UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Improving User Privacy in Emerging Platforms

Permalink
https://escholarship.org/uc/item/64s9r9dm

Author
Singh, Indrajeet

Publication Date
2014

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Improving User Privacy in Emerging Platforms

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Indrajeet Singh

December 2014

Dissertation Committee:

Professor Srikanth Krishnamurthy, Chairperson
Professor Harsha Madhyastha
Professor Eamonn Keogh
Professor Neal Young
Professor Iulian Neamtiu

Copyright by
Indrajeet Singh
2014

The Dissertation of Indrajeet Singh is approved:

Committee Chairperson

University of California, Riverside

To my family and my near and dear ones. Thank you all for the constant support, encouragement, and love throughout my life.

ABSTRACT OF THE DISSERTATION

Improving User Privacy in Emerging Platforms

by

Indrajeet Singh

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2014
Professor Srikanth Krishnamurthy, Chairperson

With the advent of personalized services and devices, like online social networks and smartphones, a user's (otherwise private or sensitive) information is shared with these platforms. While this sharing of information undoubtedly adds value to the user's experience, it raises some pressing concerns about the user's privacy, or the lack of it.

In this dissertation, we propose three major frameworks designed and implemented specifically to protect the users' privacy on such emerging and highly personalized platforms. The three frameworks are as follows.

1. **Twitsper**, a wrapper for Twitter that allows user to exchange private messages without hurting Twitter's commercial interest.

2. **Hermes**, a decentralized social network with architecture the explicit goal of preserving a user's privacy and anonymity (including even their sharing patterns) while minimizing the cost each user incurs due to the decentralized model.

3. **ZapDroid**, a version of Android that is modified and enhanced specifically for the purpose of detecting and negating the impacts of zombie apps. In the context of our work, zombie apps

are unused apps that still run in the background, consuming resources and leaking the users' private information. As an OS, ZapDroid is compatible with any app downloadable from the Google Play Store.

Contents

List of Figures **ix**

List of Tables **x**

1 Introduction **1**

2 Twitsper **5**

2.1 Introduction 5
2.2 Related work 7
2.3 Motivating User Survey 10
2.4 Design objectives 10
2.5 `Twitsper` design 13
2.6 Protecting Privacy 21
2.7 Implementation 27
2.8 Evaluation 31
2.9 Conclusions 36

3 Hermes **41**

3.1 Introduction 41
3.2 Related Work 44
3.3 Goals and Threat Model 45
3.4 Hermes Architecture 47
3.4.1 Hermes Architecture Overview 47
3.4.1.1 Sharing new content 48
3.4.1.2 Enabling OSN-like conversations 49
3.5 Hiding users' sharing patterns 52
3.5.1 Hiding the membership information within each private conversation: . . . 52
3.5.1.1 Strawman approach 52
3.5.1.2 Obfuscating group size 53
3.5.1.3 Preventing inference of group membership based on comments . 54
3.5.1.4 Selecting the length of a round 55
3.5.2 Hiding users' conversation patterns by handling intersection attacks: 57

3.5.2.1 Tuning the membership of $\mathbb{D}$ 59
3.5.2.2 Using fake conversations 60
3.6 Other Security properties of Hermes 61
3.7 Quantifying Cost, Anonymity, and Timeliness Trade-offs 63
3.8 Prototype Implementation 71
3.9 System Evaluation 74
3.10 Miscellaneous Issues 76
3.11 Conclusions 78
3.12 Background on ECDH 79
3.13 Propagating updates via $ufiles$ 79

4 ZapDroid **83**
4.1 Introduction 83
4.2 Understanding Zombie Apps 87
4.2.1 User Studies 87
4.2.2 Offline Measurement Methodology 89
4.2.3 Measurement Results and Inferences 93
4.3 An Overview of ZapDroid 97
4.4 Identifying and Profiling Zombie Apps 101
4.5 Quarantining Zombie Apps 107
4.6 Restoring Zombie Apps 110
4.7 Evaluation 111
4.8 Discussion 117
4.9 Related Work 119
4.10 Conclusions 122

Bibliography **124**

List of Figures

2.1 Comparison of architectural choices. 17
2.2 System architecture using supporting server. 18
2.3 Steps for posting a whisper 24
2.4 `Twitsper` on Android OS 24
2.5 Database performance 30
2.6 Server Metrics 30
2.7 Network activity on server; same legend on (b) and (c) 32
2.8 Comparison of power consumption 32
2.9 Total energy consumption (mJ) 35
2.10 Client energy consumption 35

3.1 Illustration of conversation timeline. 48
3.2 Round structure in *Hermes*. 57
3.3 Analyses with Facebook data 65
3.4 Anonymity trade-offs per conversation 65
3.5 Optional caption for list of figures 69
3.6 End-to-end delays on *Hermes* and on Facebook 75
3.7 Illustration of comment propagation in *Hermes*. 80

4.1 App statistics from the user study. 90
4.2 Network transfers by zombie apps. 91
4.3 Energy consumption by zombie apps. 93
4.4 Permissions used by zombie apps. 96
4.5 Storage consumed by zombie apps. 96
4.6 High-level architecture of *ZapDroid*. 100
4.7 Modifications made to the Android OS. 102
4.8 *ZapDroid*'s modules. 103
4.9 Comparison of IPC mechanisms. 112
4.10 Permission Service overhead. 112
4.11 Quarantine overhead 113
4.12 Restoration overhead 113
4.13 Comparing *ZapDroid* with other solutions 113

List of Tables

2.1 Results of survey about privacy shortcomings on Twitter. 11
2.2 Comparison of `Twitsper` with previous proposals for improving user privacy on OSNs. 13
2.3 `Twitsper`'s API beyond normal Twitter functionality. 14

3.1 Cost for various values of A_1 71
3.2 *Hermes*'s resource consumption on GAE 71

4.1 Types of apps in the top 5 categories. 89

Chapter 1

Introduction

Though little time has passed since their emergence, smartphones and Online Social Networks (OSNs) have become the major communication platforms for many. Yet, their growing pervasiveness has exposed and exacerbated a phenomenon that is just as new and never considered before: the ever shrinking ability of a person to preserve their privacy. In the context of this dissertation, privacy is the ability of a person to decide when, how, and where they may disclose a piece of personal or sensitive information like healthcare information, credit card numbers, or location history. The goal of this dissertation two-fold: (a) discuss how a person's privacy is at risk due to the advent of those emerging platforms and (b) present three systems that are designed and implemented specifically to address and mitigate the aforementioned risks.

The very first system is Twitsper, the purpose of which is to enable a user to exchange private messages with another person or a group on Twitter. 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.

Twitsper, our main contribution 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 [32, 61, 62], 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).

The second system proposed in this dissertation is Hermes, a decentralized OSN architecture designed explicitly with the goal of hiding sharing patterns while minimizing users costs. Today, a user has to implicitly trust an OSN with their personal information [22]. Leakage of information from OSN servers [21, 20], coupled with the need for OSN providers to mine user data (e.g., for profit via targeted advertisements), have raised many concerns from privacy-conscious users [56]. Though there have been prior efforts that seek to enable private communications on OSNs, we argue that all of these have limitations that either hinder their applicability in practice, or simply result in other privacy leaks that are not adequately dealt with.

Hermess key goal is to ensure that any content shared by a user as well her sharing habits are kept private from both the OSN provider and undesired friends. In doing so, Hermes seeks to (i) minimize the costs borne by users, and (ii) preserve the interactive and chronologically consistent conversational structure offered by a centralized OSN.

Hermes uses three key techniques to meet these goals. First, it judiciously combines the use of compute and storage resources in the cloud to bootstrap conversations associated with newly shared content. This also supports the high availability of the content. Second, it employs a novel cost-effective message propagation mechanism to enable dissemination of comments in a timely and consistent manner. It identifies and purges (from cloud storage) content that has been accessed by all intended recipients. Lastly, but most importantly, Hermes carefully orchestrates how fake postings are included in order to hide sharing patterns from the untrusted cloud providers used to store and propagate content, while minimizing the additional costs incurred in doing so. A key feature of Hermes is its flexibility in deployment where it can either be implemented as a stand alone distributed OSN or as an add-on to todays OSNs like Facebook (while maintaining the decentralized nature of content sharing). The latter option is especially attractive for potentially quick and widespread deployment.

The third and last work proposed in this dissertation is ZapDroid, a system that enables users to detect and appropriately silo Android applications (apps) that, even though they are not being used by the user, sit quietly in the background, consume resources, and sometimes even disclose the user's private information. Statistics indicate that for a typical app, less than half of the people who downloaded it use it more than once [18] and 15% of the users never delete a single app that they download [54]. These apps continue to operate in the background long after the user

has stopped interacting with them. Such background activities have significant negative impacts on the user, e.g., leaking private information or significantly taxing resources such as the battery or network. We call such apps, which are dead from the perspective of the user but indulge in undesired activities, as “zombie apps”.

The four key contributions that we make with the ZapDroid project are as follows. First, is a method to efficiently track the state of an app in the foreground and in the background to detect when an app has become a zombie app. Prior approaches [132] relies on continuous monitoring of apps and can be too resource-intensive to be practical. Second, a lightweight addition to the Android kernel to monitor how an app is accessing sensitive informations that are protected by Android permissions. This is not trivial because Android itself does not allow a mechanism implemented at the application level to track another application. Third, safeguards put in place to make sure a quarantined app cannot reactivate, which is an important point to make because previous work only manage to constrain the app’s activity temporarily [68] before it is waken up by timeouts or external stimuli [10, 28]. The fourth contribution is a method to restore the previously quarantined app quickly and to the exact same state pre-quarantine if the user wants to reuse the app. This last point is crucial because although a user can always reinstall from the Google Play Store, a reinstall is not optimal under limited connectivity and it may not be able to preserve the app’s state before it is quarantined.

Chapter 2

Twitsper

2.1 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 [95, 72] 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 [79, 120, 19]. On the other hand, though *new OSNs* have

been developed with privacy explicitly in mind [55, 30], 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 [59]. 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 work, 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 [32, 61, 62], `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 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.

2.2 Related work

Characterizing privacy leakage in OSNs: Krishnamurthy and Willis characterize the information that users reveal on OSNs [103] and how this information leaks [104] 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 [120], Confidant [108], DECENT [100], Polaris [128], and PeerSoN [79]). 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 [23, 59], 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 [125], 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 [95, 72, 74] or via out-of-band servers [47]. Users can share the decryption keys with a select subset of their connections (friends). Hummingbird [88] 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. [112] 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 [94] 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 [62], GroupTweet [32], and Tweetworks [61], 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 [121] 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.

2.3 Motivating User Survey

While privacy concerns with OSNs have received significant coverage [103, 104], 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 [45].

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 2.1 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.

2.4 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

[1] Participant details removed for anonymity reasons

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 2.1: Results of survey about privacy shortcomings on Twitter.

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 [120, 79, 19]), 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 [63]. Since the value of a network grows quadratically with the growth in the number of users on it (the network effect [107]), 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 [64]) or via information flow control [130]. 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 [89] 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., [117]).

Table 2.2 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 com-

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 2.2: Comparison of `Twitsper` with previous proposals for improving user privacy on OSNs.

mercial interests. Moreover, we introduce user controls via `Twitsper` without adding another entity for users to trust, unlike proposals such as Lockr [125], 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 [32], Tweetworks [61], and Twitter Groups [62], we ensure that Twitter is privy to private conversations but `Twitsper` is not.

2.5 `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.

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 2.3: `Twitsper`'s API beyond normal Twitter functionality.

- ***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 2.3. 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

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

Figure 2.1: Comparison of architectural choices.

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

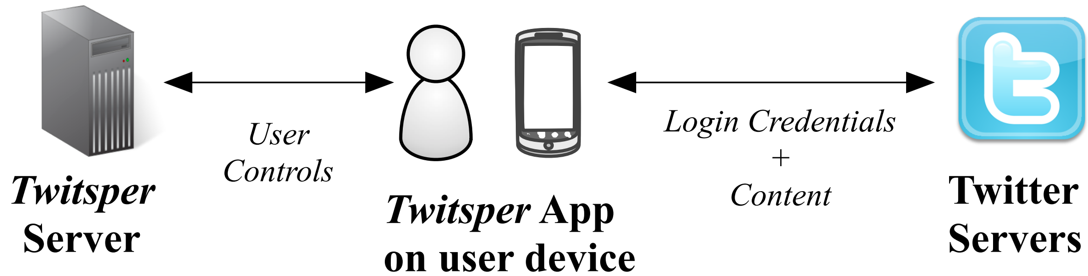


Figure 2.2: System architecture using supporting server.

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 2.6 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 [119].

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 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 [49]. 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 2.1 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 2.2.

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.

2.6 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 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 $text$ cor-

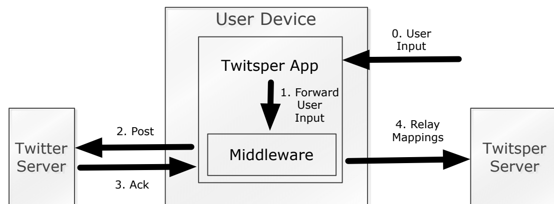
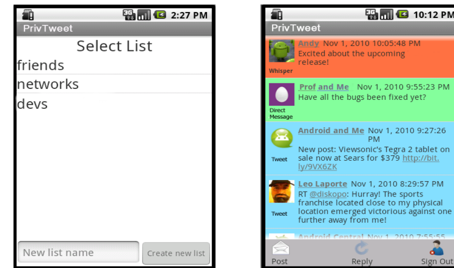


Figure 2.3: Steps for posting a whisper



(a) List selection

(b) A user's timeline

Figure 2.4: `Twitsper` on Android OS

responds 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 [99, 109, 86, 78], 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) [24] 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.

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 [52] 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.

2.7 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 [9, 16].

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 2.3 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 [44]. 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.

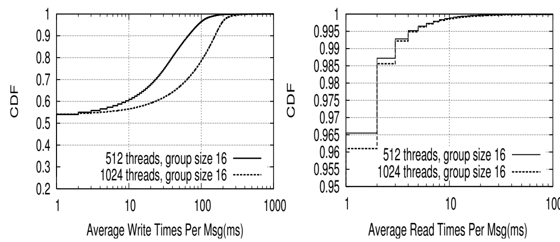
When the `Twitsper` client program at a recipient receives a Direct Message, it queries the `Twitsper` server to check whether the message is a whisper or a standard Direct Message (API Call 2). To do so, it first computes the SHA-512 hash from the content in the Direct message and its own user ID (API 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.

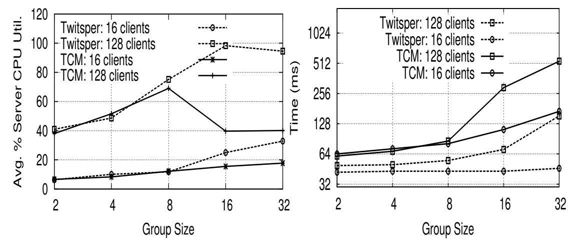
Finally, we point out that with the list ID corresponding to the group, the client can retrieve the user IDs on that list from Twitter if the original whisper sender has made the list public. If the list is private, the recipient's response can only be received by the original sender. In the future, 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.

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



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

Figure 2.5: Database performance



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

Figure 2.6: Server Metrics

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.

