allows it to perform very effective traffic analysis. For example, if Alice follows Bob on a given ht, HS would trivially see when both Alice's current token t_{seq} starts matching Bob's counterpart. HS would thus easily infer the periodicity of seq and, with high probability, link Bob's tweets with the same ht. On the other hand, there might still be some value in this method if followers subscribe to many hashtags by many tweeters. We defer further investigation of this topic to future work.

D. Collusions

As discussed in Section III, our HS adheres to the honest-but-curious (HbC) adversarial model. In particular, though it diligently follows all Hummingbird protocols, HS can attempt to violate privacy of either (or both) tweeters or followers. The former entails HS learning the hashtag ht^* used to derive the token t^* that accompanies a tweet. Whereas, HS violates follower privacy if it learns ht used to derive t previously deposited by a follower. We claim that, since HS is an HbC adversary, it does not create spurious (or phantom) followers. If HS were to be treated as a fully malicious adversary, it is easy to see that creating phantom followers would allow it to obtain cryptographic tokens on arbitrary hashtags and match them to existing tweets or follow requests.

On the other hand, HS might still try to collude with other users. In particular, a collusion between HS and a tweeter would immediately disclose all interests of subscribers pertaining to that tweeter. Similarly, a collusion with a follower, would allow HS to learn all subscribers who have any common interests with the colluding subscriber. Nevertheless, we claim that users (tweeters or followers) who collude with HS necessarily lose some of their own privacy. Suppose that Alice is one of Bob's followers on hashtag ht who has previously deposited the corresponding token t at HS during the *Finalize Request* protocol. In order to learn ht corresponding to t, HS must collude with (1) any Bob's follower on ht, e.g., Alice, or (2) Bob himself.

In the former case, Alice could reveal to HS ht that was used as secret OPRF input in the *Issue Request* protocol. By doing so, Alice clearly looses her own privacy with respect to ht. Whereas, HS learns the identities of all other Bob's followers on ht. This is a serious breach of privacy. However, this "feature" unfortunately appears to be unavoidable, since, as discussed in Section III-A, HS's ability to simultaneously match a given token to all of is followers is a fundamental component of Twitter and any similar OSN.

Now, suppose that HS colludes with Bob with the goal of learning ht corresponding to some t deposited at HS by one or more Bob's followers. As part of colluding with HS, Bob could compute its PRF over arbitrary hashtags until the output matches t. However, in general, this might be computationally infeasible if the hashtag of interest – ht – is not easily predictable by Bob, i.e., not in Bob's usual *repertoire*. We also note that, by revealing to HS that ht corresponding to t, Bob would give up privacy of all of his past and future tweets containing the same ht. We therefore consider that Bob has a low incentive to collude with HS.

E. Handling Retweets

Twitter allows forwarding other tweeter's tweets to one's own followers. Hummingbird has been designed with privacy in mind and one of its main goals is to allow tweeters to control who can access their data. For this reason, we do not allow retweets at this stage. Nevertheless, followers in Tweeter, Hummingbird or any other messaging system can always redistribute the content of the messages they decrypt. Effective means of addressing this issues are beyond this paper's scope. Similarly, Hummingbird does not allow followers to reply to a tweet. As Alice follows Bob on "private" hashtags, replying to a tweet would immediately sacrifice Alice's privacy.

F. Extending Hummingbird to Mobile Applications.

One of Twitter's most attractive features is its pervasiveness. User can tweet and follow others from virtually any computing device, e.g., a laptop, a tablet, a PDA or a smartphone. Our current prototype only supports Firefox 3.x. However, Hummingbird's low overhead makes it perfectly suitable for resource-constrained devices, such as smartphones. In fact, even in a smartphone setting, the overhead of most CPU-demanding operation in Hummingbird (i.e., RSA signatures) is still quite low. For example, on a 2010 HTC Nexus One, running Android 2.3.6, it takes less than $20ms$ to generate an RSA signature.

G. Support for Multiple Hashtags

The description of Hummingbird in Section IV assumes that all tweets and follow requests contain a single hashtag. We now describe how to efficiently extend our protocol for tweeting or issuing follow requests on multiple hashtags.

