Enabling Private Conversations on Twitter

Indrajeet Singh[1], Michael Butkiewicz[1], Harsha V. Madhyastha[1],
Srikanth V. Krishnamurthy[1], Sateesh Addepalli[2]
[1] University of California, Riverside [2] Cisco Systems

ABSTRACT

User privacy has been an increasingly growing concern in online social networks (OSNs). While most OSNs today provide some form of privacy controls so that their users can protect their shared content from other users, these controls are typically not sufficiently expressive and/or do not provide fine-grained protection of information. In this paper, we consider the introduction of a new privacy control—group messaging on Twitter, with users having fine-grained control over who can see their messages. Specifically, we demonstrate that such a privacy control can be offered to users of Twitter *today* without having to wait for Twitter to make changes to its system. We do so by designing and implementing `Twitsper`, a wrapper around Twitter that enables private group communication among existing Twitter users while preserving Twitter's commercial interests. Our design preserves the privacy of group information (i.e., who communicates with whom) both from the `Twitsper` server as well as from undesired `Twitsper` users. Furthermore, our evaluation shows that our implementation of `Twitsper` imposes minimal server-side bandwidth requirements and incurs low client-side energy consumption. Our `Twitsper` client for Android-based devices has been downloaded by over 1000 users and its utility has been noted by several media articles.

1. INTRODUCTION

OSNs have gained immense popularity in the last few years since they allow users to easily share information with their contacts and to even discover others of similar interests based on information they share. However, not all shared content is meant to be public; users often need to ensure that the information they share is accessible to only a select group of people. Though legal frameworks can help limit with whom OSN providers can share user data, users are at the mercy of controls provided by the OSN to protect the content they share from other users. In the absence of effective controls, users concerned about the privacy of their information are likely to connect with fewer users, share less information, or even avoid joining OSNs altogether.

Previous proposals to address these privacy concerns on *existing OSNs* either (a) jeopardize the commercial interests of OSN providers [31, 23] if these solutions are widely adopted and thus, are likely to be disallowed, or (b) require users, who are currently accustomed to free access to OSNs, to pay for improved privacy [26, 42, 3]. On the other hand, though *new OSNs* have been developed with privacy explicitly in mind [16, 7], these OSNs have seen limited adoption because users are virtually "locked in" to OSNs on which they have already invested significant time and energy to build social relationships. Consequently, users have, in many cases today, raised privacy-related concerns in the media and organizations such as the EFF and FTC have tried to coerce OSNs to make changes. Though OSNs have introduced new privacy controls in response to these concerns (e.g., Facebook friend lists, Facebook groups, Google+ circles), such controls do not provide sufficiently fine-grained protection.

In light of this, we consider the privacy shortcomings on Twitter, one of the most popular OSNs today [17]. Twitter offers two kinds of privacy controls to users—a user can either share a message with all of her followers or with one of her followers; there is no way for a user on Twitter to post a *tweet* such that it is visible to only a subset of her followers. In this paper, we fill this gap by providing fine-grained controls to Twitter users, enabling them to conduct private *group communication*. Importantly, we provide this fine-grained privacy control to Twitter users by implementing a wrapper that builds on Twitter's existing API, and hence, users do not have to wait for Twitter to make any changes to its service.

As our main contribution, we design and implement `Twitsper`, a wrapper around Twitter that provides the option of private group communication for users, without requiring them to migrate to a new OSN. Unlike other solutions for group communication on Twitter [8, 19, 20], `Twitsper` ensures that Twitter's commercial interests are preserved and that users do not need to trust `Twitsper` with any private information. Further, in contrast to private group communication on other OSNs (e.g., Facebook, Google+), in which a reply/comment on information shared with a select group is typically visible to all recipients of the original posting, `Twitsper` strictly enforces privacy requirements as per a user's social connections (all messages posted by a user are visible only to the user's followers).

When designing `Twitsper`, we considered various choices for facilitating the controls that we desire; surprisingly, a relatively simple approach emerged as the best fit for fulfilling our objectives. Thus, our `Twitsper` implementation is based on this simple design which combines a Twitter client (that retains much of the control logic) with a server that maintains minimal state. Importantly, we ensure that no privately shared content is revealed to the `Twitsper` server, and furthermore, the privacy of group memberships is also preserved from both the `Twitsper` server and from other undesired users. Our evaluation demonstrates that

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA
Copyright 2012 ACM 978-1-4503-1312-4/12/12 ...$15.00.

this simple design does achieve the best trade-offs between several factors such as backward compatibility, availability, client-side energy consumption, and server-side resource requirements.

Overall, our implementation of Twitsper is proof that users can be empowered with fine-grained privacy controls on existing OSNs, without waiting for OSN providers to make changes to their platform. Our client-side implementation of Twitsper for Android phones has been downloaded by over 1000 users and several articles in the media have acknowledged its utility in improving privacy and reducing information overload on Twitter.

2. RELATED WORK

Characterizing privacy leakage in OSNs: Krishnamurthy and Willis characterize the information that users reveal on OSNs [34] and how this information leaks [35] to other entities on the web (such as social application providers and advertising agencies). Our thesis is that legal measures are necessary to ensure that OSN providers do not leak user information to third-parties. However, it is not in the commercial interests of OSN providers to support systems that hide information from them. Therefore, we focus on enabling users to protect their information from other *undesired* users, rather than from OSN providers.

Privacy controls offered by OSNs: Google+ and Facebook permit any user to share content with a *circle* or *friend list* comprising a subset of the user's friends. However, anyone who comments on the shared content has no control; the comment will be visible to all those with whom the original content was shared. Even worse, on Facebook, if Alice comments on a friend Bob's post, Bob's post becomes visible to Alice's friend Charlie even if Bob had originally not shared the post with Charlie. Facebook also enables users to form groups; any information shared with a group is not visible to users outside the group. However, a member of the group has to necessarily share content with all other members of a group, even if some of them are not her friends. Twitter, on the other hand, enables any user to restrict sharing of her messages either to only all of her followers (by setting her account to *private* mode) or to exactly one of her followers (by means of a *Direct Message*), but not to a proper subset. We extend Twitter's privacy model to permit private *group communication*, ensuring that the privacy of a user's reply to a message shared with a group is in keeping with the user's social connections.

Distributed social networks: Several proposals to improve user privacy on OSNs have focused on de-centralizing OSNs (e.g., Vis-a-Vis [42], Confidant [37], DECENT [33], Polaris [45], and PeerSoN [26]). These systems require a user to store her data in the cloud or on her own or her friends' personal devices, thus removing the need for the user to trust a central OSN provider. However, users have put in tremendous effort in building their social connections on today's OSNs [4, 17], and rebuilding these connections on a new OSN is not easy. Thus, unlike these prior efforts, we build a backward-compatible privacy wrapper on Twitter.

Improving privacy in existing OSNs: With Lockr [44], the OSN hosting a user's content is unaware of with whom a user is sharing content; Lockr instead manages content sharing. Other systems allow users to share encrypted content, either by posting the encrypted content directly on OSNs [31, 23, 24] or via out-of-band servers [13]. Users can share the decryption keys with a select subset of their connections (friends). Hummingbird [28] is a variant of Twitter in which the OSN supports the posting of encrypted content in such a manner that preserves user privacy. Narayanan et al. [39] ensure users can keep the location information that they divulge on OSNs private via private proximity testing. All of these techniques either prevent OSN providers from interpreting user content, or hide users' social connections from OSNs. Since neither is in the commercial interests of OSN providers, these solutions are not sustainable if widely adopted. In contrast, we respect the interests of OSN providers while exporting privacy controls to users.