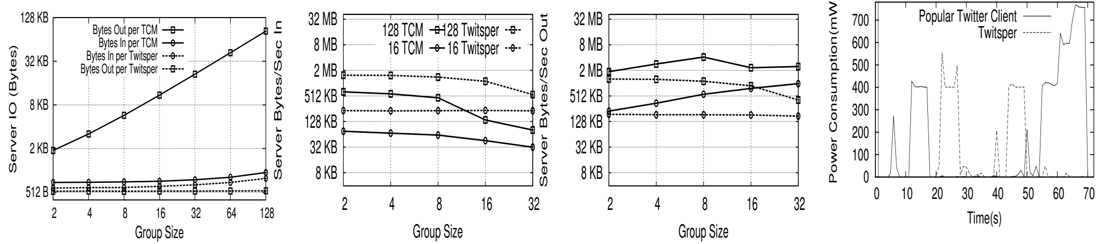
When the `Twitsper` server is unavailable, we cache whisper mappings on the client and piggyback this data with future interactions with the server. On the other hand, recipients of whispers interpret them as Direct Messages and cannot reply back to the group until the server is again reachable. In future 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 2.4. Our client application is freely available on the Android market, and to date, our `Twitsper` Android application has been downloaded by over 1000 users.

2.8 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



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

Figure 2.7: Network activity on server; same legend on (b) and (c)

Figure 2.8: Comparison of power consumption

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 [25] 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 2.5b, 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 2.5a 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 2.6a and 2.6b show the average CPU utilization and user service time, respectively, for each server version. We see in Figure 2.6a that the `Twitsper` server has a higher CPU utilization than the TCM server. This is because the TCM server spends more idle time (Figure 2.6b) 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 2.6b have corresponding drops in CPU utilization in Figure 2.6a. 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 2.7a shows the number of bytes in and out with the TCM and `Twitsper` servers for a single client connection. Each line in Figure 2.7a 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 2.7b and 2.7c show the bandwidth consumed at the server as the number of bytes in and out per second. In Figure 2.7b, 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 [46] application to measure the power

consumed at the client. We made 100 posts back to back and measure the average energy consumed.

Figure 2.10 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 [38]. 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 [46]). 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 con-

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

Figure 2.9: Total energy consumption (mJ)

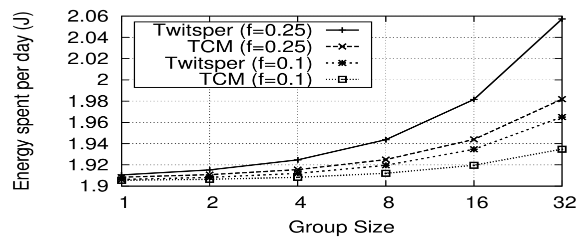


Figure 2.10: Client energy consumption

sumption of `Twitsper` with that of a popular Twitter client (TweetCaster[60]), 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 2.8, `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 2.9. Referring the reader back to Figure 2.10, 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.

2.9 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 work, 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.

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 ← select a random number
7: $Entry_a$ ← 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: $Dummy_a^i$ ← 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 $\leftarrow$ First 8 bytes of SHA-512(msg)

6: PASS $\leftarrow$ msg concatenated with sender's ID

7: $K_g \leftarrow$ PBKDF2(PASS, SALT)

8: Decrypt $response$ using K_g and cache embedded $listID$ with $msgID$ for future replies

9: M $\leftarrow$ 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 $\cap$ user's followers

7: PrivSend(msg,$group$)

Chapter 3

Hermes

3.1 Introduction

Today, a user has to implicitly trust an OSN with all her content [22]. Leakage of information from OSN servers [21, 20], coupled with the need for OSN providers to mine user data (e.g., for profit via targeted advertisements), have concerned users [56]. Though there have been prior efforts that seek to enable private communications on OSNs, we argue that all of these have limitations that either hinder their applicability in practice, or simply result in other privacy leaks that are not adequately dealt with. While posting encrypted data on OSNs [72, 95] can work in theory, it compromises the profit motives of an OSN if done at scale. Alternatively, one could share private content with OSN friends by storing data outside the OSN provider's control. Prior approaches that follow this approach either store private content in the cloud [17, 120, 58] or across client machines [100, 108]. The former simply leaks private information to the cloud providers in lieu of the OSN providers, and also increases user costs. The viability of an approach based on the latter depends on the availability of consistent access to client machines.

Our contributions. In this work, we design a decentralized OSN architecture, *Hermes*, with cost-effective privacy in mind. *Hermes*'s key goal is to ensure that any content shared by a user as well her sharing habits are kept private from both the OSN provider and undesired friends. In doing so, *Hermes* seeks to (i) minimize the costs borne by users, and (ii) preserve the interactive and chronologically consistent conversational structure offered by a centralized OSN.

Hermes uses three key techniques to meet these goals. First, it judiciously combines the use of compute and storage resources in the cloud to bootstrap conversations associated with newly shared content. This also supports the high availability of the content. Second, it employs a novel cost-effective message propagation mechanism to enable dissemination of comments in a timely and consistent manner. It identifies and purges (from cloud storage) content that has been accessed by all intended recipients. Lastly, but most importantly, *Hermes* carefully orchestrates how fake postings are included in order to hide sharing patterns from the untrusted cloud providers used to store and propagate content, while minimizing the additional costs incurred in doing so. A key feature of *Hermes* is its flexibility in deployment; it can either be implemented as a stand alone distributed OSN or as an add-on to today's OSNs like Facebook (while maintaining the decentralized nature of content sharing). The latter option is especially attractive for potentially quick widespread deployment.

To summarize, our contributions are as follows:

Design of *Hermes*: As our primary contribution, we design *Hermes*. It utilizes extremely small amounts of storage, bandwidth, and computing on the cloud to facilitate real-time, consistent and anonymous exchange of private content. Importantly, *Hermes* ensures that cloud providers cannot discover the users involved in private conversations and is robust to the intersection attack

[83].

Analyzing OSN data to determine resource requirements: Based on 1.8 million posts crawled from Facebook, we 1) perform an analysis to determine key parameters for implementing *Hermes*, and 2) conduct realistic simulations to show that (a) *Hermes* effectively anonymizes users' sharing patterns and (b) *Hermes*'s use of cloud resources is low enough to facilitate its practical deployment. Our analysis suggests that, for 90% of users, *Hermes* would typically require 1) cloud storage of much less than 5 MB, and 2) a compute instance on the cloud that is active for roughly 4 days every month. This corresponds to a monthly cost of less than $5 per user. With this budget, *Hermes* ensures that cloud service providers are unable to guess the members or the group size of any private conversation. If the cloud provider attempts to randomly guess the group members, it is correct less than 15% of the time.

Implementation and evaluation: We implement a working prototype of *Hermes* as a rudimentary add-on to Facebook. Our evaluations show that *Hermes* incurs low cost, and the user experience with it, in terms of delays, is similar to that with Facebook.

Scope: The privacy preserving features of *Hermes* can be used in conjunction with a centralized component that can be used for posts that are not intended to be private. In fact, our prototype of *Hermes* as an add on to Facebook achieves just that; private posts are directed to *Hermes* while other content is shared in the traditional way. We wish to also point out that we do not explicitly consider mobile users in this work; however, *Hermes* can be used in such contexts, and across multiple devices.

3.2 Related Work

Next, we describe relevant related work on privacy in OSNs.

Improving privacy in OSNs: Several systems propose to post encrypted content on OSNs to protect privacy [125, 95, 72, 110, 91, 74]. However, encryption precludes OSN providers from interpreting posted content and/or hides users' social connections from OSNs. These are not in the commercial interests of OSN providers, who may thus disallow such postings. *Hermes* does not post any encrypted content on an OSN; it uses either cloud storage or users' personal devices to do so. Further, it does not use a centralized OSN framework to inform users of new content; doing so also informs the OSN provider of the specifics of ongoing conversations.

Distributed OSNs: Other efforts propose storing private shared data on devices other than OSN servers [108, 100, 120, 17, 113]. However, unlike *Hermes*, they either expose user sharing patterns to cloud providers [120, 17] or degrade user experience in terms of timely and consistent sharing. Systems that store private data in the cloud do not control either storage or bandwidth costs which will increase over time as the volume of shared data grows. While other systems store the data on users' personal machines [108, 100] to reduce costs, the low availability of these machines (they may be turned off when not in use) reduces the timeliness of conversations and compromises data consistency. *Hermes* combines resources on cloud services (within limit) with that on users' personal machines to support cost-effective sharing that is held privy from cloud providers.

Priv.io [131] is a new decentralized OSN that aims to minimize the cost incurred for facilitating private content sharing. However, Priv.io critically relies on support for advanced messaging APIs from cloud services, which restricts the generality of Priv.io's architecture. In contrast, *Hermes* only requires cloud storage services to offer a minimal PUT, GET, DELETE interface. Most

importantly, due to Priv.io's reliance on messaging APIs offered by cloud services, unlike *Hermes*, it does not attempt to hide sharing patterns (i.e., whom does a user share data with) from cloud providers.

Other related work: Other efforts [92, 123, 91] that have tried to secure the data stored on untrusted servers or on the cloud do not try to account for OSN-specific characteristics (e.g., hiding content sharing patterns). Several techniques, such as oblivious transfer [111] and private information retrieval (PIR) [105], have been developed to prevent a third party from observing what information is exchanged between any two communicating parties. However, unlike *Hermes*, these solutions would either significantly increase cost or degrade timeliness. Moreover, *Hermes* enables anonymity in OSN conversations without requiring all members of a conversation to be simultaneously online.

3.3 Goals and Threat Model

Goals and challenges: Our over-arching goal is to design a decentralized, private OSN architecture. In doing so, we have the following three objectives.

- *High availability, timeliness, and consistency:* First, we seek to preserve the desirable properties enabled by a central provider. Specifically, (a) users should always be able to access content shared with them, (b) content shared by a user should be received by the intended recipients in a timely manner, so as to preserve the interactive comment "threads" associated with content shared on OSNs, and, (c) all users involved in a conversation should receive comments in the same causally consistent order. How do we preserve these desirable properties despite the fact that content is stored in a decentralized manner in *Hermes*?

- *Protect the privacy of content and sharing patterns:* While *Hermes* lacks any central OSN provider, cloud services used to store and disseminate content may be able to monitor conversations. How do we preserve the privacy of shared content from cloud providers and prevent them from discovering the participants in any conversation?

- *Minimize cost:* Finally, we seek to minimize the storage, bandwidth, and compute costs incurred by users in *Hermes*'s use of cloud services. This is made particularly challenging due to the previous two goals. For example, one could enable timely dissemination of comments if every user were to maintain her own compute instance in the cloud at all times. Similarly, the members of any particular conversation can be hidden from cloud providers by having all users constantly exchange fake comments with each other. However, such measures will result in high cost.

Threat model: We assume that all service providers (of cloud services or of a centralized OSN) preserve the integrity and availability of the data that users store on them. This may be either in fear of bad publicity or because users pay for the service. However, we assume that all service providers may benefit from inferring information associated with private conversations. Thus, we treat all service providers as "curious but honest", as in [131]. Moreover, if cloud providers discover the members of private conversations, this information may leak. Therefore, we seek to ensure that, when a group of users are involved in a private conversation using *Hermes*, no one outside the group learns either the size or membership of this group. Here, we assume that cloud providers can perform network-level traffic analysis (e.g., a provider can map the IP addresses from which it is accessed, to user identities). The use of anonymity networks such as Tor [85] would not scale to meet the traffic demands of a large-scale OSN. Lastly, ensuring the privacy of a users' conversation

group via fake messages (as in *Hermes*) requires that the user has a sufficiently large set of friends; if a user has very few friends (e.g, < 5), preserving the anonymity of a private conversation group is hard. We assume that users have friends of the order of hundreds, as is typical on OSNs [5]; however, we assume the sizes of private conversation groups to be much smaller.

3.4 Hermes Architecture

In this section, we describe the *Hermes* architecture with a simple running example. We defer the analysis of the security properties of *Hermes* to Section 3.6.

3.4.1 Hermes Architecture Overview

Consider an OSN user (Alice), who wishes to share some content (say a photo) meant only for her friends Bob and Chloe. To ensure that neither the private content nor the intended recipients are exposed to anyone other than the intended recipients, Alice encrypts the photo with an appropriate key (known only to Bob and Chloe) and shares it using resources in the cloud. There are four main issues that we need to address to enable this: 1) how do Bob and Chloe discover this content *and* the associated key to decrypt it?, 2) how can comments on the content, posted by Alice, Bob, and Chloe, be disseminated in a timely manner?, 3) how do we prevent the cloud provider from inferring the members of this private exchange?, and 4) how to minimize costs incurred by Alice, Bob, and Chloe? We next describe how *Hermes* tackles these questions.

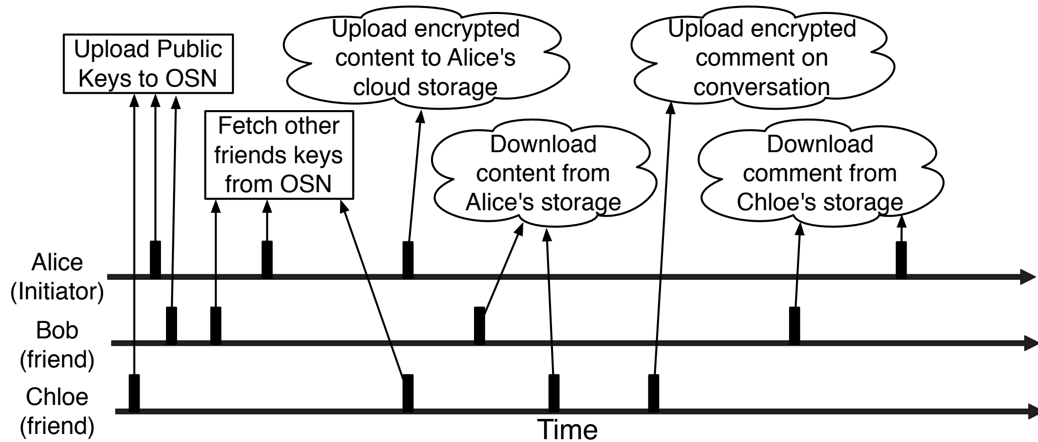


Figure 3.1: Illustration of conversation timeline.

3.4.1.1 Sharing new content

As shown in Fig. 3.1, every user (including Alice) first posts her public key component to enable an ECDH key exchange (a brief overview of ECDH is provided in Section 3.12) on her OSN profile [1], which is visible to all of her friends. Any particular user can thus fetch the public key components from her friends' profiles and derive pairwise keys with any of her friends (more details in Section 3.6).

To share a photo, Alice's *Hermes* client chooses a new *group key* and creates two encrypted copies (using a cipher such as AES) of this key, one copy encrypted using her pairwise key with Bob and the other using her pairwise key with Chloe. Alice's client then stores these encrypted group key copies in Alice's cloud store. The client also puts the photo, encrypted with the group key (again using AES), in her cloud storage.

Bob's and Chloe's *Hermes* clients periodically check Alice's cloud store for new content shared with them. When new content exists, they fetch their respective encrypted group key copies from Alice's store (the process is discussed later) and extract it using their respective pairwise keys

[1] This could be on her favorite OSN or *Hermes*'s servers depending on the implementation.

with Alice (more details in Section 3.6). Bob's and Chloe's clients then store the extracted group key locally on their personal devices. The clients can fetch and decrypt the photo using this group key.