Tweeting with multiple hashtags: Bob tweets a message M and associates it with n hashtags, $ht_1^*, \ldots, ht_n^*$. Anyone with a follow request accepted on *any* of these hashtags should be able to read the message. We modify the tweeting protocol in Figure 3 as follows: Bob selects $k^* \leftarrow_R \{0,1\}^{\tau_1}$, and computes $ct^* = Enc_{k^*}(M)$. He then computes:

$$\{\delta_i^*\}_{i=1}^n, \leftarrow \{H(ht_i^*)^{d_b} \bmod N_b\}_{i=1}^n$$

Finally, Bob sends to HS:

$$\left(ct^*, \{Enc_{H_1(\delta_i^*)}(k)\}_{i=1}^n, \{H_2(\delta_i^*)\}_{i=1}^n\right)$$

The rest of the protocol involving matching at HS, as well as Alice's decryption, is straightforward and we omit it.

Following on multiple Hashtags: Alice follows Bob on *any* hashtags: $(ht_1, \ldots, ht_l)$. The Follow Request protocol in Figure 2 needs to be trivially extended to include l parallel Blind-RSA executions, one for each hashtag in $(ht_1, \ldots, ht_l)$. Thus, Alice obtains $(\delta_1, \ldots, \delta_l)$, i.e., Bob's RSA signatures on hashtags of interest. She then deposits at HS: $(t_1, \ldots, t_l) \leftarrow (H_2(\delta_1), \ldots, H_2(\delta_l))$.

H. Oblivious AES as OPRF

Another possibility for realizing OPRFs is to use Oblivious AES. An Oblivious AES construction involves a sender, on input a secret AES key s, and a receiver on input a message x. Using an oblivious evaluation of the AES circuit (about $30,000$ gates), e.g., relying on Yao's garbled circuits for secure two-party computation [54], the receiver can obtain $AES.Enc_s(x)$ without disclosing x to the sender, and without learning s. Therefore, assuming that AES is secure pseudorandom permutation, one could securely realize, in ROM, the OPRF $f_s(ht)$ as $H'(AES.Enc_s(ht))$. Use of Oblivious AES would remove virtually any overhead during tweeting as the tweeter would no longer need to perform any public-key operation. However, overhead incurred by the oblivious evaluation of the AES garbled circuit (e.g., see [45]) is still relatively high compared to the Blind-RSA protocol. Therefore, we do not use it as it would significantly slow down the *follow* requests. Nonetheless, as improved constructions become available, we can replace Blind RSA based techniques with Oblivious AES with no architectural change.

VIII. Related Work

This section reviews related work: we distinguish between (i) results focusing on OSN security and privacy, (ii) techniques for securing publish/subscribe networks, and (iii) constructs for Attribute-based Encryption.

A. Privacy in OSNs

Increasing proliferation and popularity of Online Social Networks (OSNs) prompted privacy concerns with the amount and sensitivity of disseminated and collected personal information. Consequently, the security research community started to explore emerging issues and challenges. A number of recent studies, such as [14, 21, 23, 53], considered privacy in Facebook, MySpace, and other OSNs. Recently, [5] overviewed the state of the art in OSN privacy from different perspectives (including psychology and economics) and highlighted a number of research gaps. The work in [32] measured the extent of personally identifiable information disclosed by Twitter users. Also specific to Twitter is the analysis of privacy leaks in [41] and [40].

Several cryptographic protocols have been proposed to improve confidentiality, access control, or anonymity in OSNs. One proposal closely related to Hummingbird is #h00t [6]. It allows a group of users (communities that share a common password or plaintext hashtag) to communicate with privacy. Similar to Hummingbird, cryptographic tags in #h00t are computed by hashing plaintext keywords. #h00t also leverages hash collisions to provide *deniability*, i.e., the ability for an user to prove that they were not tweeting/following about a particular hashtag.. The work in [17] applies the concept of "virtual private networks" to OSN, i.e., it establishes a confidential channel between "friends" in order to share sensitive data. A similar approach is taken by FaceCloack [38], where sensitive data is encrypted and stored on third party servers (distinct from the OSN provider), while only authorized users holding decryption keys can access them. NoYB [31] focuses on user privacy with respect to the OSN provider and also tries to conceal users that are encrypting their contents. Finally, Scramble! [8] uses broadcast encryption [19] for improved access control on Facebook, i.e., allowing each user to specify the recipients of shared information, similar to the concept of *circles* in Google+.

