Building a Wrapper for Fine-Grained Private Group Messaging 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 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 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 from other users the content they share. 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 [30, 24] 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, 37, 4]. On the other hand, though *new OSNs* have been developed with privacy explicitly in mind [15, 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; 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 [16]. 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, without having to wait for Twitter to make any changes to its service.

As our primary 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, 18, 21], `Twitsper` ensures that Twitter's commercial interests are preserved and that users do not need to trust `Twitsper` with their 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 simple approach seemed to be the best fit for fulfilling all our objectives. Thus, our implementation of `Twitsper` 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. Our evaluation demonstrates that this simple design does achieve the best tradeoffs 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 provided 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 [20, 14, 19].

2 Related work

Characterizing privacy leakage in OSNs: Krishnamurthy and Willis characterize the information that users reveal on OSNs [31] and how this information leaks [32] to other entities on the web (such as social application providers and advertising agencies). Our thesis is that ensuring OSN providers do not leak user information requires legal measures that mandate appropriate controls. 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 users to restrict sharing of their messages either to only all of their followers (by setting their account to *private* mode) or to exactly one of their 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 [37], DiSo [4], and PeerSoN [26]). These systems require a user to store her data on a personal device or in the cloud, thus removing the need for the user to trust a central

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.

OSN provider. However, users have put in tremendous effort in building their social connections on today's OSNs [5, 16]; 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 [38], 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 [30, 24, 25] or via out-of-band servers [12]. Users can share the decryption keys with a select subset of their connections (friends). Hummingbird [27] 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. [34] 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.

Group communication: Like `Twitsper`, listserv [29] 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 GroupTweet [8], Tweetworks [18], and Twitter Groups [21], 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.

3 Motivating User Survey

While privacy concerns with OSNs have received significant coverage [31, 32], 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 contributed by a user from other users on the OSN, which has began to receive some attention [10]. 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.

---

[1]Participants consisted of staff and students at UCR.

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.

4 System design goals

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 [37, 26, 4]) we want our solution to be compatible with Twitter. This goal stems from the fact that Twitter already has an extremely large user base—over 100 million active users [22]. Since the value of a network grows quadratically with the growth in the number of users on it (the network effect [33]), 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 [23]) or via information flow control [39]. 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 [28] 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., [35]).

Table 2 compares our proposal with previous solutions for improving user privacy on OSNs. Unlike

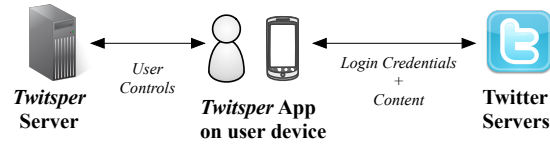| API call	Function
PrivSend(msg, group)	Send *msg* to all users specified in *group*
isPriv?(msg)	Determine if *msg* is a private message
PrivReply(msg, orig_msg)	Send *msg* to all of the user's followers who received *orig_msg*

Figure 1: `Twitsper`'s API beyond normal Twitter functionality.

Figure 2: System architecture.

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 [38], 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 [18] and Twitter Groups [21], we ensure that Twitter is privy to private conversations but `Twitsper` is not.

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-off. As mentioned earlier, surprisingly, a fairly straightforward simple approach seems to yield the best trade-offs 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.

Interface: Our primary goal is to extend Twitter's privacy model. In addition to sharing messages with all followers or precisely one follower, we seek to enable users to privately share messages with a subset of their followers. To do so, we extend Twitter's API with the additional functionality shown in Table 1.

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 communica-

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)	✗	✓	✓	✓	✓	✓

Table 3: Comparison of architectural choices.

tion, it is necessary that 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 this change.

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 by sending a Direct Message to each of those users. To enable replies, a supporting server can maintain the list of the original recipients of a whisper; when a client sends a whisper, it can send the identifiers of the Direct Messages and the list of recipients to the supporting server. Thus, a user can query this supporting server to check if a received Direct Message corresponds to a whisper. When the user chooses to reply to a whisper, the user's client can retrieve the list of recipients of the original whisper from the server, 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 (list of recipients of a whisper) from the original sender. Another option is to have the client embed the list of recipients of 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 smart phones. On the flip side, in this architecture, the content posted by a user is *never* exposed to the supporting server i.e., privacy from `Twitsper`'s server is preserved. The server is simply a facilitator of group communications across a private group and only maintains meta data related to whispers. 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.