3.4.1.2 Enabling OSN-like conversations

We next describe how *Hermes* enables OSN like conversations with low cost.

Disseminating comments: After Bob and Chloe discover Alice's photo, the three of them may post comments on it. Our goal is to ensure that these comments are disseminated in a timely and consistent manner, as is the case with a centralized OSN. If all users involved in the conversation are always online, whenever a user posts a comment, that user's client can establish secure connections with the clients of the other members of the conversation and inform them of the new comment. However, in practice, Alice, Bob, and Chloe may come online at different times. Thus, there has to be a common arbitrator that enables a user to discover comments posted when she is not online and facilitates the chronological ordering of posted comments.

For this, we propose that the user who initiates the conversation (Alice) uses a computing instance in the cloud to act on her behalf as the arbitrator. Today, there are many such online computation resources available (e.g., Google App Engine [26], Heroku [33], and Amazon EC2 [2]). Alice's instance acts as a proxy for her.

Reducing compute instance costs for Alice: However, Alice may not be able to afford to keep a compute instance active at all times. Thus, by default, Alice's *Hermes* client terminates her instance following a preset period after Alice has shared any new content (discussed later in Section 3.7). However, there may be users who come online much after the instance has been terminated. To deal with such cases, the *Hermes* framework uses log files called *update files*

or $ufiles$ for short. Every user maintains a $ufile$ in her cloud store for each of her friends; these $ufiles$ are created and the location of the $ufiles$ are exchanged between friends either when a user installs the *Hermes* client or when the user adds a friend. Thereafter, whenever a user (Alice) posts a new piece of content relevant to a specific friend (Chloe), Alice's *Hermes* client adds an entry to the $ufile$ for Chole. In all subsequent discussion, for the purposes of clarity, we only provide a high level description of how $ufiles$ are used and defer a detailed description to Section 3.13.

If Bob comes online after Alice's compute instance has been terminated, his client retrieves her $ufile$ for him and locates any new updates. This allows Bob's client to retrieve any content or comments shared by Alice. His client then indicates that the content has been retrieved in his own $ufile$ for Alice. Upon checking this entry when Alice comes online, her *Hermes* client deletes the original entry in her $ufile$ for Bob.

Note that $ufiles$ also enable a user to discover comments without waiting for the initiator of a conversation to come online. For example, if Bob is also Chloe's friend, Bob's $ufile$ for Chloe will indicate that he has commented on Alice's photo. Chloe can thus retrieve the comment and associate it with the original photo received from Alice (based on an associated conversation ID).

Ensuring consistency of comments with** $ufiles$**: It is easy to see that the above framework allows a user who comes online after the instance is terminated to retrieve the object and reconstruct the conversation associated with it (i.e., put the comments in chronological order using vector timestamps [106]) as long as all the group members are his friends. However, if a group member (say Chloe) is not Bob's friend, Chloe is unable to read Bob's $ufiles$; in fact, such a file for Chloe will not exist in Bob's cloud storage, since $ufiles$ are only maintained for friends. This violates the structure of an OSN conversation.

To deal with such cases, Alice relays the locations of the comments associated with her content via her own $ufiles$ for each member of the conversation (who are her friends since she initially shared the content with them). Since there may be delays in relaying these locations (in rare cases where multiple users come online much after the compute instance is terminated), there may be temporary loss in the chronological consistency for a user who comes online at a late stage. There is an inherent trade-off here; the longer Alice's compute instance is active, the less likely is that there is such a loss in consistency. However, this will incur a higher cost.

Reducing storage costs: Finally, Alice cannot store her photo (or for that matter, Bob cannot store his comment) on the cloud forever. This would result in a monotonic growth in the consumed storage and thus, the associated cost. Instead, with *Hermes*, content is removed from cloud storage after a certain time (the duration can be set by Alice, but we discuss what might be appropriate in Section 3.7). A simple way of ensuring that all group members have seen the content before it is purged is for Alice to check if they have indicated this to be the case in their $ufiles$ for her. If a user (say Bob) comes online after a prolonged absence (much after when the content was removed from the cloud), he may still learn of its existence via Alice's $ufile$ meant for him. Via his own $ufile$ for Alice, Bob's client then requests Alice for the purged content. When Alice comes online next, her client then copies the requested content back on to the cloud. In fact, Bob can request the purged content from any or all of the group members of that conversation (information on the group can be embedded as metadata in the encrypted content) to restore the content on the cloud for him. Once a group member (say Chloe) restores the content, Bob's $ufiles$ can be updated to indicate that the content is no longer needed from other members.

This process increases the complexity of *Hermes*'s design, and thus, is not currently im-

plemented in our prototype; however, as we show in Section 3.7, such cases are rare if one looks at typical content sharing on Facebook. Here, we also point out that *Hermes* enables users to access their content from multiple devices; we provide the details of how this is made possible in Section 3.10.

3.5 Hiding users' sharing patterns

While confidentiality of the shared information is ensured with encryption, cloud providers may be able to infer the group of users involved in any conversation simply by monitoring the access patterns of users' compute instances and storage in the cloud. Here, we discuss how *Hermes* ensures that cloud providers cannot determine any of the following: a) "when" a private conversation is occurring, b) the group size of any given conversation, and c) the individual members taking part in that conversation.

3.5.1 Hiding the membership information within each private conversation:

First, let us consider a single private conversation initiated by Alice. Our goal here is to ensure that the identities of the members of this private group and the size of the group are not exposed to anyone outside the group.

3.5.1.1 Strawman approach

To hide the group members in a given conversation initiated by Alice, one simple approach is to make $ufiles$ indistinguishable across all of Alice's friends. Whenever Alice's *Hermes* client needs to insert an entry into the $ufile$ for a particular friend, it can also insert dummy entries into

the $ufiles$ for all of Alice's remaining friends; the entries in the $ufile$ for any particular friend are encrypted with the shared pairwise key between Alice and that friend, thus preventing the cloud provider from inferring which entries are fake. Thus, based on the writes to and reads from the $ufiles$ in Alice's cloud storage, the cloud provider will not be able to determine which subset of Alice's friends are involved in ongoing private conversations.

However, this simple approach has two limitations. First, it results in high storage, bandwidth, and operational query costs for Alice, because a large number of fake entries will need to be stored by Alice and accessed by Alice's friends. Second, the cloud provider may still be able to infer the members of Alice's private conversation by observing which of Alice's friends insert updates into the $ufiles$ in their own storage space; group members will post comments, but friends who are not part of the group will not. We next discuss how we address both of these issues in *Hermes*.

3.5.1.2 Obfuscating group size

Instead of making the $ufiles$ for all of Alice's friends indistinguishable, *Hermes* attempts to hide the group members ($\mathbb{G}$) among a subset of Alice's friends ($\mathbb{D}$), where $\mathbb{G}$ is a subset of $\mathbb{D}$ (referred to as the anonymity set). Whenever an entry has to be added to the $ufile$ for any user in $\mathbb{G}$, dummy entries are also added to the $ufiles$ for those users in $(\mathbb{D} - \mathbb{G})$. The number of users in $(\mathbb{D} - \mathbb{G})$ follows an exponential distribution, with its minimum, mean, and maximum values set to α, $|(\mathbb{N} - \mathbb{G})|/4$ (rationale in Section 3.7), and $|(\mathbb{N} - \mathbb{G})|$ [2], where $\mathbb{N}$ contains all of Alice's friends. The parameter α allows us to handle small groups and is set to $\max(15, |\mathbb{G}|)$.

The effect of these parameters is that the size of the anonymity set is always at least double that of the private group. As a result, random guessing as to whether a particular user in

[2] Since private group sizes are typically small, we assume that $|(\mathbb{N} - \mathbb{G})| > |\mathbb{G}|$.

the anonymity set is a member of the group will be correct with a probability of at most 50%. For small groups of size less than 15, randomly guessing as to whether a user in $\mathbb{D}$ is a group member succeeds with probability $|\mathbb{G}|/(|\mathbb{G}|+15)$. In addition, the exponential distribution biases the anonymity set towards smaller sizes. This reduces the additional storage and bandwidth costs incurred for providing anonymity, as compared to a uniform distribution that chooses the size of the anonymity set at random from the range $[\alpha, |\mathbb{N}-\mathbb{G}|]$. Lastly, note that it is insufficient to determine the size of the anonymity set simply by inflating the group size by a fixed factor (since this clearly reveals the group size).

3.5.1.3 Preventing inference of group membership based on comments

So far, Alice has been able to share content with $\mathbb{G}$ without revealing $\mathbb{G}$ or its size ($|\mathbb{G}|$). However, since only members of $\mathbb{G}$ will post comments on the shared content, the cloud provider will be able to distinguish the users in $\mathbb{G}$ from all those in $\mathbb{D}$. Thus, the additional fake members in $\mathbb{D}$ must also post fake comments as part of the conversation (these fake comments are discarded and we discuss the implementation details in Section 3.8).

A naive approach would require all the additional members in $\mathbb{D}$ to post as per either some random distribution or based on their previous posting habits. However, it will be hard to provide any anonymity guarantees with such an approach. Moreover, since we assume that the source code for the *Hermes* client is publicly accessible, cloud service providers will have access to any distributions hard-coded into the client software.

Instead, our approach for posting of dummy comments works as follows. We divide time into slots, where all the members of a conversation can derive the slot boundaries based on the time at which the conversation was initiated (see Figure 3.2). We refer to each time slot as a round. In

each round, every member of the conversation who is online during that period posts at least one comment, at a random point in time during that round. Those group members who have no real comments to post in a particular round—this includes both the users in $(\mathbb{D} - \mathbb{G})$ and the users in $\mathbb{G}$ who have no comments to post during that round—post at least one dummy comment during that round. All entries added to any $ufile$ are padded to a fixed size in order to hide the number of comments being posted by a user; this is necessary because a user who posts real comments may post multiple comments in a single round.

Importantly, every user in $\mathbb{D}$ posts either real or fake comments at only one particular time during each round. This ensures that the cloud provider cannot distinguish between users in $\mathbb{G}$ and those in $(\mathbb{D} - \mathbb{G})$, since it observes the same pattern of writing to and reading from $ufiles$ for all users in $\mathbb{D}$. Thus, when all users in $\mathbb{D}$ are online, the cloud provider has only a $\frac{\mathbb{G}}{\mathbb{D}}$ probability of correctly inferring whether a particular user in $\mathbb{D}$ is indeed a member of the private group $\mathbb{G}$.

3.5.1.4 Selecting the length of a round

A key design decision in instantiating the approach described above is to determine how time should be divided into rounds. Shorter rounds lead to more timely dissemination of comments. This is because when one user posts a comment in a particular round, another user can respond to this comment only in the next round; note that every user can post comments only once in each round. In contrast, longer rounds result in lower cost since fewer fake comments are posted, but compromise timeliness. Based on this trade-off, we split the timeline of a conversation into rounds as follows.

Our design is based on the observation that the commenting activity associated with most conversations is high when the conversation initially begins. After this initial period, the conversa-

tion goes stale and users may have few new comments.

Given this, to reduce the costs incurred to guarantee anonymity (hiding user sharing patterns), we partition any conversation into two phases. The first phase is when the conversation is fresh and one is likely to expect a comment in the near future. In this phase, the timeliness of comment dissemination is important, and therefore we keep a round's length short. Once several rounds with no real comments are observed, the conversation transitions to the second phase. The second phase aims to capture that phase of a conversation where no user has posted a comment for a while and there is a low probability of new comments. In this phase, we want to limit the cost associated with the conversation by minimizing the number of fake comments. The key property we exploit in this second phase is that, since the conversation is already stale, the timeliness of straggler comments posted during this period is not of concern.

In the first phase, all rounds are of equal length as long as at least one real comment is posted in each round. When there are no real comments in a particular round, we increase the length of the round by a multiplicative factor. The round length in the first phase is reset to its original value when a real comment is posted in the previous round. After a certain number of consecutive rounds with no real comments, the conversation transitions to the second phase. We model round durations in the second phase as a geometric series also, but use a larger multiplicative factor to increase round durations as compared to that used in the first phase. When a real comment is posted in the second phase, the conversation is reset to the first phase, but a fewer number of rounds of inactivity transitions the conversation back to the second phase in this case.

Note that the users who are in $\mathbb{D}$ but not in $\mathbb{G}$ cannot distinguish between real and fake comments; this is intentional, since we seek to hide group membership not only from cloud

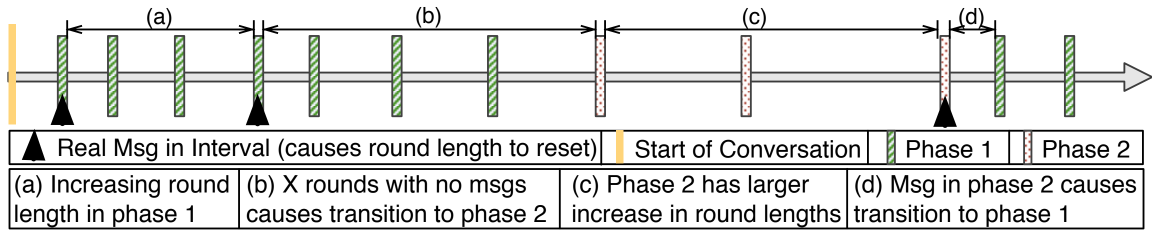


Figure 3.2: Round structure in *Hermes*.

providers but also from users who are not in $\mathbb{G}$. Therefore, in every round, every user in $\mathbb{G}$ broadcasts to all of her friends who are also in $\mathbb{D}$ as to whether a real comment was posted in the previous round or not. Every user in $\mathbb{D} - \mathbb{G}$ who receives this notification relays this on to all of the user's friends who are in $\mathbb{D}$, exactly once. Thus, a user who receives a notification cannot distinguish between whether this was an original broadcast or a relayed broadcast. Once a user receives this information, she can independently determine what the length of the next round will be and when the transition between phases is triggered. Note that, from the cloud provider's perspective, these notification messages that convey whether a real comment was posted in a particular round are indistinguishable from real and fake comments. Moreover, though the cloud provider may be able to infer when real comments are posted based on when inter-comment spacings decrease, *no one other* than the users in $\mathbb{G}$ can determine *which* users posted the real comments.

3.5.2 Hiding users' conversation patterns by handling intersection attacks:

Thus far, we have only considered hiding the identities of group members within a conversation. Unfortunately, the above approach is insufficient in completely hiding a users' sharing patterns across conversations. If fake users (in $\mathbb{D} - \mathbb{G}$) are chosen randomly from the user's friends ($\mathbb{N}$), the cloud provider can infer that users who appear repeatedly in different conversations are likely to indeed be real members of private groups.

To prevent such intersection attacks [83], we need to preserve anonymity *across* conversations. For this, we seek to ensure that a consistent group of $\mathbb{K}$ friends ($\mathbb{K} \subset \mathbb{N}$) appear across the conversations initiated by a user (Alice); we refer to this group as the Top $\mathbb{K}$ group. Thus, if a private, repetitive, group initiated by Alice is of size $\mathbb{G}$, the provider can only randomly guess if a user in the group of $\mathbb{K}$ friends ($\mathbb{K} >> \mathbb{G}$) is a true repetitive member with probability $\frac{\mathbb{G}}{\mathbb{K}}$. In essence, this provides $|\mathbb{K}|$-anonymity [124].