Decentralized OSNs have been advocated for privacy-aware social networking[1–3]. For example, [7] provides a cryptographic API that achieves improved access control, anonymity and confidentiality. Whereas, our approach focuses on privacy for centralized OSNs, a social-network model that supports high availability and real-time content dissemination. In fact, decentralized architectures – e.g., those based on p2p networks [15] – might hinder *real-time* availability of information (a crucial feature in Twitter) or require users to buy cloud storage for their data [50].

B. Privacy-Friendly Publish/Subscribe Networks

The service provided by Twitter could be considered as a kind of a Publish/Subscribe network (Pub/Sub). In Pub/Sub, publishers do not program messages to be sent directly to specific subscribers. Rather subscribers register their interest in an event, or a pattern of events, and are subsequently asynchronously notified of events generated by publishers. Subscribers register arbitrary complex queries over events to a multitude of brokers (usually organized in an hierarchy) that receive events by publishers and notify subscribers. (For more details on Pub/Sub, we refer to [22].) In contrast, Twitter has a single central server, handles simple keyword-based queries and requires interaction between followers and tweeters. Nevertheless, we review recent Pub/Sub privacy-enhancing techniques an discuss their (un)suitability to the Twitter scenario.

The work in [48] focuses on confidentiality in content-based Pub/Sub systems and introduces some techniques for efficient matching at the broker, who learns nothing about the data being matched. Another recent result [51] uses multiple-level commutative encryption to privately share information in an hierarchical content-based multi-broker Pub/Sub system. While data producers and consumers might not know each other, the system requires each user to interact and trust each peer at two-hop distance in the network. Finally, [33] proposes a cryptographic mechanism, based on Attribute-Based Encryption (ABE) [28] and Searchable Encryption [13, 52], to support confidentiality for events and matching filters.

All above solutions require publishers and subscribers to trust (i) each other [48], (ii) the brokers [51], or (iii) a third party [33]. Thus, none can offer data privacy in Twitter-like settings. Tweeters and followers might have no mutual knowledge and no third party should be trusted to access sensitive data.

C. Attribute-Based Encryption

Attribute-based encryption (ABE) was introduced by Sahai and Waters [49]. In ABE, decryption capability can be given to any party whose decryption keys satisfy arbitrary (encryptor-selected) policy. In particular, in [49], a ciphertext and decryption keys are labeled with set of attributes. Thus, decryption is possible only if the sets of ciphertext and decryption key attributes have at least d attributes in common. ABE was later extended by [29] and [11]. The former proposed Key-Policy ABE where keys are associated to access structures and a ciphertext can be decrypted by a key only if the ciphertext attributes satisfy the key access structure. Whereas, the latter proposed Ciphertext-Policy ABE where ciphertexts are associated with access structures and keys with attributes: a ciphertext can be decrypted only if the key attributes satisfy its access structure.

In general, ABE could be used in Hummingbird substituting cryptographic tokens of tweets with attributes required for decryption. Similarly, tokens computed by followers could be replaced by attributes of their decryption keys. However, there are some issues complicating the adoption of ABE in Hummingbird.

First, it is unclear how to achieve follower privacy in ABE, i.e., how can a follower obtain the decryption key for "attributes" of her choice without leaking her interests to the tweeter. Also, matching tweets and follow requests would require the server to get the decryption key of each follower. In other words, checking if the set of attributes associated with a tweet and the one associated with a follow request are disjoint, requires the secret decryption key of the follower.

Finally, ABE schemes incur considerable overhead by requiring a number of bilinear-map computations linear in the number of attributes associated to a ciphertext or a decryption key. For example, encryption/decryption operations in [11] are one order of magnitude slower than in Hummingbird.