Category	%
Consider privacy a concern	77
Would like to control who sees information they post	70
Declined follower requests owing to privacy concerns	50

Table 1: Results of survey about privacy shortcomings on Twitter.

Group communication: Like Twitsper, listserv [30] enables communication between groups of users. However, unlike with Twitsper, group communications on listserv lack a social structure and listserv was never designed with privacy in mind. Prior implementations of group messaging on Twitter, such as Twitter Groups [20], GroupTweet [8], and Tweetworks [19], have either not focused on privacy—they require users to trust them with their private information—or require users to join groups outside their existing social relationships on Twitter. Similar to Twitsper, a recent workshop paper [43] advocated the use of a wrapper that offers private group communication on Twitter, but unlike Twitsper, they ignored the leakage of private information, such as the sizes of conversation groups, to the server maintained by the wrapper.

3. MOTIVATING USER SURVEY

While privacy concerns with OSNs have received significant coverage [34, 35], the media has mostly focused on leakage of user information on OSNs to third-parties such as application providers and advertising agencies. Our motivation is the need for a more basic version of privacy on OSNs—protecting content shared by a user from other users on the OSN, which has began to receive some attention [11].

To gauge the perceived need amongst users for this form of privacy, we conducted an IRB approved user study across 78 users of Twitter [1]. Our survey questioned the participants about the need they see for privacy on Twitter, the measures they have taken to protect their privacy, and the controls they would like to see introduced to improve privacy. Table 1 summarizes the survey results. More than three-fourths of the survey participants are concerned about the privacy of the information they post on Twitter, and an almost equal fraction would like to have better control over who sees their content. Further, rather tellingly, half the survey takers have at least once rejected requests to connect on Twitter in order to protect their privacy. These numbers motivate the necessity of enabling users on Twitter to privately exchange messages with a subset of their followers, specifically allowing them to choose which subset to share a message with on a per-message basis.

4. DESIGN OBJECTIVES

Given the need for enabling private group messaging on Twitter, we next design Twitsper to provide fine-grained privacy controls to Twitter users. Our over-arching objective in developing Twitsper is to offer these controls to users without having to wait for Twitter to make any changes to their service. Our design for Twitsper is guided by three primary goals.

Backward compatible: Rather than developing a new OSN designed with better user controls in mind (e.g., proposals for distributed OSNs [42, 26, 3]), we want our solution to be compatible with Twitter. This goal stems from the fact that Twitter already has

[1] Participant details removed for anonymity reasons

Proposal	Backward Compatible	Preserves Commercial Interests	No Added Trust Required
Distributed OSNs	×	×	✓
Encryption	✓	×	✓
Separating content providers from social connections	✓	×	×
Existing systems for group messaging on Twitter	✓	✓	×
Twitsper	✓	✓	✓

Table 2: Comparison of Twitsper with previous proposals for improving user privacy on OSNs.

an extremely large user base—over 100 million active users [21]. Since the value of a network grows quadratically with the growth in the number of users on it (the network effect [36]), Twitter users have huge value locked in to the service. To extract equal value from an alternate social network, users will not only need to re-add all of their social connections, but will further require all of their social contacts to also shift to the new service. Therefore, we seek to provide better privacy controls to users by developing a wrapper around Twitter, eliminating the burden on users of migrating to a new OSN and thus maximizing the chances of widespread adoption of Twitsper.

Preserves commercial interests: A key requirement for Twitsper is that it should not be detrimental to the commercial interests of Twitter. For example, though a user can exchange encrypted messages on Twitter to ensure that she shares her content only with those with whom she shares the encryption keys, this prevents Twitter from interpreting the content hosted on its service. Since Twitter is a commercial for-profit entity and offers its service for free, it is essential that Twitter be able to interpret content shared by its users. Twitter needs this information for several purposes: to show users relevant advertisements, to recommend applications of interest to the user, and to suggest others of similar interest with whom the user can connect. Though revealing user-contributed content to Twitter opens the possibility of this data leaking to third-parties (either with or without the knowledge of the provider), user content can be insured against such leakage via legal frameworks (e.g., enforcement of privacy policies [22]) or via information flow control [46]. On the other hand, protecting a user's content from other users requires enabling the user with better controls—our focus in building Twitsper.

No added trust: In attempting to give users better controls without waiting for Twitter to change, we want to ensure that users do not have another entity to trust in Twitsper; users already have to trust Twitter with their information. Increasing the number of entities that users need to trust is likely to deter adoption since users would fear the potentially greater opportunity for their information to leak to third-parties. Therefore, we seek to ensure that users do not need to share with Twitsper's servers any information they want to protect, such as their content or their login credentials. Tools such as TaintDroid [29] can be used to verify that Twitsper's client application does not leak such information to Twitsper's servers. We design Twitsper for the setting where Twitsper's servers are not malicious by nature, but are inquisitive listeners; this attacker model is similar to that used in prior work (e.g., [40]).

Table 2 compares our proposal with previous solutions for improving user privacy on OSNs. Unlike proposals for distributed OSNs, Twitsper enables users to reuse their social connections on Twitter, and unlike calls for exchange of encrypted content, we respect Twitter's commercial interests. Moreover, we introduce user controls via Twitsper without adding another entity for users to trust, unlike proposals such as Lockr [44], which call for the separation of social connections from content providers. Lastly, in contrast to prior implementations of group messaging on Twitter such as GroupTweet [8], Tweetworks [19], and Twitter Groups [20], we ensure that Twitter is privy to private conversations but Twitsper is not.

API call	Function
PrivSend(msg, group)	Send *msg* to all users specified in *group*
isPriv?(msg)	Determine if *msg* is a private message
PrivReply(msg, orig_msg)	Send *msg* to all of the user's followers who received *orig_msg*

Table 3: Twitsper's API beyond normal Twitter functionality.

5. Twitsper DESIGN

Next, we present an overview of Twitsper's design. We consider various architectural alternatives and discuss the pros and cons with each. Our design objectives guide the choice of the architecture that presents the best trade-offs. As mentioned earlier, surprisingly, a fairly simple approach seems to yield the best trade-off and is thus, used as the basic building block in Twitsper.

Basic definitions: First, we define a few terms related to Twitter and briefly explain the Twitter ecosphere.

- ***Tweet*:** A tweet is the basic mode of communication on Twitter. When a user posts a tweet, that message is posted on the user's Twitter page (i.e., http://twitter.com/username), and is seen on the timeline of everyone following the user.
- ***Direct Message*:** A direct message is a one-to-one private tweet from one user to a specific second user, and is possible only if the latter follows the former.
- @ ***Reply*:** A user can use a @*reply* message to reply to another user's tweet; this message will also appear on the timeline of anyone following both users.
- ***Twitter page*:** Every user's Twitter page(http://twitter.com/username) contains all tweets and @*reply* messages posted by the user. By default, this page is visible to anyone, even those not registered on Twitter. If a user sets her Twitter account to be private, all messages on her page are visible to any of the users following her account.
- ***Timeline*:** A user's timeline is the aggregation of all tweets, direct messages, and @*reply* messages (sorted in chronological order) visible to that user. In addition to her timeline, note that a user can view *any* tweet or @*reply* message posted by any user that she follows by visiting that user's Twitter page.
- ***List*:** Twitter allows every user to create lists—groups of Twitter users selected by the user. Lists can either be public and world viewable, or private and viewable to the user alone.
- ***Whisper*:** Twitsper's private messaging primitive to allow a user to contact any subset of followers

Twitter associates every tweet, Direct Message, user, and list with a unique ID.