Our approach to form the Top $\mathbb{K}$ group (algorithmically depicted below) is to (1) tune the membership of $\mathbb{D}$ and (2) use fake conversations. We identify the friends with whom Alice consistently has private conversations (say $\mathbb{K}_1 \supset G$) and include them in the Top $\mathbb{K}$ group. We then fill the remainder of the Top $\mathbb{K}$ group with other friends with whom Alice rarely initiates private conversations (say $\mathbb{K}_2$).

Stage 4 Learn user habits

1: **for** Next M_1 conversations **do**

2: $\{\forall x \in \mathbb{G} : x.count+ = 1\}$

3: set $\mathbb{D} = \mathbb{N}$ and start conversation with entire friends list.

4: **end for**

5: Select $\mathbb{K}_1$ users with highest count values

6: Select $\mathbb{K}_2$ random friends s.t. $\{\forall x \in \mathbb{K}_2 : x \in \mathbb{N} \wedge x \notin \mathbb{K}_1\}$

7: reset count Values

8: **return** $\mathbb{K} = \mathbb{K}_1 \cup \mathbb{K}_2$

Stage 5 Use learned habits

1: **for** Next M_2 conversations **do**

2: Select size for $|\mathbb{D}| = \alpha + Exp(\frac{|\mathbb{N}|-|\mathbb{G}|}{4})$

3: $\forall x \in (\mathbb{N} - (\mathbb{G} \cup \mathbb{K})) : \mathbb{P}(x \in \mathbb{D}) = p$

4: Fill $\mathbb{D}$ from $\mathbb{K} - \mathbb{G}$ with probability of $x \in \mathbb{D} \propto Max(c - x.count, delta)$, where $c = Max(\forall x \in \mathbb{K} : x.count)$

5: $\forall x \in \mathbb{D} : x.count{+}{=}1$

6: schedule $\lceil \frac{\mathbb{K}-\mathbb{D}}{\mathbb{D}} \rceil$ fake conversations with $\mathbb{G} = \emptyset$ in current M_2 conversations

7: **end for**

3.5.2.1 Tuning the membership of $\mathbb{D}$

As the first step, we need to determine which of Alice's friends consistently belong in private conversations. While doing so, in order to preserve anonymity, we simply use the naive approach wherein all of her friends are included in all conversations. This is referred to as the *first stage* or the *learning stage* of anonymizing conversations (Stage 4). This stage is executed for M_1 (tunable parameter) conversations. During this stage, the *Hermes* client learns of the user's posting habits and with which friends the user is more likely to privately exchange information (set $\mathbb{K}_1$). It then forms the Top $\mathbb{K}$ group as described above.

In the second stage (Stage 5) which is then executed for the subsequent M_2 (tunable parameter) conversations, we reduce the total cost incurred by a user (Alice) by only consistently including the Top $\mathbb{K}$ group in private conversations. In each true conversation initiated by Alice, we now form the group $\mathbb{D}$ for that conversation as follows. First, all the user's friends that are neither

part of the conversation group $\mathbb{G}$ nor the Top $\mathbb{K}$ group (determined in the first stage) are considered as candidates for inclusion in $\mathbb{D}$. Each of these candidates is included in $\mathbb{D}$ with a very small fixed probability p. This ensures that friends outside of Alice's Top $\mathbb{K}$ group, i.e., users with whom she rarely exchanges private content, are included with a small probability; this protects against the server correctly identifying true rare inclusions of such friends. Subsequently, Alice's friends that are part of her Top K group but not in $\mathbb{G}$, are considered for inclusion. The probability that a particular user (say Chloe) in the Top $\mathbb{K}$ group is selected is proportional to the difference between the maximum number of conversations any member of Top $\mathbb{K}$ group is involved in (both true or fake roles), and the number of conversations that Chloe is involved is involved in (both true or fake roles). This ensures that all of the members of the Top $\mathbb{K}$ group are consistently involved in conversations.

3.5.2.2 Using fake conversations

In spite of filling the groups as above, it is possible that real users appear more often than fake users. To address this, we schedule $\lceil \frac{\mathbb{K}-\mathbb{D}}{\mathbb{D}} \rceil$ *fake* conversations (with fake comments) where $\mathbb{G} = \emptyset$ (since each real conversation already includes $\approx \mathbb{D}$ members from the Top $\mathbb{K}$ group). The groups, $\mathbb{D}$, for such fake conversations are filled exactly as the real conversations are filled. Together, the above two steps of stage two ensure that every member of the Top K group is in (approximately) the same number of conversations on average.

To cope with the dynamics of Alice's sharing behaviors (she could converse more often with Bob and Dave at some point in time, and at a different time, exchange more private content with Chloe and Eve), we return to the first stage periodically to recompute the Top $\mathbb{K}$ group. Here, we take care to ensure that only minimal changes are made to the group $\mathbb{K}_2$ to prevent the server from identifying these as fake users.

Finally, instead of using fake conversations, to reduce costs one can think of suppressing an initiator's conversations with particular users with whom she is conversing too frequently. We do not explore this option as it violates our goal of ensuring timely sharing as in a traditional OSN.

3.6 Other Security properties of Hermes

We next discuss other security properties of *Hermes*, given the threat model described earlier in Section 3.3.

***Hermes*'s Encryption Mechanics:** *Hermes* uses ECDH to generate pairwise keys between every pair of friends. In our example, Alice and Bob generate $s_{Alice,Bob}$ and Alice and Chloe generate $s_{Alice,Chloe}$ (by using each other's public key components). Since users authenticate themselves to an OSN (which may be a *Hermes* server, depending on the implementation) over a TLS (Transport Layer Security) session, using an HTTPS connection, the public keys posted on a user's profile can be verified to belong to that user. This is similar to the approach in [50] except that, instead of having static key components, the OSN serves as an authenticated channel that implicitly verifies the true source of dynamic DH components.

Once Alice has computed pairwise keys with each of her friends, in any particular conversation with Bob and Chloe, her *Hermes* client encrypts separate versions of the group key (chosen at random) with $s_{Alice,Bob}$ and $s_{Alice,Chloe}$. This encryption protects the group key a) from any storage media used (OSN and the cloud), and b) from Alice's friends who are not part of the conversation. Secured this way, the group key can subsequently be used to encrypt/decrypt all content shared between Alice, Bob, and Chloe in this particular conversation.

Protecting the leakage of content: Given the above security mechanisms, we next dis-

cuss how the confidentiality of shared information is protected with *Hermes*. Specifically, we consider how the information is protected from the service providers, and on an insecure communication channel.

Storage, Compute and OSN services: With *Hermes*, users' public key components are stored on user profiles on the *Hermes* server. As long as these key components are uploaded by a user's *Hermes* client using a HTTPS connection (and the server certificate from a CA is valid), the transfer is secure. We assume that the *Hermes* server preserves the integrity of the uploaded public key components (see threat model). Neither the server nor a third-party can reconstruct the pairwise key for a pair of users simply based on their public key components.

Channel: We next consider the possible attacks on the channel on which the keys and content are exchanged.

Man-in-the-middle (MIM) attack: Conventional ECDH systems address the MIM attack [50] using Public Key Infrastructure (PKI). With PKI, users verify via a certificate authority (CA) that a given public key $x_U * p$ belongs to user U; thus A and B would discover that the public keys received from C are invalid. While *Hermes* can use this approach, it bypasses the need for adding additional keys (given the scale of today's OSNs) to the CA. This decreases the load on the PKI and reduces the cost to end users. Since users authenticate themselves to an OSN over a TLS (Transport Layer Security) session, using an HTTPS connection, the public keys posted on a user's profile can be verified to belong to that user.

Over the channel modification of content: Since shared content is downloaded from cloud storage, an attacker can intercept a *Hermes* client's HTTP request and respond with modified content. To protect against this, *Hermes* clients append an AES-CMAC [122] (Cipher Message Au-

thentication Code) to any content encrypted using the group key R. If an attacker modifies the post, the CMAC verification fails at the recipient; note that an attacker cannot compute the MAC for the modified file since it does not have R.

3.7 Quantifying Cost, Anonymity, and Timeliness Trade-offs

In order to tune *Hermes*'s configuration, we seek to understand the trade-offs between anonymity, timeliness, and cost. To do this, we crawl a large dataset from Facebook, and use the posting habits seen to perform a trace-driven simulation of *Hermes*.

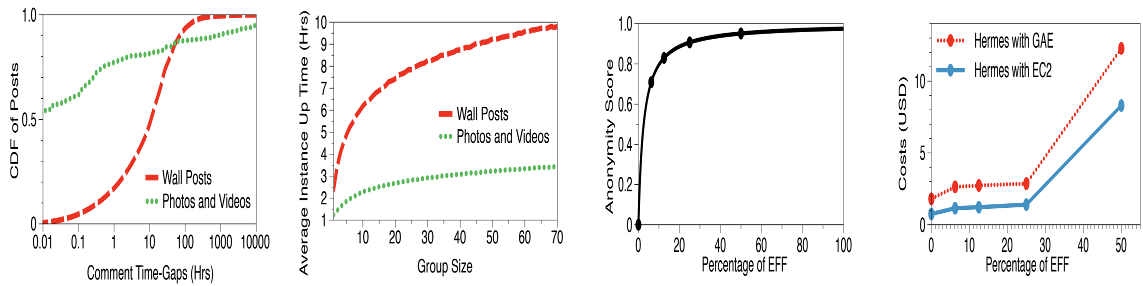
Understanding the temporal nature of conversations: We first seek to understand how long a posting is likely to be of interest to a user's friends, in the common case. Our particular interest is the *time gap* between when specific content was posted by a user and when the friends of that user lose interest in viewing it (the interested friends have already viewed it with high probability). However, it is impossible to accurately determine this duration without access to Facebook's server-side logs. Therefore, we instead use users' comments on a post as a proxy for their interest in the post. Though all users who find a posting to be of interest may not comment on it, previous studies have shown that the number of friends that see a post and the number of friends commenting or liking it are positively correlated [76]. Thus, we ask the question: for those postings that have associated comments, what is the time gap between the instant when the posting was made and when the last associated comment was posted?

Due to the lack of a publicly available dataset on users' posting habits, we crawled the profiles of 68,863 Facebook users using a combination of FQL (Facebook query language) and RestFB. Our crawled dataset, which spans a month, roughly comprises 1) 1.8 million wall posts and

associated comments, and 2) 40K posts of either photos or videos with ≈ 35K associated comments. Remarkably, 70% of the 1.8 million posts did not have any associated comments. Thus, we look at the other 30% and the photos/videos to determine the time-gap between when the initial content was posted and when the last associated comment was seen. Based on Fig. 3.3a, we set the duration for which a user caches data on her cloud storage to 3 days; 90% of posts do not receive new comments beyond this period. In outlier cases, where content is sought long after it was posted, we sacrifice timeliness for resource thriftiness as discussed earlier.

A simulation of *Hermes*: Next, we build a simulator to capture user interactions with *Hermes* in a large-scale setting; the simulation provides both 1) an understanding of how *Hermes* may perform, and 2) a validation of *Hermes*'s ability to provide anonymity with limited resources (small volumes of storage and bandwidth, few operational queries, and short uptimes for a user's computing instance). To the best of our knowledge, there does not exist a simulator that mimics user interactions on an OSN.

Determining simulation parameters: The first input required by our simulator is a measurement of how often users come online. This dictates the expected time for disseminating content across *Hermes* clients, and thus, impacts how long the computing instance, or data stored on the cloud, will need to be active. Note that the *Hermes* client on a user's device does not need her to interact with it to fetch new content. Thus, the only time of interest is when the device is powered on and connected to the Internet. Here, we use data from [126], which provides the time per day for which users' devices are active. We assume that most powered on devices today are connected to the Internet. The weighted average of this time for desktops is 9.7 hours a day. The weighted average of online time for portable computers is comparable at 8.3 hours a day [126].



(a) Distribution of gap times. (b) Server up time vs. group size.

Figure 3.3: Analyses with Facebook data

(a) Anonymity as a function of % EFF (b) Trade-off between anonymity and cost

Figure 3.4: Anonymity trade-offs per conversation

Second, to determine when a friend retrieves a private posting made by a user, we compute the relative time-gap between when the user is online and when the friend comes online later. We assume that users in similar time-zones are online during similar periods; if users are in time zones far apart, this time gap may be larger. Unfortunately, we were unable to access the location information of users in our data set; Facebook does not allow programmatic access to this information. Hence, we use two approximations to characterize the distribution of when the friends of a user come online. 1) We assume that users come online at random instances uniformly distributed over a 24 hour period, and stay online for a uniformly distributed period with an average of 9.7 hours; we believe that this model represents the likelihood that a user's friends are distributed all over the globe. 2) We consider a best case scenario wherein all of a user's friends are in her time-zone; here, we assume that the user and her friends come online within a 12 hour period. Again, the time at which the user comes online is uniformly distributed within this period, and the duration for which one stays online is chosen as before.

Third, to accurately represent a user's posting habits, we replay the posts in our Facebook data set. Since the posts we crawl are those shared by a user with all her friends, we obtain an estimate for the expected *private* group size from [87] and [5]; these studies suggest that while the social group size of a user is about 190, the more intimate size of a social group is 12. On this basis, we consider expected group sizes of 15, and use a uniform distribution with variance 10 (to cover group sizes from 5 to 25).

Selecting system parameters: To simulate *Hermes*, we also need to choose the parameters that control how the system trades off timeliness for anonymity. The two phases of a conversation, as discussed in Section 3.5, depend on four factors, namely the initial length of a round (l), the multiplicative growth rate in phase one (A_1), the growth rate in phase two (A_2), and the number of rounds with no real comments in phase 1 (X), after which a conversation transitions to phase 2. To set X, we observe from the Facebook data that 95% of the time, the time gap between two consecutive comments is less than 24 hours. In other words, a conversation is unlikely to be of interest to friends if there are no comments for about 24 hours. Hence, we transition a conversation from the first to the second phase if we do not see a comment for 24 hours. Later we vary l and A_1 in our simulator and examine the effects on average cost and timeliness. For phase 2, we seek an exponential growth, but want to simultaneously keep in check the delay incurred in retrieving straggler comments; thus we set $A_2 = 2$.

Simulator design: Our simulator captures all the features of *Hermes* described in Sections 3.4 and 3.5. In our simulation, every user initiates conversations and posts comments as per her posting activity on Facebook. For each private conversation initiated by a user, we select a randomly chosen subset of the user's friends based on her posting habits and the expected group size

considered. We consider the size of every shared photo as 2 MB and the size of all other private posts as 0.5 KB (these numbers are much larger than what we got from our crawled data). Since a user's comments in our crawled data may be on posts made by users outside our crawled population, we post any comment by a user to a pre-existing randomly chosen conversation that she is involved in.