Using encryption to hide content: To address the shortcoming in the previous architecture of being unable to link replies to the original whispers, in our next candidate architecture, clients post whispers 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 [36].

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 is used for sending whispers. Since a Direct Message from one user to another is possible only if they are connected, links between these private accounts are dynamically created so as to allow a private message to propagate only to the desired users. However, Twitter has been quite clear that an application that creates significant fluctuation in links between users will be considered detrimental to the Twitter ecosphere [1], thus discouraging the dynamic creation and deletion of associations between private accounts. Further, a supporting server that has the relevant credentials of all users is necessary to setup and authorize follower-followee links between accounts dynamically; requiring users to trust their OSN login credentials with a supporting server is undesirable.

A different method for facilitating selective sharing of information is to use an @*reply* from these private accounts. However, as of mid-2009, Twitter discontinued the "capability" of @*reply* messages between
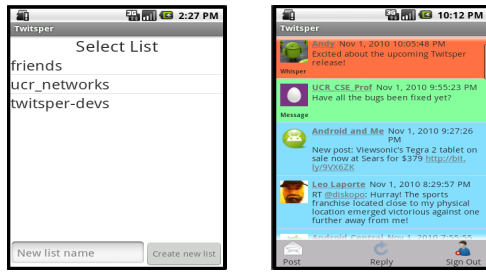
(a) List selection   (b) A user's timeline

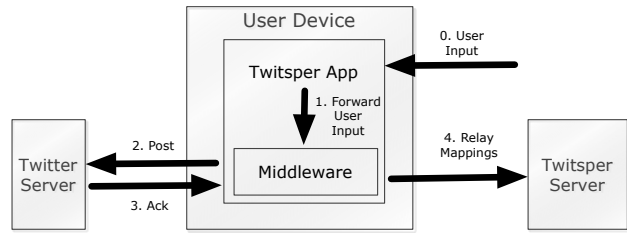Figure 3: `Twitsper` on Android OS

Figure 4: Steps for posting a whisper

disconnected users after concluding that less than 1% of the users found this feature useful and that it contributes to spam messages [13]. 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 3 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`, whose architecture is as shown in Figure 2.

6 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 [2, 3].

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.

`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. The client sends a Direct Message via Twitter to each group member and gathers the message IDs returned by Twitter. These IDs and the Twitter list ID are then sent to the `Twitsper` server. The server creates and stores a mapping of the message IDs to the IDs of the group members.

When a `Twitsper` client receives a Direct Message, it queries the `Twitsper` server to check whether the message is a whisper or a standard Direct Message. If the server finds a mapping from the message ID to a list of users, this indicates that the message corresponds to a whisper. The server reports its finding to the client. 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.

`Twitsper` allows a whisper recipient to reply not only to the sender, but also to a *subset* of the original group receiving the whisper. This subset is simply the intersection of the original group and the *followers* of

the responding user. Thus, it respects the social relations established by users. A whisper recipient's client is provided with the list ID corresponding to the group, and the client can then retrieve the user IDs on that list from Twitter if the original whisper sender 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.

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 unique message IDs for the consequent Direct Messages (along with a list ID) are sent. This ensures that the server can never "overhear" conversations between users unless it has either a user's password,which with `Twitsper` is never transmitted to the server.

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 is performing 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 also 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 3. Our client application is freely available on the Android market, and to date, our `Twitsper` Android application has been downloaded by over 1000 users.

7 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

(a) DB write time   (b) DB read time   (a) CPU utilization   (b) Service time per client
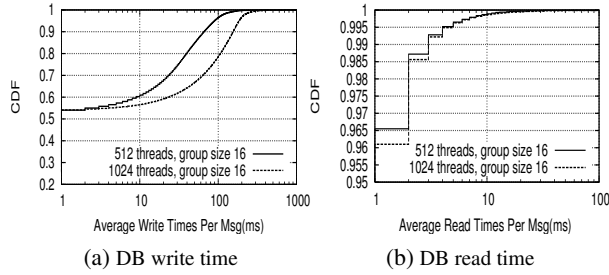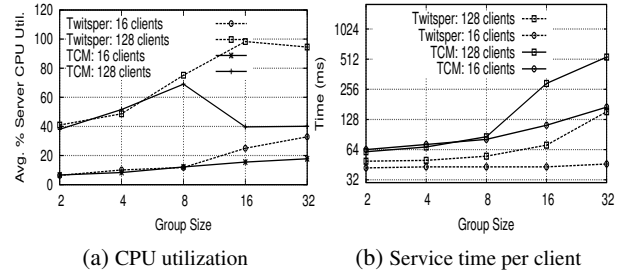
Figure 5: Database performance   Figure 6: Server Metrics