Interface: Our primary goal is to extend Twitter's privacy model. In addition to sharing messages with all followers (tweet) or precisely one follower (Direct Message), we seek to enable users to privately share messages with a non-empty proper subset of their followers. To do so, we extend Twitter's API with the additional functionality shown in Table 3. We present the algorithmic representations of these API calls later.

First, the *PrivSend* API call allows users to post *private* messages that can be seen by one or more members in the user's network, who are *specifically* chosen to be the recipients of such a message. However, simply enabling a message to be shared with a group of users is insufficient. To enable richer communication, it is necessary that the recipients of a message (shared with a group) be able to reply back to the group. In the case of discussions that need not be kept private, a user may choose to make her reply public so that others with similar interests can discover her. However, when Nina responds to a private message from Jack, it is unlikely that Nina will wish to share her reply with all the original target recipients of Jack's message since many of them may be "unconnected" to her. Nina will likely choose to instead restrict the visibility of her reply to those among the recipients of the original message whom she has approved as her followers. Therefore, the *PrivReply* API call enables replies to private messages, while preserving social connections currently established on Twitter via follower-followee relationships. Finally, the *isPriv?* API call is necessary to determine if a received message is one to which a user can reply with *PrivReply*. Hereafter, we refer to the messages exchanged with the *PrivSend* and *PrivReply* calls as whispers.

It is important to note that, since our goal is to build a wrapper around Twitter, rather than build a new OSN with these privacy controls, this extended API has to build upon Twitter's existing API for exchanging messages. Though Twitter's API may evolve over time, we rely here on simple API calls—to post a tweet to all followers and to post a Direct Message to a particular follower—that are unlikely to be pruned from Twitter's API. Also note that, in some cases, multiple rounds of replies to private messages can result in the lack of context for some messages for some recipients, since all recipients of the original whisper may not be connected with each other. In the trade-off between privacy and ensuring context, we choose the former in designing Twitsper.

Architectural choices: Next, we discuss various architectural possibilities that we considered for Twitsper's design, to support the interface described above. While it may be easy for Twitter to extend their interface to support private group messaging, we note that Twitter has not yet done so in spite of the need for this amongst its users. Therefore, our focus is in designing Twitsper to offer this privacy control to users without having to wait for Twitter to make any changes.

Using a supporting server: The simplest architecture that one can consider for Twitsper is to have clients send a whisper to a group of users (represented by a list on Twitter) by sending a Direct Message to each of those users. To enable replies, when a client sends a whisper, it can send to the supporting server the identifiers of the Direct Messages and the ID of the Twitter list which contains the recipients. Thus, a user can query this supporting server to check if a received Direct Message corresponds to a whisper and to obtain the ID of the associated Twitter list. When the user chooses to reply to a whisper, the user's client can retrieve the Twitter list containing the recipients of the original whisper, locally compute the intersection between those recipients and the user's followers, and then send Direct Messages to all those in the intersection.

If the supporting server is unavailable, users can continue to use Twitter as before, except that the metadata necessary to execute the *isPriv?* and *PrivReply* API calls cannot be retrieved from the server. However, the client software can be modified to allow a recipient to obtain relevant mappings (ID of the list of recipients of a whisper) from the original sender. Another option is to have the client embed the ID of the list associated with a whisper in every Direct Message sent out as part of a whisper. However, given Twitter's 140 character limit per Direct Message, this can be a significant imposition, reducing the permissible length of the message content.

This design places much of the onus on the client and may result in significant energy consumption for the typical use case of Twitter access from smartphones. On the flip side, in this architecture, the content posted by a user is not exposed to the supporting server, i.e., privacy of user content from Twitsper's server is preserved. The server is simply a facilitator of group communications across a private group and only maintains metadata related to whispers (we discuss later in Section 6 how we protect the privacy of this metadata as well from the supporting server). Further, Twitter is able to see users' postings and thus its commercial interests are protected. We note that the alternative of the client sending messages to the supporting server for retransmission to the recipients is not an option, since this would require users to trust the supporting server with the content of their messages.

This design however does have some shortcomings. Twitter lacks sufficient context to recognize that the set of Direct Messages shared to send a whisper constitute a single message rather than a local trending topic. Similarly, Twitter cannot link replies with the original message, since all of this state is now maintained at the supporting server.

Posting encrypted content: To address the shortcoming in the previous architecture of being unable to link replies to the original whispers, in our next candidate architecture, we consider clients posting a whisper just as they would a public message (tweet) but encrypt it with a group key which is only shared with a select group of users (who are the intended recipients of the message). This reduces the privacy problem to a key exchange problem for group communications. An out-of-band key exchange is possible.

However, since only intended recipients can decrypt a tweet, Twitter's commercial interests are compromised. Furthermore, filtering of encrypted postings not intended for them is necessary at the recipient's side; if not, a user's Twitter client will display indecipherable noise from these postings. In other words, the approach is not backward compatible with Twitter. Note here that if these issues are resolved, e.g., by sharing encryption keys with Twitter, encryption can be used with any of the other architectural choices considered here to enhance privacy.

Using community pages to support anonymity: Alternatively, one may try to achieve anonymity and privacy by obfuscation. Clients post tweets to a obfuscation server, which in turn re-posts messages on behalf of users to a common "community" account on Twitter. Except for the server, no one else is aware of which message maps to which user. When a user queries the obfuscation server for her timeline, the server returns a timeline that consists of messages from her original timeline augmented with messages meant for that user from the "community" page. The obfuscation prevents the exposure of private messages to undesired users. Since the "community" page is hosted on Twitter, the shortcoming of the encryption-based architecture is readily addressed—Twitter has access to all information unlike in the case of encryption. An approach similar to this was explored in [41].

However, this architecture has several drawbacks. First, Twitter cannot associate messages with specific users; this precludes Twitter from profiling users for targeted advertisements and such. Second, all users need to trust the obfuscation server with the contents of their messages. Finally, since the architecture is likely to heavily load the server (due to the scale), the viability of the design in practice becomes questionable. When the server is unavailable, no private messages can be sent or received.

Using dual accounts: In our last candidate architecture, every user maintains two accounts. The first is the user's existing account, and a second private account (with no followers or followees) is used for sending whispers. When Alice wishes to send a whisper

Design	Twitter's interests preserved	No added trust	Easily scales	Same text size	Always available	Linkable to orig message
Supporting server	✓	✓	✓	✓	✓	×
Embed lists	✓	✓	✓	×	✓	×
Encryption	×	✓	✓	✓	×	✓
Community pages	×	×	×	✓	×	×
Dual accounts (No longer possible)	×	✓	✓	✓	✓	✓

Figure 1: Comparison of architectural choices.