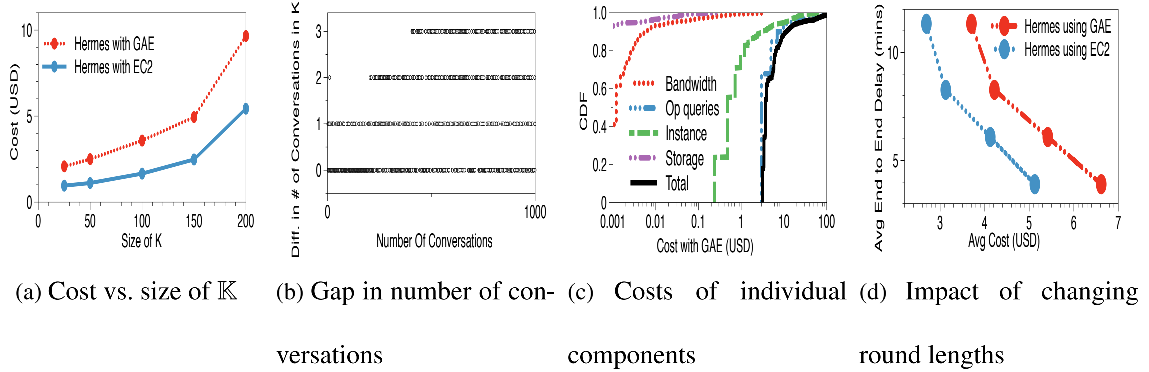
Results and interpretations: Our metrics of interest are (i) the time for which a user's compute instance needs to be active, (ii) the anonymity (likelihood of guessing if a friend is a true group member) a conversational group is provided, (iii) the total cost incurred, and (iv) the loss of timeliness due to users receiving stale data.

Compute costs: First, we seek to determine the time for which the instance associated with any object (post) needs to be active. Recall that a *Hermes* client of a group member obtains new content as soon as she comes online. Considering the two approximations discussed above for when users come online, Fig. 3.3b plots the distribution of the time it takes for all the members of a private group to access the object. This is the time for which the compute instance has to be up. To handle the common cases where the group size is small (< 15), the compute instance needs be up for 6–7 hours even if a user's friends are globally distributed; if the friends are all local, it needs to be up for ≈ 4 hours. One may expect that in a typical case (when a user has both global and local friends), the compute instance will have to be up for a duration somewhere in between 4 and 7 hours; we conservatively choose the duration to be 10 hours. In rare cases where not all members access a posted object within the 10 hours, we trade-off timeliness in serving the content for lower cost. Based on this, our simulations indicate that, for $\approx 90\%$ of the users, *Hermes* will need to keep their instances active for less than 100 hours (or 4 days) in a month, in order to privately exchange

all the Facebook data that they shared in the entire month. In comparison, prior solutions for OSN privacy [120, 17] require every user to persistently have a compute presence in the cloud.

Quantifying anonymity: Second, we seek to quantify the anonymity provided to a conversational group. First, we consider each conversation individually. We define the *anonymity score* to be the probability that the server is unable to correctly identify a group member as true or fake (as discussed in Section 3.5 this is $(1 - \frac{|\mathbb{G}|}{|\mathbb{D}|})$). In Figure 3.4a, we plot the anonymity score with *Hermes*, while varying the number of fake group members. The x-axis represents the percentage of the initiator's friends outside the group, who are added as fake members (denoted as external fake friends or EFFs). Specifically, $\mathbb{D} = \mathbb{G} \cup \text{EFF}$ where, $\text{EFF} \subseteq \{\mathbb{N} - \mathbb{G}\}$. The y-axis represents the anonymity score. Since the likelihood that the server is able to guess correctly is approximately proportional to the ratio of the number of true members (fixed) to the size of the composite group (with true and fake members), the anonymity score steeply increases as the size of the composite group increases initially. Beyond a certain point, we reach a point of diminishing gains, wherein the increase in the anonymity score is less significant with an increase in the composite group size. To achieve an anonymity score of about 0.9, we need to add 25% of the friends outside the true group as fake members in each conversation.

Per conversation anonymity vs. cost trade-off: Next, in Figure 3.4b, we depict the expected (total) cost incurred as a function of the percentage of EFFs. We obtain the per-resource costs for different contributing factors from [2, 3, 27], and multiply this with the amount of resources consumed. For an anonymity score of 0.9 (% of EFFs = 25), we see that the expected *total* monthly cost per user is relatively low ($<$ \$ 4) with both Google App Engine (GAE) and Amazon EC2 (storage and computing are from the same provider). Thus, an EFF of 25% (or $|\mathbb{N} - \mathbb{G}|/4$)



(a) Cost vs. size of $\mathbb{K}$ (b) Gap in number of conversations (c) Costs of individual components (d) Impact of changing round lengths

Figure 3.5: Anonymity across conversations and system costs

presents the best trade-off between per conversation cost and anonymity in *Hermes*.

Handling intersection attacks: In the scenario described above, we only considered the anonymity in a given conversation; the cost due to fake conversations (included for protection against intersection attacks) was not considered. Next, we present results that capture the effect of these conversations, which appear at a rate of $\lceil \frac{\mathbb{K}-\mathbb{D}}{\mathbb{D}} \rceil$ (recall Stage 5). The parameter $\mathbb{K}$ determines the level of anonymity provided across conversations as discussed earlier. The value of $\mathbb{G}$ is specific to each user and varies from 5 to 25; we fix an EFF of 25 % based on our previous results. In Figure 3.5a, we change $\mathbb{K}$ by varying $\mathbb{K}_2$ ($\mathbb{K}_1$ is estimated in the first stage of the process for each user as discussed in Section 3.5). We immediately see that $\mathbb{K} \approx 150$ yields the highest anonymity at a reasonable cost. This is only marginally higher than the value of $\mathbb{D}$ (with the chosen EFF value). This implies that, on average, we need only one fake conversation for every real conversation in order to thwart the intersection attack. In Figure 3.5b, we plot the difference in the maximum and minimum number of conversations that the members of Top $\mathbb{K}$ group have participated in. We see that this difference is no greater than 3 at all times; this demonstrates the high degree of anonymity within the Top $\mathbb{K}$ group.

Cost breakdown: Figure 3.5c shows the distribution (across users) of the total costs due

to the various components required by *Hermes*, viz., storage, bandwidth, operational queries, and the computing instance, with GAE. We see that, for about 95% of the users, the total cost is $< \$10$ a month. In comparison, if a compute instance is always active, the cost of this alone would be $>$ \$60 per month. We also see that the cost due to operational queries and the instance are the biggest contributors to the total cost. This is expected since storage and bandwidth are relatively cheap, especially since *Hermes* purges the cloud storage regularly. Operational costs are *relatively* high since storing, retrieving, or even checking for content, incurs a cost [3, 27]. The total cost with EC2 is slightly lower than that with GAE ($< \$9$ for 95% of the users) but the trends in the cost components are similar; we do not plot the results here due to space considerations.

Timeliness vs. cost: In Figure 3.5d, we plot the expected delay incurred in accessing posts (a measure of timeliness) versus the expected (total) cost. We again use GAE. If the length l of each round (recall Section 3.5) is reduced, the operational costs are increased, but the timeliness is improved as well. If instead, we increase l, the timeliness suffers but queries are made less often (to check for content) and thus, cost decreases. Even if the desired expected delay is as low as 5 minutes, the incurred cost with GAE is no more than \$5 per month. With EC2, this cost is even lower (\$3). These results demonstrate that good timeliness is possible with *Hermes*, with fairly low cost.

Timeliness vs. anonymity: Next, we quantify the impact of varying A_1 on timeliness for a fixed l (set to 5 minutes for our experiments). We measure this impact in terms of average costs and the average delay over all conversations in the simulation. In Table 3.1 we show a representative subset of our results that is of interest. As evident, increasing A_1 decreases average cost but increases the average delay of message propagation. From the table we see that when A_1 is decreased

A_1	Avg. Delay (min)	Avg. Ops Cost ($)
1.025	2.8867	5.13
1.05	3.096	2.80
1.10	22.52	2.32

Table 3.1: Cost for various values of A_1

Operation	MS/Req	Bytes Rx	Bytes Sent
Posts links	130	78	4
Check and retrieve update	50	22	35
Add comment	45	42	2

Table 3.2: *Hermes*'s resource consumption on GAE

to 1.05 from 1.10, the marginal reduction in delay is significant; however, the additional reduction is marginal when A_1 is further reduced to 1.025. The cost growth is almost linear. These results suggest that setting A_1 (by default) to 1.05 provides the best trade-off between delay and cost.

3.8 Prototype Implementation

We implement a *Hermes* prototype in Java as an add-on to Facebook. Specifically we use the Facebook front end and a user's profile therein is used for making her public key component available. *Hermes* runs as a middleware and intercepts posts classified as private. Dropbox and Google App Engine (GAE) are used for storage and computation.

Bootstrapping the system: Upon installation, the *Hermes* client requests OAuth 2.0 [42] access tokens from both Facebook and Dropbox and stores these locally for later use. The client crawls the list of the user's friends on Facebook and creates one $ufile$ for each of them on Dropbox. The client also initializes a web-based application on GAE on behalf of the user. When the user is

offline, this application handles requests from members of groups with whom the user has shared private content.

Establishing keys: The *Hermes* client posts a user's public key on her Facebook profile. When the user goes online, the client fetches the public keys of all her friends. *Hermes* uses these keys to establish a pairwise key for each friend using ECDH (see Section 3.6). When the user shares new content, *Hermes* randomly generates a group key. It makes a copy of the group key for each specified group member, and encrypts the copy with the pairwise key corresponding to that member.

Storing information: *Hermes* encrypts private content with the group key generated above; the group key itself (encrypted as above) is included as metadata. Then, the *Hermes* client uploads a file (the content encrypted with the group key), as described in Section 3.4, to the user's space on Dropbox and requests Dropbox for the public URLs for these files.[3] Although anyone with knowledge of this URL can download the stored (encrypted) file, only those who have the requisite keys can view the files.

When the user shares content, the *Hermes* client also invokes a GAE instance and uses the GAE data store to save the encrypted public URL to the file on Dropbox. The GAE instance is provided with the identities of the users in $\mathbb{D}$, for that conversation. All communications with the instance are over HTTPS. For efficiency, the instance shuts itself down after a configurable time has elapsed after creation (10 hours by default).

Retrieving stored private content: When a recipient (say Bob) comes online, the *Hermes* client on his machine fetches the URLs for the compute instances of his friends (listed on their Facebook profiles). The client then queries all of these instances to find if there is any new shared

[3] With typical file sharing on Dropbox, when Alice shares a file with Bob, the shared file is counted towards the storage capacity of both Alice and Bob. In our implementation, public links are simply pointers to Alice's files; the files are then directly accessed by Bob.

data. If new data was shared by Alice with Bob (Bob $\in \mathbb{D}$), the instance shares the URL to the file shared by Alice with Bob's *Hermes* client. The *Hermes* client retrieves the newly shared data and stores it locally. We use OAuth to grant the GAE instance relevant permissions so that it can write to Alice's storage on Dropbox.

Note that fake group members in $\mathbb{D}$ (included to protect anonymity) are also notified of new data. The client devices of such users are then provided a fake group key (with a prefix that indicates that it is fake); the server cannot detect that the member is fake, since the fake group key is encrypted using the pairwise keys. The client of a fake member simply discards all content retrieved with respect to the conversation (both the original posting as well as comments).

Fake conversations: Fake group members are added to conversations and fake conversations are initiated as described in Section 3.5. We set the values of $\mathbb{K}$ and $\mathbb{D}$ to the values determined by our simulations. We disable these features in our prototype evaluations since we do not have real conversations taking place currently.

Commenting: Every group member independently posts her comments (real or fake) to the $ufiles$ in her own storage. When a user wishes to see all comments related to a post, her *Hermes* client initially downloads the contents of all $ufile$ entries marked by the particular conversation ID associated with the post. It then discards fake content (identified with a prefix) and combines the information from the true conversations it is involved in. Thereafter, it discovers new comments by periodically polling $ufiles$ in the cloud storage of the user's friends. The client also posts fake comments in rounds during which it is online, as discussed in Section 3.5.

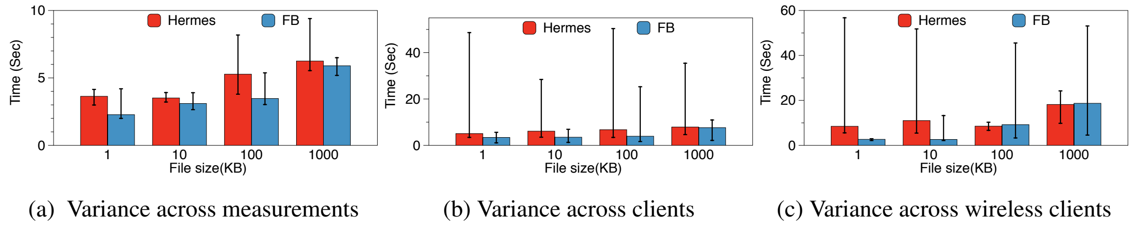
Storage overhead associated with initial shared content: Our implementation uses AES (256 bit key) to encrypt data and ECDH to establish a symmetric key. Specifically, we use the

P-256 curve as defined in [48]. The parameters (such as the curve and p) are defined in P-256 and available to all users.

Hermes adds overhead to shared content in three ways. (1) As described in Section 3.4, each $ufile$ entry occupies 16 bytes for a hash value and $\approx$ 20 bytes for an encrypted URL on Dropbox. (2) Each $ufile$ entry also includes the group key encrypted with the pairwise key of the sender and its corresponding receiver, with information for associating the entry with the receiver and authenticating her. In our implementation, the size of each $password$ tuple is 62 bytes (recall Section 3.6). (3) *Hermes* stores information about the uploaded files and the access tokens for writing to a user's Dropbox account on the user's GAE instance. In our implementation, every post accounts for 440 bytes of space on the GAE instance. In essence, (1) *Hermes*'s storage overhead is a few KB for sharing data of any size; as an illustration, storage overhead is 82, 820, and 1640 bytes for group sizes of 1, 10, and 20 members (including fake), and (2) storage overhead of *Hermes* increases linearly with the composite group size. Note that we expect private groups to be typically small [87, 5].

3.9 System Evaluation

We next demonstrate the efficiency of our *Hermes* implementation. We first evaluate our prototype by comparing the delay incurred in sharing data with *Hermes* to that with Facebook. We share files of different sizes and measure the total delay between when a user shares a file and when a recipient completes receiving that file. To mimic the overhead seen by real users, the receiver program contacts 250 compute instances (250 is the average number of friends on Facebook [71]) to check for new content.



(a) Variance across measurements (b) Variance across clients (c) Variance across wireless clients

Figure 3.6: End-to-end delays on *Hermes* and on Facebook

Fig. 3.6a shows the variance in delays incurred (the minimum, median, and maximum values across 5 trials) in the above experiment. The overhead imposed by *Hermes* as compared to sharing and receiving data on Facebook, especially for delay-sensitive sharing of small files, is within reason (a few seconds). Delays on *Hermes* are higher than delays on Facebook because *Hermes* not only posts the shared content on Dropbox, but also sends the links to these files to the user's GAE instance. Furthermore, *Hermes* uploads and downloads $ufiles$ in addition to the content being shared.

We repeat the experiment on 6 PlanetLab [116] nodes, two on the US west and east coasts, and one each in Europe, Asia, Australia, and South America, and with 10 clients that access *Hermes* and Facebook via WiFi. Figs. 3.6b and 3.6c show the variance in median access times across the PlanetLab nodes and across the wireless clients. We again see that the access times with *Hermes* are comparable to that of direct data sharing on Facebook.