IX. Conclusion

This paper presented one of the first efforts to mitigate rampant lack of privacy in modern micro-blogging OSNs. We analyzed privacy issues in Twitter and laid out an architecture (called Hummingbird) that offers a Twitter-like service with increased privacy guarantees for tweeters and followers alike. While the degree of privacy attained is not absolute, it is still valuable considering current complete lack of privacy and some fundamental limitations inherent to the centralized gather/scatter message dissemination paradigm. We implemented Hummingbird architecture as a research prototype and evaluated its performance. Since almost all cryptographic operations are conducted off-line, and none is involved to match tweets to followers, the resulting costs and overhead are very low. Our work clearly does not end here. In particular, several extensions, including revocation of followers, anonymity for tweeters as well as unlinking same-hashtag tweets, require further consideration and analysis.

Acknowledgments. We gratefully acknowledge Filipe Beato, Elie Bursztein, and Bruno Crispo for helpful discussions in the early stage of protocol design, as well as David Molnar, Elaine Shi, Daniele Perito, and Matthew Wright for their valuable feedback.

References

[1] DSNP: The Distributed Social Network Protocol. http://www.complang.org/dsnp/.

[2] SocialFortress. https://socialfortress.com/.

[3] The Diaspora Project. http://diasporaproject.org/.

[4] Twitter Privacy Policy. https://twitter.com/privacy, 2011.

[5] A. Acquisti, B. Van Alesenoy, E. Balsa, B. Berendt, D. Clarke, C. Diaz, B. Gao, S. Gurses, A. Kuczerawy, J. Pierson, F. Piessens, R. Sayaf, T. Schellens, F. Stutzman, E. Vanderhoven, and R. De Wolf. The SPION Project. https://www.cosic.esat.kuleuven.be/publications/article-2077.pdf.

[6] D. Bachrach, C. Nunu, D. S. Wallach, and M. Wright. #h00t: Censorship resistant microblogging. *CoRR*, abs/1109.6874, 2011.

[7] M. Backes, M. Maffei, and K. Pecina. A security API for distributed social networks. In *NDSS*, 2011.

[8] F. Beato, M. Kohlweiss, and K. Wouters. Scramble! your social network data. In *PETS*, pages 211–225, 2011.

[9] M. Belenkiy, J. Camenisch, M. Chase, M. Kohlweiss, A. Lysyanskaya, and H. Shacham. Randomizable Proofs and Delegatable Anonymous Credentials. In *CRYPTO*, pages 108–125, 2009.

[10] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, 2003.

[11] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *IEEE Symposium on Security and Privacy (S&P)*, pages 321–334, 2007.

[12] D. Boneh and X. Boyen. Short signatures without random oracles. In *EUROCRYPT*, pages 56–73, 2004.

[13] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key Encryption with Keyword Search. In *EUROCRYPT*, pages 506–522, 2004.

[14] D. Boyd and N. Ellison. Social Network Sites: Definition, History, and Scholarship. *Journal of Computer-Mediated Communication*, 13(1):210–230, 2008.

[15] S. Buchegger, D. Schiöberg, L. H. Vu, and A. Datta. PeerSoN: P2P Social Networking - Early Experiences and Insights. In *SNS*, pages 46–52, 2009.

[16] D. Chaum. Blind signatures for untraceable payments. In *CRYPTO*, volume 82, pages 199–203, 1983.

[17] M. Conti, A. Hasani, and B. Crispo. Virtual private social networks. In *CODASPY*, pages 39–50, 2011.

[18] E. De Cristofaro and G. Tsudik. Practical Private Set Intersection Protocols with Linear Computational and Bandwidth Complexity. In *FC*, pages 143–159, 2010. http://eprint.iacr.org/2009/491.

[19] C. Delerablée, P. Paillier, and D. Pointcheval. Fully Collusion Secure Dynamic Broadcast Encryption with Constant-Size Ciphertexts or Decryption Keys. In *PAIRING*, pages 39–59, 2007.

[20] K. Dozier. CIA Tracks Revolt by Tweet, Facebook. http://abcn.ws/uFdpVQ, 2011.

[21] C. Dwyer, S. Hiltz, and K. Passerini. Trust and Privacy Concern Within Social Networking Sites: A Comparison of Facebook and MySpace. In *AMCIS*, pages 1–13, 2007.

[22] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[23] L. Fang and K. LeFevre. Privacy wizards for social networking sites. In *WWW*, pages 351–360, 2010.

[24] M. Fischetti. Data theft: Hackers attack. *Scientific American*, 305(100), 2011.