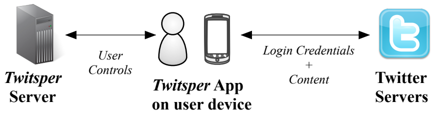


Figure 2: System architecture using supporting server.

to Bob and Charlie, she posts an @*reply* message from her private account to Bob's and Charlie's private accounts. Since Alice's private account has no followers, these @*reply* messages are visible to no users other than to the intended recipients. However, as of mid-2009, Twitter discontinued the "capability" of @*reply* messages between disconnected users after concluding that less than 1% of the users found this feature useful and that it contributes to spam messages [14]. Thus, @*reply* messages posted from these disconnected private accounts will not be visible to intended recipients. Other problems with this architectural choice are that Twitter is unable to associate private messages with the normal accounts of users and responding to private messages is a challenge.

Figure 1 summarizes the comparison of the various architectural choices with respect to our design goals. While no solution satisfies all desirable properties, we see that the use of a supporting server presents the best trade-off in terms of simplicity and satisfying our goals. Therefore, we choose this to be the architectural choice for implementing `Twitsper`, as shown in Figure 2.

While the basic structure of the architecture is simple as discussed above, there exist certain challenges in making the server and other undesired users oblivious to the specifics of a group conversation. We discuss these issues and our approaches for handling them in the following section.

6. PROTECTING PRIVACY

With the supporting server architecture, users do not directly send content to the `Twitsper` server. However, there is metadata that is provided to the server in order to support group conversations —the mapping of Direct Message IDs to list IDs. This metadata could reveal the identities of the members that belong to a private conversation or the group size, and a user may desire to keep such information private. Hence, we incorporate several features that hides this information both from the `Twitsper` server and other undesired users.

Threat model: The components of `Twitsper` are a) Twitter itself, b) user devices, c) the `Twitsper` server, and d) the channel connecting these entities.

We trust Twitter not to leak a person's private information and this has been the premise of our work. We assume that a user's personal device does not compromise a user's privacy; this problem is orthogonal to our work. Thus, the two potential sources of leakage are the `Twitsper` server and the channel. Note here that the `Twitsper` server is the only new addition to the pre-existing Twitter architecture. As discussed earlier, in our supporting server architecture, private content is always posted to Twitter's servers, thus ensuring that this content is not leaked due to the `Twitsper` server. The threat is then the leakage of the metadata associated with private content that may be exposed to the server. Since we administer the `Twitsper` server, we assume that the server will not modify or delete metadata stored on it. Therefore, we focus instead on ensuring that the manner in which metadata is shared with and stored on the `Twitsper` server does not reveal private information either to the server or to undesired users (those not involved in private conversations).

In light of this, we seek to ensure that the following security properties hold:

- An undesired user should not be able to infer which of his/her friends are involved in ongoing private conversations.
- The server should not infer the memberships in ongoing conversations, or determine the size of a private group.

We wish to point out here that if the supporting server has no access to user information (which if made available can compromise the privacy of a user by revealing information such as the number of private conversations that the user is involved in), it cannot authenticate the veracity of whisper postings. We recognize that this exposes the `Twitsper` server to a possible DoS attack wherein fraudulent information could be sent to the server. We defer the exploration of defenses against such attacks on the server for future work, and focus here on protecting user privacy from the server.

Use of certificates to avoid over the channel modifications: The `Twitsper` server has an SSL certificate which validates the authenticity of the server. Thus, a secure HTTPS channel can be established with the server, precluding the possibility of over the channel modifications (as with man in the middle attacks).

Protection from undesired users: A curious user who is not privy to a private conversation may wish to trick the `Twitsper` server into disclosing if one of his friends has initiated a private conversation. To do so, the user may try to guess the message IDs associated with whispers posted by the friend, e.g., based on recent tweets posted by that friend. Note that a whisper results in a set of Direct Messages being posted to Twitter, each of which has an associated message ID.

First, we seek to understand if it is easy for a user to carry out such an attack. Towards this, we perform the following experiments wherein we use three accounts (say) Alice, Bob and Charlie. In our first experiment, Alice sends 50 Direct Messages to Charlie. In our second experiment, Alice sends a Direct Message to Charlie, and immediately thereafter Bob follows by sending a Direct Message to Charlie; we repeat this sequence 50 times. In our final experiment, Alice sends a Direct Message to Charlie and follows that message with a tweet, whereupon Bob does the same. Again, we repeat this sequence 50 times. We observe that while the ID space of tweets and Direct Messages grows monotonically (across both), the gap between the IDs in any pair of posts (sent in quick succession) was *at least* 10^7. We observe no visible pattern using which a user can guess the ID for a Direct Message posted by a friend based on either a recent tweet or Direct Message posted by that friend. While this experiment does indicate that it is hard for an undesired user to query the `Twitsper` server and obtain information with regards to specific private conversations, it does not completely rule out the possibility. Thus, we incorporate the following into our design.

Recall that an initiator of a private conversation sends Direct Messages to a private group, and then seeks to create a mapping on the supporting server between the identifiers for those messages

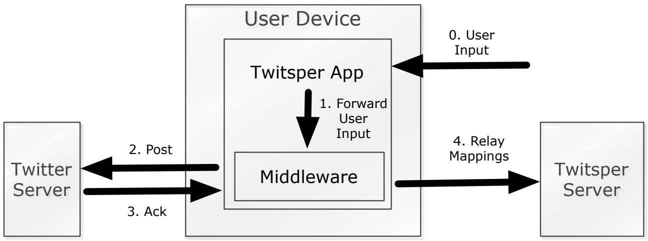
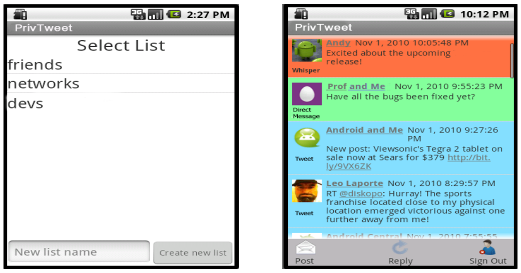


Figure 3: Steps for posting a whisper



PrivTweet

2:27 PM

Select List

friends

networks

devs

New list name

Create new list

(a) List selection

PrivTweet

10:12 PM

Andy Nov 1, 2010 10:05:48 PM
Excited about the upcoming release!
Whisper

Prof and Me Nov 1, 2010 9:55:23 PM
Have all the bugs been fixed yet?
Direct Message

Android and Me Nov 1, 2010 9:27:26 PM
New post: Viewsonic's Tegra 2 tablet on sale now at Sears for $379 http://bit.ly/9VX6ZK
Tweet

Leo Laporte Nov 1, 2010 8:29:57 PM
RT @diskopo: Hurray! The sports franchise located close to my physical location emerged victorious against one further away from me!
Tweet

Android Central Nov 1, 2010 7:55:55

(b) A user's timeline

Figure 4: `Twitsper` on Android OS

and the recipient list. Instead of storing this message ID to list ID mapping on the `Twitsper` server simply as $(whisperID, listID)$ tuples, where $whisperID$ is the message identifier assigned by Twitter, we replace the first component in this tuple with the SHA-512 hash value of $(whisperID|userID|text)$. Here, $userID$ corresponds to a receiver of the whisper and $text$ corresponds to the actual content in the message. This way of storing the mappings on the server has two benefits. First, since the hash function is non-invertible, the server cannot infer the identity of the user involved (the $text$ input to the hash function is only known to the group members and thus, not available to the server). Second, even if an undesired user guesses the IDs of the posted messages, he cannot retrieve the desired mapping, again because he does not know the $text$ provided as input to the hash.