Next, we study *Hermes*'s resource usage on the compute instance. We measure the compute and bandwidth resources consumed by the three *Hermes* client operations that require interactions with the instance: (i) post links to newly shared content to the instance, (ii) serve requests from friends who check if anything new has been shared with them (and download new comments, if any), and (iii) receive the link to a recipient's comment and post it to the conversation. We perform

each operation 1000 times and examine GAE's reports for resource usage. Table. 3.2 shows that the compute time and incoming/outgoing network traffic incurred on average, for each operation is low.

3.10 Miscellaneous Issues

Encryption on OSNs: One can argue that, if the fraction of private content is small, an OSN may allow private encrypted postings. This however will lead to two problems. First, the OSN has to have a means of limiting the volume (and possibly types) of private postings; today there is no clean way of enforcing this. Second, if proper care is not taken by users themselves, encrypted content may appear on their friends' news feeds; this would seem like seemingly unwanted garbage that would reduce the appeal of the OSN for many users. One may conceive the implementation of filters to remove such content, but this is not implemented today.

Accessing content from multiple devices: Users often access OSNs from multiple devices. As long as *Hermes* is installed on all of a user's devices and the user's private key is stored on all of these devices, he can retrieve all content associated with ongoing private conversations. When Bob first accesses content from a private conversation initiated by Alice, (say using a desktop), the content is downloaded onto his desktop. Except for the encrypted content, the *Hermes* client also makes copies of all relevant conversational content (e.g., $ufiles$, retrieved comments) in Bob's cloud storage. If Bob seeks to access the conversation later from his laptop, the *Hermes* client on his laptop retrieves the relevant conversational content from his own cloud storage (syncing). It then checks to see if Alice's cloud storage still has the originally shared post (in our example, the photo that Alice shared) and downloads it if available. As long as Alice maintains the encrypted photo for a few days ($\approx$ 3 days, as shown in Section 3.7), the majority of cases are handled seamlessly

using this approach. If some content is to be retrieved much later, it is handled as discussed earlier in Section 3.4.

Communicating beyond friends: In our discussion thus far, we implicitly assumed that Alice shares content (a photo) with her friends. *Hermes* can be extended to allow the sharing of private content with a subset of OSN users beyond the user's immediate friends. This requires friends of a private conversation to *re-share* the content. In other words, a user's immediate friends act as "proxies" to relay messages to and from that user. We defer a study of this possibility to future work.

Changing group membership: It is possible that an initiator wishes to change the membership of a private group. She may seek to add new members or remove members from a group. Adding new members simply requires the owner to share the group key with such members (as before).

$ufiles$ enable the deletion of members from a group in *Hermes*. Specifically they support key revocation. If Alice wishes to exclude some members from a conversation group, she generates a new group key and pushes these updates to group members via $ufiles$; specifically, she includes the new group key encrypted with the pairwise keys for each user in the updated group and encrypts the list of group members with the new group key.

For example, to exclude Bob from a private conversation, Alice modifies the group key for that conversation and updates her $ufile$ for Dave (with an appropriate update code indicating a key change). When Dave comes online, he obtains the new key (via Alice's $ufile$ for Dave) and re-encrypts his new comments with this key instead. When Bob comes online, he will think that the conversation is over since there is no update in Alice's $ufile$ for him, and his key will not decrypt

content any longer.

Generality of design: *Hermes* can interface with any cloud storage or computation service and has inherent deployment flexibility as discussed earlier. It is also decoupled from assumptions such as the presence of SSL certificates. While we use symmetric key encryption enabled by ECDH-based key exchange, we can use other encryption solutions within *Hermes*. For example, Attribute-based Encryption (ABE) [77] has been considered in prior work [72]; it provides primitives for private content access by only a group of users satisfying an *access structure* $\mathbb{A}$. Using the master key and a set of attributes as inputs, the user then generates a private key that corresponds to the given attributes. *Hermes* can support ABE by including $\mathbb{A}$, encrypted with the initiator's ABE public key, in the tuples entries for every member in $\mathbb{A}$. Any group member decrypts the encrypted metadata included with the posted content, using an ABE private key distributed (possibly using PKI) by the initiator, and obtains $\mathbb{A}$. Now, she can encrypt her comments using the initiator's ABE public key and $\mathbb{A}$. This allows only the users defined by $\mathbb{A}$ to decrypt the content. We defer a study of using other encryption schemes within *Hermes* for the future.

3.11 Conclusions

In this work, we design and implement *Hermes*, a practical, cost-effective, OSN architecture for private content sharing. *Hermes* intelligently uses limited storage and computing resources on the cloud to facilitate timeliness and high availability, while minimizing resource usage. A key property of *Hermes* is that neither the cloud providers nor other friends of a user are able to infer the membership information of a private group. Via an analysis of mined Facebook data and exhaustive simulations, we show that *Hermes* greatly reduces costs compared to alternative solutions while

ensuring the anonymity of the private group.

3.12 Background on ECDH

In our *Hermes* implementation, we use ECDH to enable private content exchange between users. Though other encryption schemes like RSA or ABE can be easily used with *Hermes*, without loss of generality, we will assume its use when describing *Hermes*. In ECDH, every user has two key components. User i's private key component x_i is a randomly selected integer $\in [1, n]$. Users also agree on a point p (which is public) that lies on an elliptic curve. The public key component[4] of user i is $b_i = x_i * p$, where $*$ represents the point multiplication operator defined in [96].

To derive a mutual key with a user j, user i computes $s_{i,j} = x_i * b_j$. Note that b_j $(= x_j * p)$ is publicly available. Thus, users i and j generate the same secret key $s_{i,j} = x_i * x_j * p$, which can be used to encrypt and decrypt messages between i and j (using a cipher like AES). The only information that is made public by a user i is $b_i = x_i * p$. The private key components x_i and x_j are not revealed, and hence, $s_{i,j}$ can only be computed by users i and j. In *Hermes*, we use the P-256 elliptic curve defined in [48].

3.13 Propagating updates via $ufiles$

As discussed in Section 3.4.1, every user maintains a $ufile$ for each of her OSN friends on her cloud storage. We reiterate that the $ufiles$ are *per-friend* rather than *per-conversation*, with every $ufile$ encrypted with the shared pairwise key for the corresponding friend. The $ufile$ contains

[4]The security of ECDH is based on the difficulty of solving the decisional Diffie-Hellman problem. Let P and Q be points on an elliptic curve *s.t.* $x_i * P = Q$, where x_i is a sufficiently large scalar. Given P and Q, it is computationally infeasible to obtain x_i since it is the discrete logarithm of Q to the base P [102].

tuples $< id, c_{url}, up, data >$, where id is a monotonically increasing counter (eventually wraps around), c_{url} is the unique URL pointing to the folder of the owner of conversation c, up indicates the type of update, and $data$ is the data associated with a specific update status (explained later). A user's *Hermes* client creates a $ufile$ for each of the user's friends when the user first begins using *Hermes* (and whenever the user adds a new friend thereafter). The first time that a pair of friends engage in a private conversation, they use the computing instance to exchange pointers to the $ufiles$ they have created for each other. The links to the $ufiles$ are then stored on their respective cloud storage permanently.

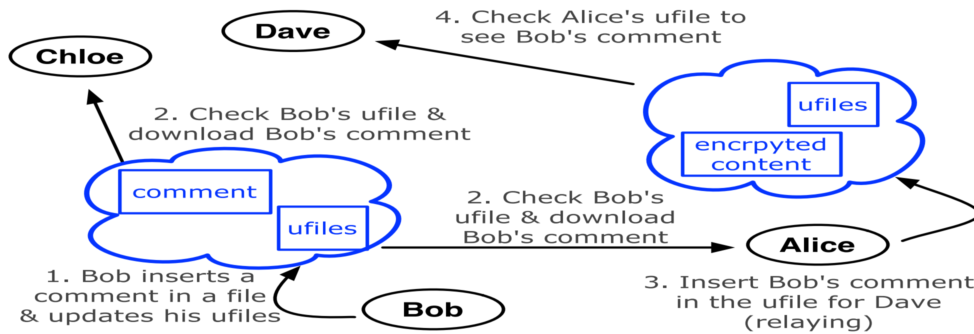


Figure 3.7: Illustration of comment propagation in *Hermes*.

Consider a new bootstrapped conversation between Alice and all her friends. We consider the scenario where Bob wants to post a comment (or reply) to content that was originally shared by Alice.

Step 1: To comment on the content posted by Alice, Bob writes an update to the $ufiles$ that he maintains for his friends Alice and Chloe. Bob's *Hermes* client writes this update only to his $ufiles$ for Alice and Chloe, and not his other friends, since they are the members of the group. This update contains the tuple $< id, c_{url}, 1, link >$, where '1' is an integer code to indicate that a new comment in conversation c is in $link$,which is owned by Bob.

Step 2: When Alice and Chloe come online, they download their respective $ufiles$ from Bob's storage and learn of the new comment in c. They individually retrieve Bob's comment and create new tuples of the form $< id, c_{url}, 2, link >$ in their own $ufiles$ for Bob. Here, '2' is an integer code indicating that they have received the last comment made by Bob in conversation c.

Step 3: When Bob comes online again and his client downloads the corresponding $ufiles$ from Alice and Chloe, it realizes that all his relevant friends have read his latest comment. It then deletes the prior update $< id, c_{url}, 1, link >$ from his $ufiles$; it also purges the corresponding comment from his cloud storage. By doing this, the space occupied by the comment and $ufiles$ do not simply grow over time, thus drastically decreasing *Hermes*'s cloud storage requirements. Upon returning online, Alice and Chloe notice that Bob's original entry is deleted from his $ufiles$. This implicitly tells them that Bob has received their update and hence, they delete their update tuples from their $ufiles$ for Bob.

Step 4: While Alice and Chloe get Bob's comment as above, Dave is not Bob's friend and hence, does not receive it. In fact, Bob does not even maintain a $ufile$ for Dave. To allow Dave to see all the comments in a conversation he is part of (as with Facebook), *Hermes* leverages the fact that Alice is a friend to all group members and incorporates an additional step (shown in Fig.3.7). When Alice notices Bob's update (step 2), she checks whether there exist group members who are not friends with him. For each such member (e.g., Dave), Alice inserts an update tuple $< id, c_{url}, 3, rc >$ in their respective $ufiles$; here '3' is a code for the relaying of a comment. rc refers to the relayed comment included in the tuple. Upon coming online, Dave downloads Alice's $ufile$ for him, finds the comment, and notifies Alice of the receipt of this update. Alice and Dave then purge the associated updates from their respective $ufiles$ (steps 2-4 as before).

Note that the above scheme for distributing comments also works for other types of notifications (e.g., ‘Likes’ on Facebook) by simply having different update codes for different types of notifications.

Chapter 4

ZapDroid

4.1 Introduction

There has been an explosion in the number of third-party smartphone apps that are available and are being downloaded. The Google Play Store has more than 1.3 million apps [41] and a report in [29] states that the number of downloads of apps from the Play Store between May'13 and July'13 alone was about 2 billion. However, after users interact with many such apps for an initial period following the download, they almost never do so again. Statistics indicate that for a typical app, less than half of the people who downloaded it use it more than once [18]; further, 15% of the users never delete a single app that they download [54]. In more general cases, users may only interact with some downloaded apps infrequently (i.e., not use them for prolonged periods). These apps, which are seemingly considered *dead* by the user, continue to operate in the background long after the user has stopped interacting with them. Such background activities have significant negative impacts on the user, e.g., leaking private information or significantly taxing resources such as the battery or network. Unfortunately, the user is completely unaware of these activities. We call

such apps, which are dead from the perspective of the user, but indulge in undesired activities, as "zombie apps".

Our goal: In this work, we seek to facilitate effective identification and subsequent quarantine of such zombie apps towards stopping their undesired activities. Since a user can change her mind about whether or not she wants to use an app, a zombie app must be restorable as quickly as possible if the user so chooses.

The classification of an app as a zombie app is inherently subjective. After an app goes unused by a user for a prolonged period, the determination of whether the app should be constrained depends on whether the app's resource usage during the period of unused is considered significant or whether the app's access of private data is deemed serious. Therefore, instead of automatically categorizing apps as zombie apps, we seek to empower the user by exporting the information that she would need to make this decision. Moreover, the manner in which a zombie app should be quarantined depends on whether the user is likely to want to use the app again in the future (e.g., a gaming app that the user tried once and decided is not interesting vs. a VoIP app that the user uses infrequently). The apps that a user is likely to use again fairly soon should not be fully uninstalled; real time restoration (when needed) may be difficult if the user does not have good network connectivity. We seek to enable users to deal with these different scenarios appropriately.

Challenges: Achieving our goal has a number of associated challenges. First, zombie apps are active (execute) in the background and hence, we need an efficient mechanism to track the foreground and background states of apps; continuous monitoring of apps, as proposed in prior approaches (e.g., [132]), can be too resource-intensive to be practical. Second, we need to monitor how apps use sensitive resources protected by permissions in a lightweight manner. Application-level

implementations are infeasible since Android does not allow one application to track the permission access patterns of other apps. Third, once a zombie app is quarantined, we must ensure that it is not re-activated unless the user chooses to do so. With current approaches, the background activity of apps are constrained only temporarily [68], until they are woken up due to time-outs or external stimuli [10, 28]. Fourth, *ZapDroid* should ensure that a previously-quarantined zombie app is restored quickly if the user seeks to access it; the restored app must be in the same state that it was in, prior to the quarantine. Reinstalls from the Google Play Store can be hard if the network connectivity is poor and hence, should be not be invoked in cases wherein the user may restore the app with high likelihood; further, clean uninstalls can result in loss of application state.

Contributions: Towards achieving our goal and addressing the above challenges, we design and implement *ZapDroid*. *ZapDroid* identifies candidate zombie apps, exports appropriate information to the user to allow her to choose if she wants to quarantine any of them, and based on the input provided, silos a zombie app appropriately. In addition, *ZapDroid* also seeks to ensure that an app will execute again (in the state prior to quarantine) if the user chooses to invoke it, that the app does not crash, or that unwanted error messages do not pop up when an app is quarantined. Specifically, we make the following contributions:

- ***Understanding zombie apps.*** *Undertake a comprehensive measurement study to showcase the unwanted behaviors of candidate zombie apps:* We have conducted a month-long user study wherein we enlist 80 users on Amazon's Mechanical Turk to download an app we have developed. This app identifies which other apps have not been used for the month-long period on the users' phones. Once we identify these apps, we undertake an in-house, comprehensive measurement study to understand the behaviors of these apps when they are not being actively

used. We find that a zombie app on a typical user's phone (the median user in our targeted experiments) could consume as much as 58 MB of bandwidth and more than 20% of the total battery capacity in a day. Further, many of such apps accessed information such as the user's location and transmitted this over the network.

- ***Identifying zombie apps.*** *Design and implement a module within* ZapDroid *to identify candidate zombie apps that are most detrimental to the user's device:* We design mechanisms that are integrated within the Android OS (we make changes to the underlying Android framework's activity management, message passing, and resource management components) to track (i) the user interactions with the apps on her device to identify unused apps, and (ii) the resources consumed and the private information accessed by these apps to determine candidate zombie apps (unused apps that indulge in unwanted activities). The user can check the list of candidate zombie apps and choose to quarantine those she considers to be zombie apps.