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 smart phone 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 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 9a shows the number of bytes in and out with the TCM and `Twitsper` servers for a single client connection. Each line in Figure 9a 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

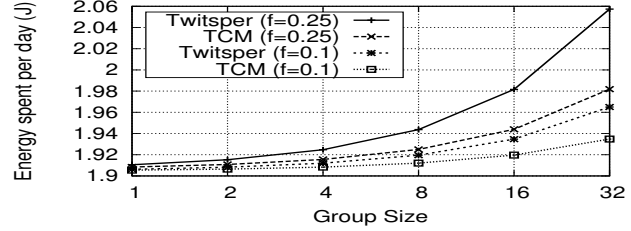| Interface	Twitsper	Other
LCD	13325	10127
CPU	755	1281
3G	4812	8232

Figure 7: Total power consumption (mJ)

Figure 8: Client power consumption

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. 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 9b and 9c show the bandwidth consumed at the server as the number of bytes in and out per second. In Figure 9b, 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 power consumption at the client is a key factor in ensuring adoption of the service. To evaluate its client side power 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 use the PowerTutor [11] application to measure the power consumed at the client. We made 100 posts back to back and measure the average energy consumed.

Figure 8 compares TCM and `Twitsper` based on the power 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 [11]). 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[17]), 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

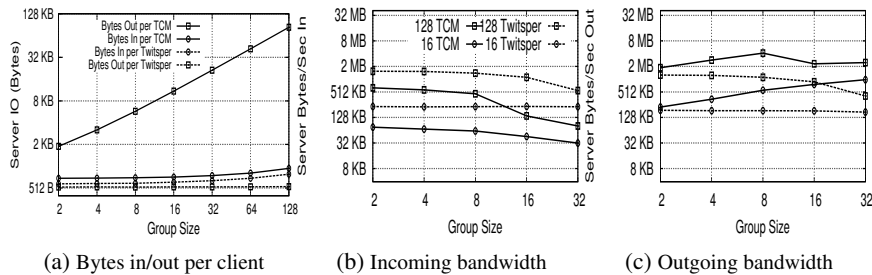(a) Bytes in/out per client (b) Incoming bandwidth (c) Outgoing bandwidth

Figure 9: Network activity on server; same legend on (b) and (c)
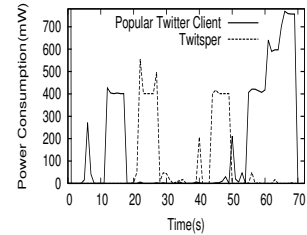
Figure 10: Comparison of power consumption

home screen. As seen from the traces of the power consumed in Figure 10, `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 7. Referring the reader back to Figure 8, 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.

8 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] Aggressive follower churn in spam subsection of "the twitter rules". http://bit.ly/a62bx1.

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

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

[4] DiSo project. http://diso-project.org/.

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

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

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

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

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

[14] Selectively tweeting via free Android application Twitsper. `http://bit.ly/gas5bS`.

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

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

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

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

[19] Twitsper app for Android enhances Twitter security and privacy. `http://bit.ly/avDV8l`.

[20] Twitsper, group tweeting app, could change Twitter as we know it. `http://huff.to/cYkGvH`.

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

[22] Twitter reveals it has 100m active users. `http://www.guardian.co.uk/technology/pda/2011/sep/08/twitter-active-users`.

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

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

[25] F. Beato, M. Kohlweiss, and K. Wouters. *Scramble! Your Social Network Data*, volume 6794 of *Lecture Notes in Computer Science*, chapter 12, pages 211–225. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[26] S. Buchegger and A. Datta. A case for P2P infrastructure for social networks - opportunities and challenges. In *6th International Conference on Wireless On-demand Network Systems and Services (WONS)*, 2009.

[27] E. De Cristofaro, C. Soriente, G. Tsudik, and A. Williams. Hummingbird: Privacy at the time of Twitter. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[28] 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.

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

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

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

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

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

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

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

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

[37] A. Shakimov, H. Lim, R. Cáceres, 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.

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

[39] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *NSDI*, 2008.