Hiding the entries in a list: The list identifiers included in the mappings stored at the `Twitsper` server can however reveal the participants of private conversations to the server. To hide this information, we encrypt the list ID stored in any tuple with a group key. Clearly, the group key should be available to all of the participants themselves but not to the server. Thus, we have all recipients derive a group key K_g from the content of the received Direct Message, which is not exposed to the `Twitsper` server. Since a user may be involved in multiple groups, the private conversation with which a particular received Direct Message is associated may not always be apparent. Therefore, we associate a new group key K_g with every whisper rather than with every conversation. The key K_g for a particular whisper is a function of the associated text and the sender of the whisper encrypts the list ID with K_g before posting the associated mappings to the `Twitsper` server. Finally, though this can impact the availability of metadata, to keep storage costs at the `Twitsper` server low, we purge entries after a pre-specified time interval (days).

Alternatively, we could use a one-to-many or many-to-many stateless broadcast encryption scheme [32, 38, 27, 25], which ensures that re-keying is infrequent and that many possible subsets can be generated with little computational effort. At this point, we did not see any direct advantage of using such approaches over simply deriving the group key for a conversation from the content of the initial Direct Message in that conversation.

Note that, in the rare case where a user has a single list on Twitter, anyone who knows that the user is using `Twitsper` can infer the set of users with whom the user is having private conversations. In practice, we expect that users will conduct private conversations with different groups at different times, and thus maintain multiple lists on Twitter.

Preventing the inference of group sizes: Even though list IDs are now encrypted, the `Twitsper` server can infer the sizes of private groups simply by counting the number of tuples with the same encrypted list identifier. Recall that the list ID is associated with a hash value that is unique to each intended group participant; thus, if there are K participants, there would be K entries corresponding to the same list. Alternatively, it can simply count the number of tuples written by a single client (the initiator) via its HTTPS connection within a short time frame.

To ensure that the $listID$ in its encrypted form cannot be directly used (via counting) to infer the group size, we store entries of the form $enc_{K_g}(listID|hash(listID)|whisperID)$. The $whisperID$ corresponds to the Direct Message sent to a specific receiver, and thus, each entry now has a unique encrypted list ID associated with it; the `Twitsper` server cannot infer group sizes simply by counting tuples with the same second component. It is easy to see that when the entries are sent to users, the client program can decrypt the content and extract the $listID$.

To preclude the server from inferring the group size by counting the number of tuples written by a client within a short time span, we take the following approach. First, note that simply having clients write dummy tuples to the server does not suffice. The server can infer which tuples are spurious by noting the tuples that are never queried. Thus, we associate each entry with a counter value n which can vary from 1 to M, where M is a random value chosen uniquely for each recipient (note that in many cases $M = 1$). We then modify the first and second components of every tuple to be $hash(n|whisperID|userID|text)$ and $enc_{K_g}(n|M|listID|hash(listID)|whisperID)$. For each recipient (say Bob), Alice creates M entries, M being specific to Bob. Of these, as may be evident, $M - 1$ entries correspond to dummy entries. When Bob queries the server for the first time (with $hash(1|whisperID|userID|text)$), he retrieves the value of M and now knows how many spurious entries are stored for him. His client software then sends $M - 1$ additional requests to retrieve the spurious entries.

Our design has several other desirable security properties, that we discuss briefly here.

- **Preventing leakage of the browsing habits of users:** Since the user ID is never directly revealed to the supporting server, the browsing habits or Twitter access patterns of users are held confidential from the server.
- **CCA security:** Our encryption scheme is based on AES (Advanced encryption standard) [5] which ensures CCA (chosen cipher text attack) security. Thus, even with the rather predictable and simple counters used, the list IDs cannot be reverted.
- **Forward and backward secrecy:** Since a new group key is generated per whisper message, even if someone guesses or uncovers the key for the metadata for a specific message, it does not uncover past or future messages both in the same, or in different conversations. This ensures both forward and backward secrecy.

API Call 1 PrivSend(msg,listID)

```
1:  SALT ← First 8 bytes of SHA-512($msg$)
2:  PASS ← $msg$ concatenated with sender's ID
3:  $K_g$ ← PBKDF2(PASS, SALT)
4:  for each User $U$ in group $listID$ do
5:      $msgID \leftarrow messageID$ returned by Twitter on successful post
6:      $M \leftarrow$ select a random number
7:      $Entry_a \leftarrow$ SHA-512($1|msgID|U|msg$)
8:      $Entry_b \leftarrow encrypt_{K_g}(1|M|listID|hash(listID)|msgID)$
9:      EntryList ← add ($Entry_a$, $Entry_b$)
10:     for $i \in [2, M]$ do
11:         $Dummy_a^i \leftarrow$ SHA-512($i|msgID|U|msg$)
12:         $Dummy_b^i \leftarrow encrypt_{K_g}(i|M|listID|hash(listID)|msgID)$
13:         EntryList ← add ($Dummy_a^i$,$Dummy_b^i$)
14:     end for
15: end for
16: for each (a,b) in EntryList do
17:     send (a,b) to Twitsper server
18: end for
```

API Call 2 isPriv?(msg)

```
1:  $msgID \leftarrow$ Twitter ID for $msg$
2:  $Entry_a \leftarrow$ SHA-512($1|msgID|$ self's ID $|msg$)
3:  $response \leftarrow$ query Twitsper server for $Entry_a$
4:  if $response \neq null$ then
5:      SALT ← First 8 bytes of SHA-512($msg$)
6:      PASS ← $msg$ concatenated with sender's ID
7:      $K_g$ ← PBKDF2(PASS, SALT)
8:      Decrypt $response$ using $K_g$ and cache embedded $listID$ with $msgID$
        for future replies
9:      M ← extracted number for spurious queries
10:     for $i \in [2, M]$ do
11:         $Dummy_a^i \leftarrow$ SHA-512($i|msgID|$ self's ID $|msg$)
12:         $response \leftarrow$ query Twitsper server for $Dummy_a^i$
13:     end for
14:     return TRUE
15: else
16:     return FALSE
17: end if
```

API Call 3 PrivReply(msg,orig_msg)

```
1:  if ID for orig_msg is not in cache then
2:      Reply with a direct message
3:      return
4:  end if
5:  $listID \leftarrow$ mapping for orig_msg's ID in cache
6:  $group \leftarrow$ group specified by the list ∩ user's followers
7:  PrivSend($msg$,$group$)
```

Collision of hash entries: Lastly, since things are indexed by the results of a hash function, the collisions of the hash values might seem to be an issue. The secure hash standard [15] states that for a 512 bit hash function (as in our implementation) we need a work factor of approximately 2^{256} entries to produce a collision which we believe leads to a minuscule probability of experiencing collisions. Thus, we ignore hash collisions for now.