- ***Quarantining zombie apps.*** *Design and implement a module within* ZapDroid *to dynamically revoke permissions from zombie apps, or offload them to external storage:* The quarantine module of *ZapDroid* is invoked based on user input. She has to categorize a zombie app as either "likely to restore" or "unlikely to restore"; the two categories are quarantined differently. For the first category, only permissions enjoyed by the zombie app are revoked but all relevant data/binaries are stored on the device itself. For the second category, the data/binaries associated with the app are removed from the device and user-specific app state is moved to either the cloud or to a different device (a desktop) owned by the user; the transfers are made when there is good network connectivity (e.g., WiFi coverage or a USB cable).

- ***Restoring zombie apps.*** *Design and implement a module within* ZapDroid *to restore an app*

with all its permissions if the user desires: We implement a mechanism in the OS that restores a zombie app on the user's device if she so desires. The state of the app is identical to that prior to the quarantine. For the "likely to restore" category of apps, the restoration time is $< 6ms$. For the "unlikely to restore" category, restoration depends on the network connectivity to where the app was stored during the quarantine and is typically on the order of a few seconds.

Note that our approach does not require changes to an external cloud store (for quarantine or restoration); all modifications are made only in the Android OS.

We evaluate *ZapDroid* via extensive measurements on 5 different Android smartphones (from 4 vendors) to quantify its efficacy in achieving its goals. We show that the overhead of *ZapDroid* is low (¡ 4% of the battery is consumed per day) and will not adversely affect user experience. We show that *ZapDroid* saves more than $2\times$ the energy expended due to zombie app activities, as compared to other popular apps on the Google Play Store used to kill undesired background processes; further, unlike these apps, it prevents access to undesired permissions by the zombie apps.

4.2 Understanding Zombie Apps

We begin with an in-depth measurement study that has two main goals: (a) revealing zombie apps, via an IRB-approved user study, and (b) profiling zombie apps to quantify their adverse effects both in terms of resource consumption as well as privacy leaks.

4.2.1 User Studies

We undertake a large-scale user study to identify apps that are installed but not used by real users. Specifically, we have created an app (`TimeUrApps`) and deployed it on the Google Play

Store. We then solicited volunteers on Amazon Mechanical Turk. Once active, our app records the following for a period of thirty days: **(i)** the names of the apps installed on users' devices and **(ii)** the timestamps of all events where an app is pushed to the foreground from the background or vice versa. Specifically, `TimeUrApps` implicitly starts the Accessibility Services [6] upon activation, that in turn, facilitates the collection of this information.

In a nutshell, `TimeUrApps` detects those apps that have been inactive for extended periods. To be eligible for our study, we required that a user must have at least 40 third-party apps installed. This number was motivated by a Nielsen study [39], which reports that the average number of third-party apps on a user's smartphone is 41 (although many users can have a significantly higher number of such apps [18]). A total of 87 users downloaded `TimeUrApps`. We filtered out 7 users that installed new apps just prior to downloading our app, to reach the required app count of 40 (so that they could claim the reward that we promised). This left us with 80 users. The data collected by `TimeUrApps` from the volunteer users' phones was transferred to our servers using WiFi as long as access was available within a five day period (a timer was set each time data was transferred). If no WiFi access was available for 5 days, the data was transferred on the cellular network.

All the apps that were unused by the users for a period of a month are considered potential zombie apps. We select the top 5 categories of unused apps (summarized in Table 4.1) based on this data set and perform a more in depth study. The categories are based on the definitions in the Google Play Store.

The number of apps in various categories that we observed in our dataset is plotted in Figure 4.1a. We see that more than 1,000 unique apps were not used in the one-month test period.

Category	Types of Apps
Games	Role playing and turn-based games
Tools	Memory cleaners and task managers
Productivity	Barcode scanners and cloud drives
Entertainment	Media streaming and ticket sales
Social	Social networks and event planning

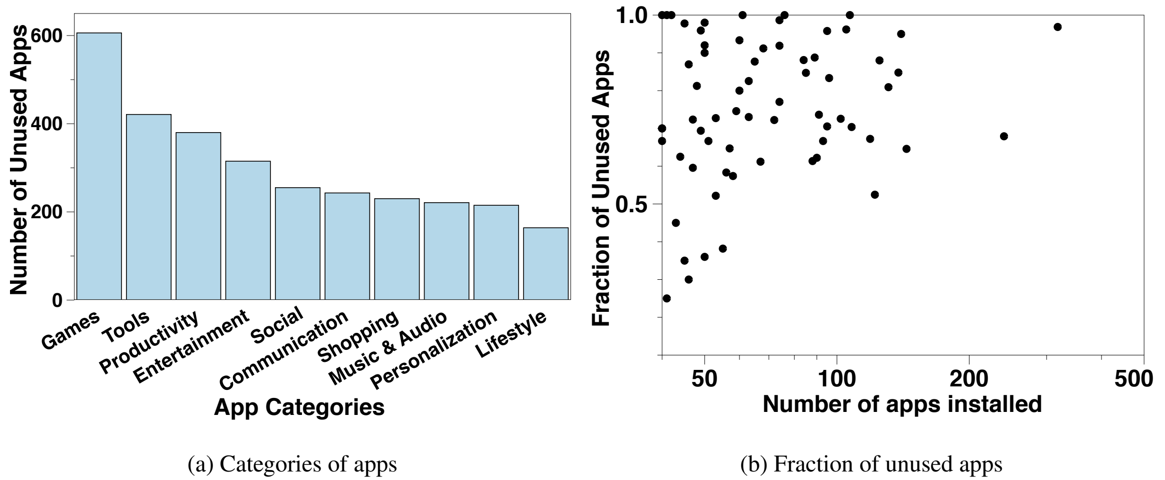
Table 4.1: Types of apps in the top 5 categories.

We also examine whether the number of unused apps on a device depends on the total number of apps installed. In Figure 4.1b, we show the fraction of apps that were never used, for each user, in the month. Interestingly, the results suggest that regardless of how many apps were installed on a phone, a user rarely interacts with more than half of them.

4.2.2 Offline Measurement Methodology

Once we identify unused apps from our user population, we undertake an offline measurement study to determine which of these consume resources on a smartphone while being in the background mode for prolonged periods, or access/leak private information (in this mode). We measure the resources consumed along multiple dimensions: specifically, we look at the battery drain, the network bandwidth consumed, and time consumed on the CPU. Note here that we did not measure the resource consumption on the study subjects' phones since this required active monitoring (rooting the phones), as well as transferring large volumes of data, which users may not agree to.

Scenarios considered: Our measurement study was conducted on five different smartphones, from four different vendors. We have a Moto X (Motorola), a Nexus 4 and a Nexus 5 (LG),



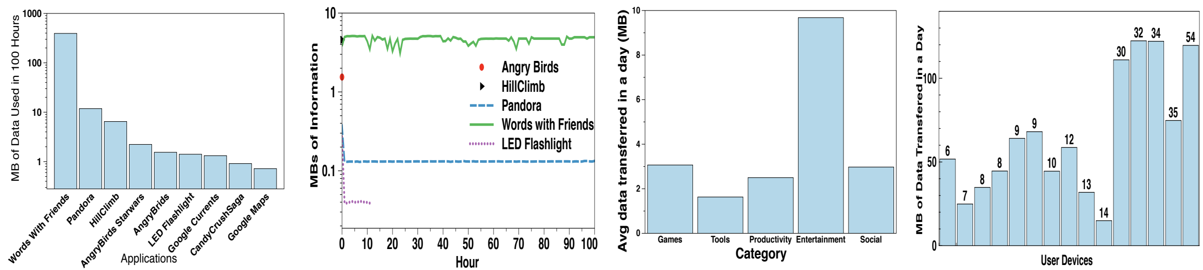
(a) Categories of apps

(b) Fraction of unused apps

Figure 4.1: App statistics from the user study.

a Samsung Galaxy S4, and an HTC One, as our set of phones. We installed our candidate apps on these phones in three ways. *First,* we did a quick examination to determine the individual apps that consumed the highest resources. We then installed these apps in isolation, and let them run in background mode (without user intervention after an initial interaction with each app) for a period of 100 hours. Our goal is to characterize individual app behavior. *Second,* we chose 15 users at random from our 80 volunteers, each with a different number of unused apps (ranging from 7 to 54), and installed all the unused apps from a user profile on a phone. We then executed all these apps in background mode without user intervention, to quantify the collective impact of the apps. *Finally,* we categorized apps (e.g., games, tools), and installed a number of apps (20) from each category on individual phones (5 different phones) to understand the collective "category" level behaviors.

Some nuances: Note that for some of the apps, we had to take additional steps in order to ensure their proper execution. For example, the Facebook app will not execute unless the user has a Facebook account and has logged onto it on the device (many of the users who had Facebook



(a) Traffic from different zombie apps

(b) Network usage per hour

(c) Avg. data transfers per category

(d) Data transfers per user

Figure 4.2: Network transfers by zombie apps.

installed never interacted with it for a prolonged period!). A second example was the Angry Birds game; unless the users finished the first level, the app would not have an initial score to check against other users and would not communicate with a server. In this case, we assume that a user who installed Angry Birds has completed at least one level.

Our rendering in the above cases provides a conservative estimate of resources consumed by the apps; a more complex (and possibly realistic) profile will incur higher resource drain. For example, a Facebook app connected to an account with a hundred friends will expend a higher network bandwidth and CPU cycles for receiving messages and updates, and a higher battery drain, than if connected to an account with just one friend.

Measuring resource consumption: Our smartphones are connected to WiFi over which network transfers occur (we have some limited experiments where we disable WiFi and transfers occur over LTE, as discussed later). Each device under our control is connected to a Man-in-the-Middle (MitM) Proxy, which logs every single IP packet that the device sends out. On the Android devices themselves we: (a) embed a certificate generated by our proxy as a "root certificate" to ensure that the proxy can decode all of the packets sent by the device and (b) install `tcpdump` on

the device to monitor the traffic sent out on the local network (e.g., UDP broadcasts), at the device itself. With these we record the content being sent and received, as well as quantify the volume of network bytes sent.

We used `PowerTutor`[132] to collect information on CPU usage and energy consumption by each app for varying durations of up to 100 hours on all our smartphones. The `PowerTutor` readings are in terms of Joules (energy). We convert the readings to express energy consumption in terms of battery percentage. For example, the Samsung Galaxy S4 has a a rating of 9.88Wh (2600mAh@3.8V [51]), which corresponds to 9.88×3600 J. The percentage of battery power is expressed relative to this rating. Note that `PowerTutor` is not used with *ZapDroid*; it is only used here for the purposes of our measurement study.

Permission access patterns: Since Android does not automatically allow us to log the permissions accessed by an app, we root one of our phones and install *ZapDroid*. This allowed us to log the default permissions that were accessed by these applications (we describe the design and implementation of *ZapDroid* in Sections 4.3, 4.4, 4.5 and 4.6).

Measuring space consumed by zombie apps: As an auxiliary resource, we also measure the disk space consumed by each zombie app on the user's smartphone. It is slightly trickier to measure the space that each app occupies on the device. There are two types of data being stored: (i) *App Binary*: This is user-independent data, which includes the app binary downloaded through the Google Play Store (`.apk`) and any additional data that the application will need for it to function. The Google Play Store puts a limit of 50MB on the former and 2GB on the latter [35]. (ii) *App Data*: This is user-dependent data, which is *not* content created by the user herself but the byproduct of her interaction with the app (an app's private data). An example of the former is photos and of the

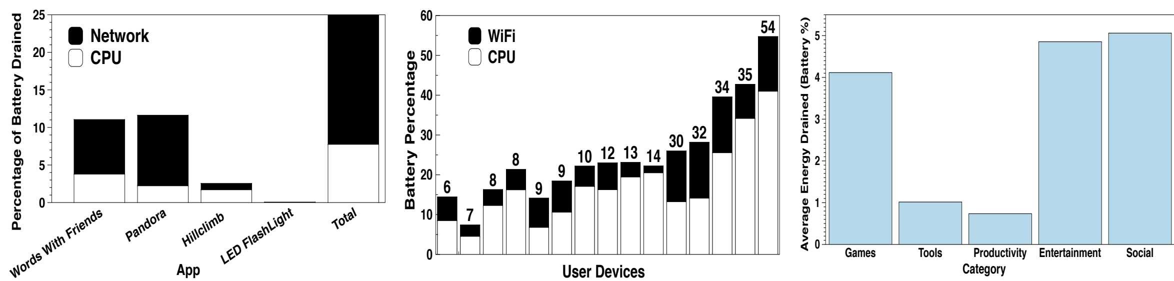
latter is any kind of temporary file that the app writes to for the purposes of recording information. We only focus on the latter because the users are likely to be unaware of how much space the latter occupies. The former cannot be classified as something that the user would want to get rid of.

Unfortunately, our large scale user study did not provide us with the information with regards to the *App data* (the phones were not rooted). Towards this, we distributed our app to 25 volunteer students at UC Riverside, who had rooted Android phones, and gathered this data separately.

4.2.3 Measurement Results and Inferences

In this subsection, we present our measurement results to showcase the impact of zombie apps[1] on a users' device. Simply put, many zombie apps consume a considerable amount of CPU time, network bandwidth, and consequently, energy. In addition, they also access information that can be considered private.

[1] We assume that any app that was not used for the period of the month is a zombie app from the user's perspective.



(a) Energy consumed by zombie apps on a Nexus 5

(b) Energy consumed per user

(c) Avg. energy drain per category

Figure 4.3: Energy consumption by zombie apps.

Bandwidth impact: In Figure 4.2a, we depict the amount of network traffic generated by some of the popular zombie apps in our dataset. Many of these consume bandwidth due to advertisements. Games like *Words with Friends*, in addition, actively try to find new games that are likely to interest the user, and recommend them to the user. These apps therefore consumed an inordinate amount of network bandwidth, due to continuous synchronization with remote servers. Worse, even if the user ignored (as in our case) the app's notifications, it continued to perform the activity in the background. The figure also shows that each zombie app could consume more than 1 MB of bandwidth over a 24-hour period; thus, over the same period, a user with 20 zombie apps could potentially consume 20 MB of bandwidth. Note that most of our 80 volunteer users had at least 20 apps that remained unused for the month.

We point out that if the WiFi is turned off, these communications happen over the cellular network. We verified this to be the case over a 12-hour period. During this time, for example, approximately 38 MB was transferred over LTE due to Words with Friends (similar to WiFi). This hurts the user since mobile operators currently impose limits on cellular data usage.

In Figure 4.2b, we show the network activities of a subset of the apps featured in Figure 4.2a. We see that while the network traffic from some of them (HillClimb, LED Flashlight, and Angry Birds) eventually plummets after being moved to the background, two of them (Pandora and Words with Friends) continue to exhibit network activities *long* afterwards. In addition to continuous synchronization with a remote server (as with Words with Friends), Pandora also sends out broadcast UDP traffic on the local network to look for other devices.

In Figure 4.2c, we show the average data transferred per app in each popular category. We see that, on average, each app transfers more than 2 MB of data per day; on average, an app in the

entertainment category transfers the most data in the background ($\approx$ 10 MB).

Next, we randomly picked 15 of our 80 Mechanical Turk users, collect the list of all zombie apps seen on each of their phones, and install those apps on one of our in-house smartphones (Samsung Galaxy S4). We interact with each such zombie app for a short period of time and then push it to the background. The app executes in this mode without user intervention for 24 hours. The network transfers due to each user's zombie apps are shown in Figure 4.2d. We observe that the collective group of zombie apps on some users' phones could transfer more than 100 MB of data per day.