[25] Fox News. Facebook Retreats on Privacy. http://www.foxnews.com/scitech/2011/11/11/facebook-retreats-on-privacy/.

[26] M. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword Search and Oblivious Pseudorandom Functions. In *TCC*, 2005.

[27] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2004.

[28] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *CCS*, pages 89–98, 2006.

[29] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *CCS*, pages 89–98, 2006.

[30] J. Gruenspecht. "Reasonable" Grand Jury Subpoenas: Asking for information in the age of big data. *Harvard Journal of Law & Technology*, 24(2):543–562, 2011.

[31] S. Guha, K. Tang, and P. Francis. NOYB: Privacy in Online Social Networks. In *WONS*, pages 49–54, 2008.

[32] L. Humphreys, P. Gill, and B. Krishnamurthy. How much is too much? Privacy Issues on Twitter. In *ICA*, pages 1–29, 2010.

[33] M. Ion, G. Russello, and B. Crispo. Supporting Publication and Subscription Confidentiality in Pub/Sub

Networks. In *SECURECOMM*, pages 272–289, 2010.

[34] S. Jarecki and X. Liu. Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection. In *TCC*, pages 577–594, 2009.

[35] S. Jarecki and X. Liu. Fast Secure Computation of Set Intersection. In *SCN*, 2010.

[36] H. Jones and J. Soltren. Facebook: Threats to privacy. In *Ethics and the law on the electronic frontier course*, pages 1–76, 2005.

[37] J. Kincaid. This is the second time a Google engineer has been fired for accessing user data. http://techcrunch.com/2010/09/14/google-engineer-fired-security/, 2010.

[38] W. Luo, Q. Xie, and U. Hengartner. FaceCloak: An Architecture for User Privacy on Social Networking Sites. In *CSE*, pages 26–33, 2009.

[39] A. Mac Guill. US ruling sets troubling precedent for social media privacy. http://t.co/Z2OZMaFo.

[40] H. Mao, X. Shuai, and A. Kapadia. Loose Tweets: An Analysis of Privacy Leaks on Twitter. In *WPES*, pages 1–12, 2011.

[41] B. Meeder, J. Tam, P. Kelley, and L. Cranor. RT@ IWantPrivacy: Widespread violation of privacy settings in the Twitter social network. In *W2SP*, 2010.

[42] A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of applied cryptography*. CRC, 1997.

[43] Mozilla Developer Network. LiveConnect. https://developer.mozilla.org/en/LiveConnect, 2011.

[44] M. Naor and O. Reingold. Number-theoretic Constructions of Efficient Pseudorandom Functions. In *FOCS*, 1997.

[45] J. Nielsen, P. Nordholt, C. Orlandi, and S. Burra. A New Approach to Practical Active-Secure Two-Party Computation. http://eprint.iacr.org/2011/091, 2011.

[46] Official Twitter Blog. One hundred million voices. http://blog.twitter.com/2011/09/one-hundred-million-voices.html, 2011.

[47] D. Pointcheval and J. Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, 2000.

[48] C. Raiciu and D. S. Rosenblum. Enabling Confidentiality in Content-Based Publish/Subscribe Infrastructures. In *SECURECOMM*, pages 1–11, 2006.

[49] A. Sahai and B. Waters. Fuzzy identity-based encryption. In *EUROCRYPT*, pages 457–473, 2005.

[50] A. Shakimov, H. Lim, R. Caceres, L. Cox, K. Li, D. Liu, and A. Varshavsky. Vis-á-vis: Privacy-preserving online social networking via virtual individual servers. In *COMSNETS*, pages 1–10, 2011.

[51] A. Shikfa, M. Önen, and R. Molva. Privacy-Preserving Content-Based Publish/Subscribe Networks. In *IFIP SEC*, pages 270–282, 2009.

[52] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *S&P*, pages 44–55, 2000.

[53] Y. Wang, S. Komanduri, P. Leon, G. Norcie, A. Acquisti, and L. Cranor. I regretted the minute I pressed share: A qualitative study of regrets on Facebook. In *SOUPS*, 2011.

[54] A. Yao. Protocols for secure computations. In *FOCS*, 1982.