7. IMPLEMENTATION

In this section, we describe our implementation of the Twitsper client and server. Given the popularity of mobile Twitter clients, we implement our client on the Android OS [1, 2].

Generic implementation details. Normal tweets (public) and Direct Messages are sent with the Twitsper client as with any other Twitter client today. We implement whispers using Direct Messages as described before. Recall that direct messaging is a one-to-one messaging primitive provided by Twitter. Mappings from Direct Messages to whispers are maintained on our Twitsper server. Instead of describing each API call separately, our description captures their inter-dependencies.

Twitsper's whisper messages are always sent to a group of selected users. The client handles group creation by creating a list of users on Twitter. This list can either be public (its group members are viewable by any user of Twitter) or private for viewing only by its creator.

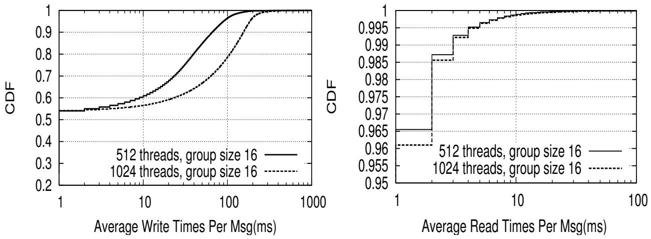
Instantiation of Twitsper API: Figure 3 shows the flow of information involved in posting a whisper. The Twitsper client at the sender first creates a 256 bit AES key from the content to be shared (msg) using the password-based key derivation function (PBKDF2) from PKCS#5 [10]. The input to PBKDF2 is the message text (msg) concatenated with the user ID of the sender. $SALT$ is a random number generated from the content string; in our implementation we simply use the first 8 bytes of the hash value SHA-512(msg). At the end of these steps, the sender has generated the group key (K_g) for the communication (API Call 1; Lines 1–3). The client then sends a Direct Message via Twitter to each group member, whereupon Twitter returns the message IDs for each recipient (API Call 1; Line 5).

The Twitsper client then creates metadata tuples that will enable recipients of the whisper to map Direct Messages to the corresponding list ID (API Call 1; Lines 6–9). Note here that the client also picks a random number M for every recipient and creates $M - 1$ dummy metadata entries on the Twitsper server (API Call 1; Lines 10–14) as discussed before. All of these metadata tuples are finally transmitted to the Twitsper server (API Call 1; Lines 16–18). As discussed earlier, in order to associate a whisper with the correct list, new metadata is created for every Direct Message sent and K_g is newly generated for every posted whisper.

When the Twitsper client program at a recipient receives a Direct Message, it queries the Twitsper server to check whether the message is a whisper or a standard Direct Message (API Call 2). To do so, it first computes the SHA-512 hash from the content in the Direct message and its own user ID (API Call 2; Line 2). If the server finds a match for the query string, it returns the corresponding tuple to the recipient client program; else it sends a null response. If an appropriate (non-null) response is received from the server, the Twitsper client of the recipient extracts the list ID embedded in the tuple. To decrypt the metadata entry, the client generates the group key K_g using the text in the received Direct Message (msg) and the sender's ID (API Call 2; Lines 5–8). The client also extracts the embedded value of M and sends $M - 1$ additional requests for the spurious entries added for this particular recipient (API Call 2; Lines 9–13).

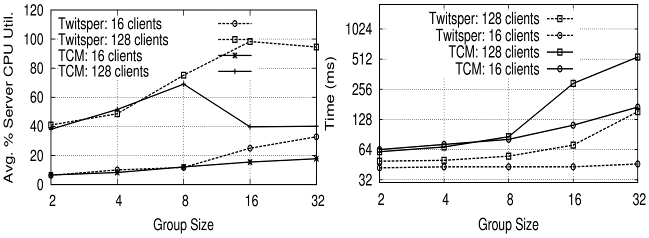
A key feature of our system is that since whispers are sent as Direct Messages, whispers can still be received and viewed by legacy users of Twitter who have not adopted Twitsper; such users cannot however reply to whispers (API Call 3). Twitsper allows a whisper recipient to reply not only to the sender, but also to a *subset* of the original group (specified by the retrieved list) receiving the whisper. This subset is simply the intersection of the original group and the *followers* of the responding user (API Call 3; Line 6). Thus, it respects the social relations established by users.

Finally, we point out that with the list ID corresponding to the group, the client can retrieve the user IDs on that list from Twitter if the original whisper sender has made the list public. If the list is private, the recipient's response can only be received by the original sender. In the future, we plan to permit Twitsper users to modify the list associated with a particular whisper in order to enable inclusion of new users in the private group communication or removal of recipients of the original whisper from future replies; this can be easily done by adding/removing entries on Twitter lists.



(a) DB write time (b) DB read time

Figure 5: Database performance



(a) CPU utilization (b) Service time per client

Figure 6: Server Metrics

Server implementation details: Our server is equipped with an Intel quad-core Nehalem processor, 24 GB of RAM, and one 7200 RPM 1 TB hard disk drive. The `Twitsper` server is implemented as a multi-threaded Java program. The main thread accepts incoming connections and assigns a worker thread, chosen from a thread pool, to service each valid API call. The server stores whisper mappings in a MySQL database. In order to ensure that writing to the database does not become a bottleneck we have multiple connections to the database; we observed that without this, the server performance was affected. These connections are used by worker threads in a round-robin schedule. Note that our server does not store any personal information or credentials of any user. The flow of information in case of a tweet (public) or a Direct Message remains unchanged. Only in the case of a whisper does the use of our system become necessary. The contents of a whisper are never sent to our server; only encrypted metadata is sent as discussed earlier. This ensures that the server can never "overhear" conversations between users or derive user-specific information unless it has either a user's password, which, with `Twitsper`, is never transmitted.

Client implementation details: Our client was written for Android OS v1.6 and was tested on the Android emulator as well as on three types of Android phones (Android G1 dev, Motorola Droid X, and HTC Hero). We use the freely available twitter4j package to access the Twitter API. The client is also multi-threaded and separates the UI (user-interface) thread from the processing, the network, and disk I/O threads. This ensures a seamless experience to the user without causing the screen to "freeze" when the client performs disk or network I/O. We profiled the power consumption of our implementation to identify inefficiencies and iteratively improved the relevant code. These iterative refinements helped us decrease the dependence on the network by caching frequently retrieved user profile images, while maintaining a thread pool rather than the fork and forget model adopted by most open source implementations of other Twitter clients, so as to not over-commit resources.

When the `Twitsper` server is unavailable, we cache whisper mappings on the client and piggyback this data with future interactions with the server. On the other hand, recipients of whispers interpret them as Direct Messages and cannot reply back to the group until the server is again reachable. In future versions of `Twitsper`, we will enable recipients to directly query the client of the original sender if `Twitsper`'s server is unavailable.

We color code tweets, Direct Messages and whispers, while maintaining a simple and interactive UI. Example screen shots from our `Twitsper` client are shown in Figure 4. Our client application is freely available on the Android market, and to date, our `Twitsper` Android application has been downloaded by over 1000 users.

8. EVALUATION

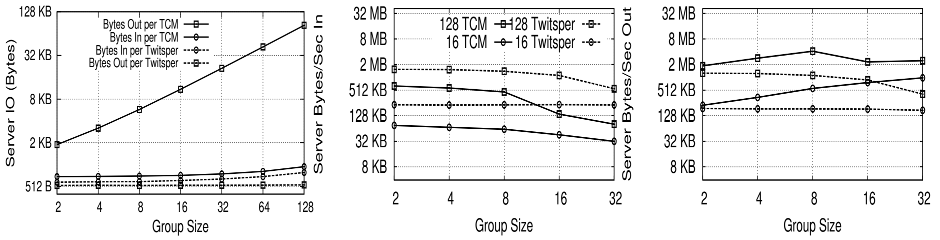
Next we present our evaluation of `Twitsper`. For the purposes of benchmarking, we also implement a version of `Twitsper` wherein a client posts a whisper by transmitting the message to the `Twitsper` server, which in turn posts Direct Messages to all the recipients on the client's behalf. Though, as previously acknowledged, this design clearly violates our design goal of users not having to trust `Twitsper`'s server, we use this *thin client* model (TCM) (we refer to our default implementation as the *fat client* model or `Twitsper` itself) as a benchmark to compare against. One primary motivation for using TCM as a point of comparison is that it can reduce the power consumption on phones (since battery drainage is a key issue on these devices). We also compare `Twitsper`'s energy consumption on a smartphone with that of a popular Twitter client to demonstrate its energy thriftiness.

Server-side results: First, we stress test our server by increasing the rate of connections it has to handle. In this experiment, we use one or more clients to establish connections and send dummy metadata to our server. All clients and the server were on the same local network and thus, network bandwidth was not the constraining factor. We monitored CPU utilization, disk I/O, and network bandwidth with Ganglia [6] and iostat to detect bottlenecks. We vary the target group size of whispers as well as the number of simultaneous connections to the server.

Disk. In Figure 5b, we plot the time taken by each thread to read information relevant to a message from the database (we preloaded the database with 10 million entries to emulate server state after widespread adoption); Figure 5a depicts the CDFs of the write times to the database. We see that as the number of clients increase, so do the database write times, but not the read times. Thus, as the system scales, the bottleneck is likely going to be the I/O for writing to the disk.

CPU. Next, we compare the server performance of TCM and `Twitsper`. We will refer to the version of the server which works in tandem with `Twitsper`, and handles only whisper metadata, as the `Twitsper` server. The TCM server must, in addition, handle the actual sending of whispers to their recipients. It is to be expected that the overhead of the TCM server would increase the computational power needed to service each client. Figures 6a and 6b show the average CPU utilization and user service time, respectively, for each server version. We see in Figure 6a that the `Twitsper` server has a higher CPU utilization than the TCM server. This is because the TCM server spends more idle time (Figure 6b) while servicing each client since it needs to wait on communications with Twitter. So even though more CPU resources are being spent per client with the TCM server, the average CPU utilization is lower.

Another interesting feature noted from these graphs is that certain increases in group size cause the server to more than double its service time. These sharp increases in service time in Figure 6b have corresponding drops in CPU utilization in Figure 6a. This is due to our server's disk writes being the throughput bottleneck. Since in each test we either double the number of client connections or the group size, we would expect a CPU bottleneck to manifest itself with drastic service time increases (of $\approx 200\%$). Instead, the



(a) Bytes in/out per client (b) Incoming bandwidth (c) Outgoing bandwidth

Figure 7: Network activity on server; same legend on (b) and (c)

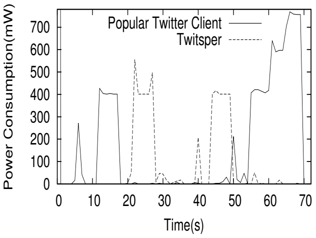


Figure 8: Comparison of power consumption

Interface	`Twitsper`	Other
LCD	13325	10127
CPU	755	1281
3G	4812	8232

Figure 9: Total energy consumption (mJ)

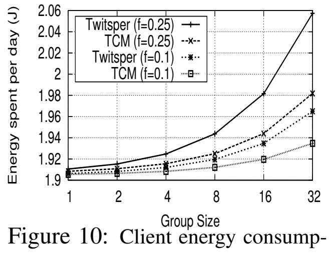


Figure 10: Client energy consumption

data points to a disk write bottleneck where the client must wait for an acknowledgment of the server database's successful write. We verify with iostat that our hard drive is used at 100% utilization during these periods. We are currently investigating the effect of adding more disks.

Network. Figure 7a shows the number of bytes in and out with the TCM and `Twitsper` servers for a single client connection. Each line in Figure 7a represents a single client sending one whisper message to a group size which is varied (x-axis). We see that increasing the group size does not cause a large increase in the received bytes as compared to the case with only 2 group members. This illustrates that the overhead increase with recipient group size (which causes either the receipt of more message IDs with the `Twitsper` server or the receipt of more recipient user IDs with the TCM server) is very minor when compared to the resources consumed by the SSL connection between the client and the server. The only additional overhead with the TCM server is the transfer of the actual whisper messages from the client; this manifests as the constant offset between these two curves. Since the `Twitsper` server has to only send a confirmation to the user that its whisper meta data was received correctly, the bytes out is independent of the recipient group size (all meta data corresponding to a whisper is sent as a single atomic block). In contrast, the burden of having to send whispers to each recipient (as a separate Direct Message) is on the TCM server. Increasing group size (x-axis) increases the number of Direct Messages sent to Twitter and this quickly results in an overshoot of the single client SSL connection overhead.

Figures 7b and 7c show the bandwidth consumed at the server as the number of bytes in and out per second. In Figure 7b, we see that the `Twitsper` server does not experience a reduction in transmission rate until it hits 128 clients and a group size of 16. At this point, we hit a disk bottleneck in writing client message metadata to our database. For the TCM server, we see a rate reduction even in the 16 clients case as we increase the group size; this is due to the latency incurred in the message exchange with Twitter. We hit a similar hard disk bottleneck at 128 concurrent client connections with the TCM server, as similar metadata needs to be stored with both server setups.

Comparing `Twitsper` and TCM clients: While `Twitsper` offers higher CPU utilization as well as lower bandwidth requirements, the energy (power$*$time) consumption at the client is a key factor in ensuring adoption of the service. To evaluate its client side energy performance, we measure *the amount of energy* needed to make a single post with `Twitsper` to Twitter and to send a message to our server. We also use the PowerTutor [12] application to measure the power consumed at the client. We made 100 posts back to back and measure the average energy consumed.

Figure 10 compares TCM and `Twitsper` based on the energy consumed on a phone. The figure shows the energy consumption per day on an Android phone, for an average Twitter user who sends 10 messages per day and has 200 followers [9]. Our experiments suggest that the best implementation depends on the fraction of a user's messages that are private (denoted by f) and the typical size of a list to which private messages are posted. The energy consumption with `Twitsper` is significantly greater than that with the TCM client when f is large or the group sizes are big. However, since we expect private postings to constitute a small fraction of all information sharing and that such communication will typically be restricted to small groups, energy consumption overhead with `Twitsper` is minimal. Even in the scenarios where client-side energy consumption increases, the energy consumed is still within reason, e.g., the energy consumed per client across various scenarios is within the range of 1.9 J to 2.5 J, which is less than 0.005% of the energy capacity of typical batteries (10 KJ, as shown in [12]). Further, as we show next, the majority of the energy consumed in practice is by the user's interaction with the phone's display, whereas the energy we consider here is only that required to simply send messages, and does not include displaying and drawing graphics on the screen.

Comparison with another popular Twitter client: We next compare the power consumption of `Twitsper` with that of a popular Twitter client (TweetCaster[18]), which supports the default privacy options on Twitter. We begin the test after both clients had been initialized and had run for 15 seconds. We then send a message from each of the clients and refresh the home screen; there was at least one update to the home screen. As seen from the traces of the power consumed in Figure 8, `Twitsper`'s power consumption is comparable. This shows that `Twitsper` only imposes energy requirements on the mobile device that are comparable to other Twitter clients. We observe that there is no noticeable loss in performance since both clients were made to carry out the same tasks functionally.

In the above test, even though the screen was kept on for as little a time as possible (less than 10% of the total time) the LCD accounted for close to 50% of the aggregate energy consumed, as seen from Figure 9. Referring the reader back to Figure 10, we see that as the group size increases there is only a marginal increase in the energy consumption associated with the sending of messages. Even if 25% of the messages are whispers and the average group size is 32 (which we believe is quite large), the energy consumed only increases from 1.92 J (for a single tweet) to 2.05 J—an increase of less than 15%; given that the LCD power consumption dominates, this is not a significant energy cost.

9. CONCLUSIONS

Today, for users locked in to hugely popular OSNs, the primary hope for improved privacy controls is to coerce OSN providers via the media or via organizations such as EFF and FTC. In this paper, to achieve privacy without explicit OSN support, we design and implement `Twitsper` to enable fine-grained private group messaging on Twitter, while ensuring that Twitter's commercial interests are preserved. By building `Twitsper` as a wrapper around Twitter, we show that it is possible to offer better privacy controls on existing OSNs without waiting for the OSN provider to do so.

Next, we plan to implement fine-grained privacy controls on other OSNs such as Facebook and Google+ as well, using a similar approach of building on the API exported by the OSN. Given the warm feedback received by `Twitsper`, we hope that the adoption of `Twitsper` and its follow-ons for other OSNs will persuade OSN providers themselves to offer fine-grained privacy controls to their users.

References

[1] Android operating system. `http://www.android.com/`.

[2] Comscore: Android is now highest-selling smartphone OS. `http://bit.ly/euR4Yb`.

[3] DiSo project. `http://diso-project.org/`.

[4] Facebook traffic reaches nearly 375 million monthly active users worldwide, led by us. `http://bit.ly/c0Z3UQ`.

[5] Fips 197, adavanced encryption standard. `1.usa.gov/8Y4V6U`.

[6] Ganglia. `http://ganglia.sourceforge.net/`.

[7] Google Plus numbers belie social struggles. `http://bit.ly/pPIwDr`.

[8] Grouptweet. `http://www.grouptweet.com/`.

[9] New data on Twitter's users and engagement. `http://bit.ly/cu8P2s`.

[10] PKCS 5: Password-based cryptography specification version 2.0. `http://tools.ietf.org/html/rfc2898`.

[11] Please rob me. `http://www.pleaserobme.com/`.

[12] Powertutor. `http://bit.ly/hVaXh1`.

[13] Priv(ate)ly. `http://priv.ly/`.

[14] Retweet this if you want non-followers replies fixed. `http://bit.ly/YwLYw`.

[15] Secure hash standard. `1.usa.gov/cISXx3`.

[16] Social networks offer a way to narrow the field of friends. `http://nyti.ms/j7dOsC`.

[17] Tweet this milestone: Twitter passes MySpace. `http://on.wsj.com/dc25gK`.

[18] Tweetcaster. `http://tweetcaster.com/`.

[19] Tweetworks. `http://www.tweetworks.com`.

[20] Twitter Groups! `http://jazzychad.net/twgroups/`.

[21] Twitter reveals it has 100m active users. `bit.ly/nJoRuk`.

[22] Twitter suspends twidroyd & UberTwitter over privacy claims. `http://bit.ly/hRcZ1w`.

[23] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: An online social network with user-defined privacy. In *SIGCOMM*, 2009.

[24] F. Beato, M. Kohlweiss, and K. Wouters. *Scramble! Your Social Network Data.* 2011.

[25] D. Boneh and M. Hamburg. Generalized identity based and broadcast encryption schemes. In *ASIACRYPT*, 2008.

[26] S. Buchegger and A. Datta. A case for P2P infrastructure for social networks- opportunities and challenges. In *WONS*, 2009.

[27] Y. Dodis and N. Fazio. Public-key broadcast encryption for stateless receivers. In *ACM Digital Rights Management*, 2002.

[28] G. T. Emiliano De Cristofaro, Claudio Soriente and A. Williams. Hummingbird: Privacy at the time of twitter. Cryptology ePrint Archive, Report 2011/640, 2011. `bit.ly/SYBEzK`.

[29] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[30] D. A. Grier and M. Campbell. A social history of bitnet and listserv, 1985-1991. *IEEE Annals of the History of Computing*, 2000.

[31] S. Guha, K. Tang, and P. Francis. NOYB: Privacy in online social networks. In *WOSN*, 2008.

[32] J. Y. Hwang, D. H. Lee, and J. Lim. Generic transformation for scalable broadcast encryption scheme. In *CRYPTO*, 2005.

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

[34] B. Krishnamurthy and C. Willis. Characterizing privacy in online social networks. In *WOSN*, 2008.

[35] B. Krishnamurthy and C. Willis. On the leakage of personally identifiable information via online social networks. In *WOSN*, 2009.

[36] S. J. Liebowitz and S. E. Margolis. Network externality: An uncommon tragedy. *The Journal of Economic Perspectives*, 1994.

[37] D. Liu, A. Shakimov, R. Caceres, A. Varshavsky, and L. P. Cox. Confidant: Protecting OSN Data without Locking it Up. In *Middleware*, 2011.

[38] D. Lubicz and T. Sirvent. Attribute-based broadcast encryption scheme made efficient. In *AFRICACRYPT*, 2008.

[39] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. In *NDSS*, 2011.

[40] R. A. Popa, H. Balakrishnan, and A. J. Blumberg. VPriv: Protecting privacy in location-based vehicular services. In *USENIX Security Symposium*, 2009.

[41] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for Web transactions. *ACM TISSEC*, 1998.

[42] A. Shakimov, H. Lim, R. CÃąceres, L. P. Cox, K. Li, D. Liu, and A. Varshavsky. Vis-ã-Vis: Privacy-preserving online social networks via virtual individual servers. In *COMSNETS*, 2011.

[43] I. Singh, M. Butkiewicz, H. V. Madhyastha, S. V. Krishnamurthy, and S. Addepalli. Building a wrapper for fine-grained private group messaging on Twitter. In *HotPETS*, 2012.

[44] A. Tootoonchian, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: Better privacy for social networks. In *CoNEXT*, 2009.

[45] C. Wilson, T. Steinbauer, G. Wang, A. Sala, H. Zheng, and B. Y. Zhao. Privacy, availability and economics in the Polaris mobile social network. In *HotMobile*, 2011.

[46] N. Zeldovich, S. B. Wickizer, and D. MaziÃÍres. Securing distributed systems with information flow control. In *NSDI*, 2008.