CPU consumption: We find that in many of our zombie apps, CPU spikes caused by the apps correlate with the network transfers. There are, however, some zombie apps that continuously consume CPU cycles in the background, even when not indulging in network activities. One such app is HillClimb, which, as revealed in our `strace`-based [57] inspection, updates its internal database that stores cached advertisements; the app also records the levels that have been completed by the user. Since this resource expenditure is implicitly captured in our energy results, we do not elaborate here due to space constraints.

Energy drain: In Figure 4.3a we depict the battery drain due to both network and CPU activities by the same popular zombie apps (as in Figure 4.2a) on a Google Nexus 5. This smartphone has a battery capacity of 2300 mAh at 3.7V. The results show that just 4 of the zombie apps consumed about 6% of the battery capacity per day ($\approx$ 25% of the battery over the 100-hour period). This amount of energy (over the 100-hour period) could have powered almost 2 hours of audio and video playback, or an hour of voice call, according to the measurements in [81]. Further, note that this measurement was conducted over WiFi. If it had been over the cellular network instead, the

energy consumed could have been three times as much, or even more [98]. Finally, the collective impact of a larger number of such zombie apps (as is the case on our volunteers' phones) could be even worse, as we show next.

In Figure 4.3b, we see that over the 24-hour period, the zombie apps installed on a typical user's phone consumed about 24% of a Galaxy S4's battery power on average (median of about 22%). In the figure, the numbers on top of each bar indicate the number of zombie apps in the particular user profile. Finally, in Figure 4.3c, we present the average per-app energy consumed, in each of the popular categories. We observe that per day, in some of these categories, a zombie app could consume up to 5% of the battery, on average. With many such apps, the drain could be even more significant.

Use of undesirable permissions and privacy leaks: Next, we install *ZapDroid* and examine the types of information accessed by zombie apps. From Figure 4.4a, we observe that almost all zombie apps access the phone's state which allows them to get a unique identifier associated with the device. This is mostly used for advertisement delivery and for tracking the end user. However, a zombie app can use this permission to track who a user may be calling [66]. In addition, most

Permission Bit	HillClimb	Candy Crush	Words With Friends	Pandora	Guess the Emoji	Angry Brids	Clash of Clans	LED Flashlight
Record Audio								×
Location	×	×	×		×	×		×
Read Phone State	×	×	×	×	×	×	×	×
Start Any Activity	×			×				
Recv Msg from GCM	×	×	×			×		
Change Network State			×	×	×	×	×	×
Internet	×	×	×	×	×	×	×	×
Access Network State	×	×	×	×	×	×	×	×
Broadcasts	×							×
External Storage	×		×		×	×		×

(a) In the background

Permission Bit	Productivity	Tools	Entertainment	Shopping	Communication
Camera	X				X
Contacts				X	X
Location	X	X	X	X	X
Call Logs		X			
Read Phone State	X	X	X		X
Change Network State	X		X		X
Start Any Activity	X	X		X	X
Recv Msg from GCM		X	X	X	X
Broadcasts	X	X	X	X	
Internet	X	X	X	X	X
Network State	X	X	X	X	X
Battery Stats		X			
External Storage	X	X	X	X	X

(b) In various categories

Figure 4.4: Permissions used by zombie apps.

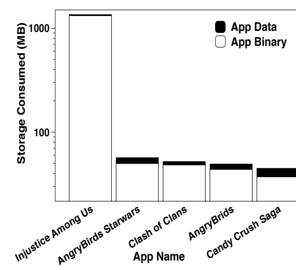


Figure 4.5: Storage consumed by zombie apps.

zombie apps requested access to the user's location, or a permission to modify the network connectivity (e.g., switch access points with WiFi, or disconnect from a network) of the device. Note that *all* the zombie apps shown accessed the Internet in the background. The part of the figure that is shaded, indicates permission bits that leak privacy; we verified that some of this information was actually communicated over the network (e.g, HillClimb accessed and transferred the unique device identifier – Read Phone State – over the network).

Next, we choose 20 random zombie apps from each of the top 5 categories across all users; this implicitly causes more popular zombie apps to be picked in each category. If a majority of the zombie apps chosen in a category accessed a permission, we mark the permission as "being accessed". The results are shown in Figure 4.4b. Again, the shaded region corresponds to information that can be considered as private. The above results demonstrate that there is a significant risk of privacy leakages when a user has zombie apps running in the background on her phone.

Storage Consumption: Finally, as discussed, with our smaller-scale study, we seek to quantify the storage consumed on users' smartphones due to zombie apps. Our measurements, reported in Figure 4.5, reveal that the amount of space taken up by an app is dominated by the size of the binary (and additional user-independent data required by the app) and not the data created by the user or via user interactions. Thus, if storage is a major concern for users, stopping the execution of zombie apps is inadequate by itself.

4.3 An Overview of ZapDroid

As mentioned earlier, the primary goal of *ZapDroid* is to eliminate the adverse effects of apps that are downloaded onto a user's device but are not actively used for prolonged periods

of time, by effectively quarantining them. However, *ZapDroid* also seeks to quickly restore such quarantined apps, if the user wishes to interact with them at a future time. In this section we provide a brief overview of the component modules in *ZapDroid* (details in Sections 4.4, 4.5 and 4.6), and list some of the auxiliary design goals that we seek to achieve.

Detecting and Profiling zombie apps: The detection and profiling module monitors all apps on a user's device to determine if a user is actively interacting with each of them. Apps with which the user does not interact for prolonged periods (a parameter that is set by the user; the default is set to 1 week in our implementation) are *candidate* zombie apps. The module then (i) tracks the resource consumption due to these candidate apps and (ii) logs the permission bits accessed by them. This allows zombie apps to be identified.

User input for quarantine: A rank ordered list of the zombie apps (based on either a resource consumption or permission access criteria) identified by the detection/profiling module is presented to the user upon request, via the front end of *ZapDroid*. The list also contains a summary of each zombie app's activity; it displays a table showing the CPU usage (in seconds), network usage (in KB/MB), battery consumption (in %), storage consumed (in MB), and the permission bits accessed by each zombie app. The user may sort the list based on any metric of her concern and tag a subset of these zombie apps to be quarantined. In addition the user may declare these zombie apps (individually) as ones that he is likely to use in the near future ("likely to restore") or as ones that she will not be using again in the near future ("unlikely to restore").[2]

Quarantining zombie apps: Based on the user's input as above, the quarantine module of *ZapDroid* seamlessly quarantines the chosen zombie apps. Upon quarantine, a zombie app does

[2] While we have implemented a front end for the complete *ZapDroid* system, our UI (user interface) is rather rudimentary at this stage.

not consume any more resources. Depending on whether the zombie app is tagged as "likely to restore" or "unlikely to restore", *ZapDroid* retains it on the device (in the first case) or moves the zombie app along with its associated data to the cloud or to another form of free storage (in the second case).

No free lunch: By choosing to quarantine an app, a user essentially "removes" the app from her phone (albeit temporarily). Thus, any updates that may change features of the app are not automatically seen by the user on her device. The user may have stopped using the app because of the absence of some features that may be incorporated by the update; now, she must get to know of these from an external channel (e.g., a website). Second, if interactive apps (e.g., Skype) are chosen for quarantine, external messages using that app are not received by the user with the current *ZapDroid* implementation, unless she explicitly restores the app (we discuss how *ZapDroid* may be modified to overcome this issue in Section 4.8). Given these factors, we do not automatically quarantine any app with *ZapDroid*; instead, we let the user decide which apps she would like to *virtually* remove from her smartphone.

User input for restoration: If a user chooses to use a previously quarantined zombie app, she can simply visit the front end of *ZapDroid* and 'untag' the app (from being a quarantined zombie app).

Restoring apps chosen for quarantine: Based on user input as above, the restoration module of *ZapDroid* returns the app to the exact same state that it was in, prior to quarantine. It removes any restrictions that were previously placed on the app by the quarantine module. The process of restoration for "likely to restore" apps just involves re-enabling their permissions. To restore an "unlikely to restore" app, the restoration module downloads the app from where it is stored

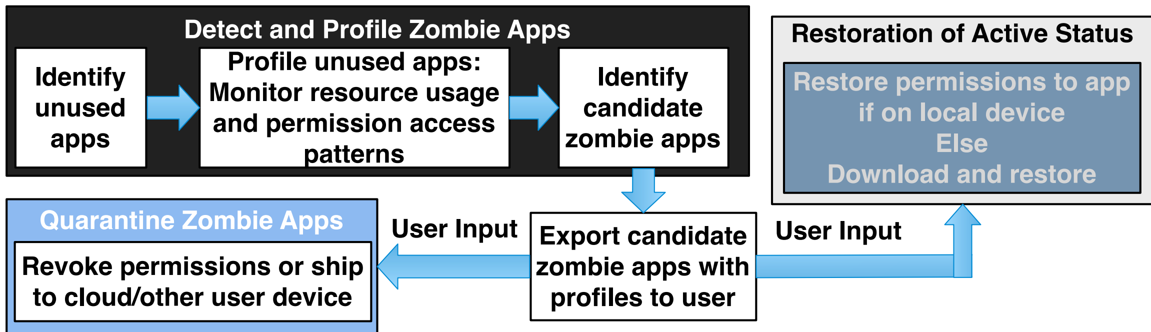


Figure 4.6: High-level architecture of *ZapDroid*.

and subsequently re-enables its permissions. If the app was stored in the cloud, it is downloaded via WiFi. If it was stored on a different user device (e.g., a desktop), the user might have to physically connect her smartphone to the device.

Remark: ZapDroid exclusively resides on the user's smartphone. No changes are needed to the store to which an "unlikely to restore" app is moved. The only requirements are that `put` and `get` interfaces be implemented by the store (as is commonly the case).

Auxiliary goals of *ZapDroid*: Next, we list some of the auxiliary goals that we seek to achieve when we design and implement *ZapDroid*.

- ***ZapDroid* should be lightweight:** *ZapDroid* should not consume significant resources on the user's smartphone. Otherwise, it defeats the purpose of improving user experience (which is one of the primary reasons why *ZapDroid* detects and quarantines zombie apps). The quarantine process should not impact device performance (e.g., offload to cloud when good WiFi connectivity is available), and the process of restoration should be quick.

- **Compatibility:** *ZapDroid* should be usable on *most* Android devices, if not all. In other words, it must not be limited to working on a specific vendor's device(s) or on a particular version of Android.

- **Security:** *ZapDroid* should not result in an escalation in privileges for any application. If such escalations are allowed, potentially new unknown vulnerabilities could arise; to avoid such cases, we set this as an implicit design requirement.

In Figure 4.6, we show the high-level architecture depicting the interactions between the different modules of *ZapDroid*, and in Figure 4.7, we show where the various components of *ZapDroid* are implemented within the Android ecosphere.

4.4 Identifying and Profiling Zombie Apps

The first module of *ZapDroid* detects unused apps. It then monitors the resource consumption and permission access patterns of these unused apps. Any unused app that consumes resources or accesses permissions is labeled a candidate zombie app.

Finding Unused Apps: To find unused apps, *ZapDroid* essentially monitors the apps with which the user interacts. Third-party apps with which the user does not interact for prolonged periods (the value is set by the user via *ZapDroid*'s front end) are classified as unused apps. Below, we describe the process in more detail.

Detecting foreground apps: An app is considered as being used if it is executed in the foreground. To detect apps that run in the foreground *ZapDroid* uses assistive technology that is part of Android. [3] *ZapDroid* adds an accessibility service (upon install) that customizes the wake up trigger by subscribing to the event type TYPE_WINDOW_ CONTENT_CHANGED. This event type notifies a user space function (see Figure 4.8a) whenever a new window, menu, or activity is launched. Similarly, whenever a new app is brought to the foreground or when the screen is locked,

[3] We could have used Activity lifecycle callbacks towards identifying candidate zombie apps; instead, we simply reuse assistive technology that was part of `TimeUrApps`. We observe that the overheads are similar (minuscule) with both approaches (results omitted due to space constraints).

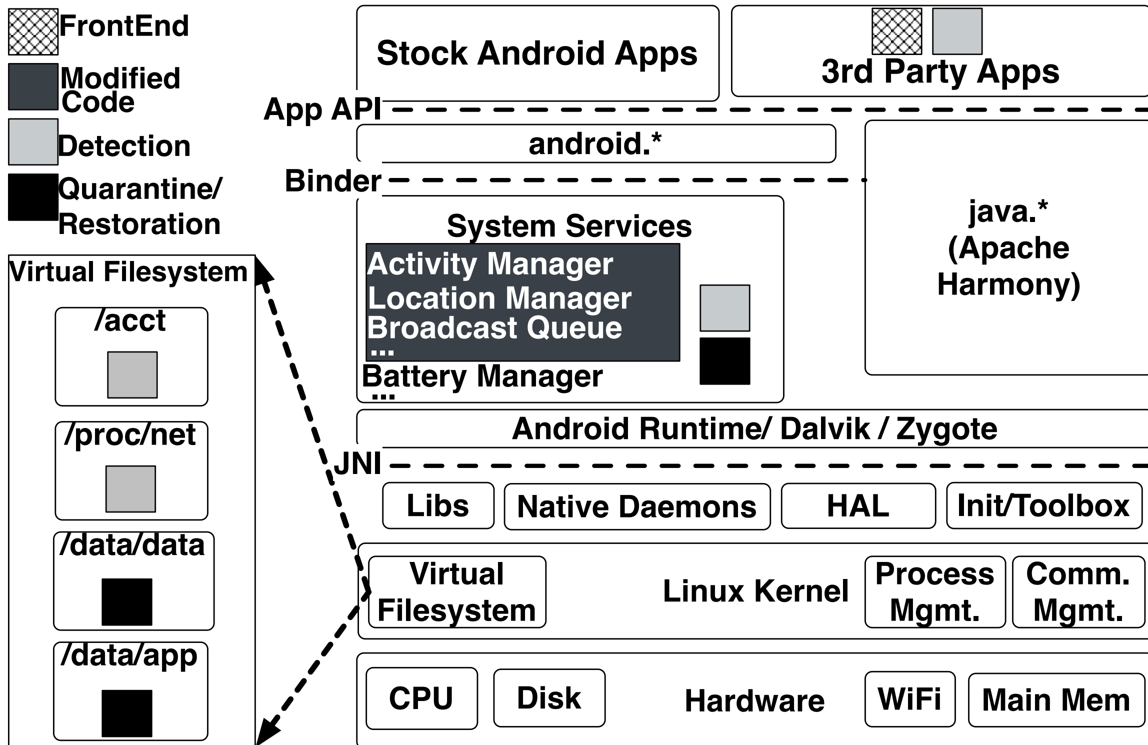


Figure 4.7: Modifications made to the Android OS.

the accessibility service notifies the user-space function. The user-space function records all of these events in a user file that is created upon install (this file contains a list of all third-party apps and is populated upon install). Periodically, *ZapDroid* parses the file to examine those apps that were not activated for a duration greater than the user-specified period. These apps are then categorized as the candidate zombie apps. We wish to point out here that since *ZapDroid* is partly implemented in Android's Linux kernel, the accessibility service used as above is enabled by default (users do not need to turn this service on).

Lightweight implementation: To optimize for space and run time, *ZapDroid* remembers the apps by their userID aka UID (instead of package name). This enables the use of the more efficient SparseArray implementation of Android. *ZapDroid*'s approach is not based on polling (for

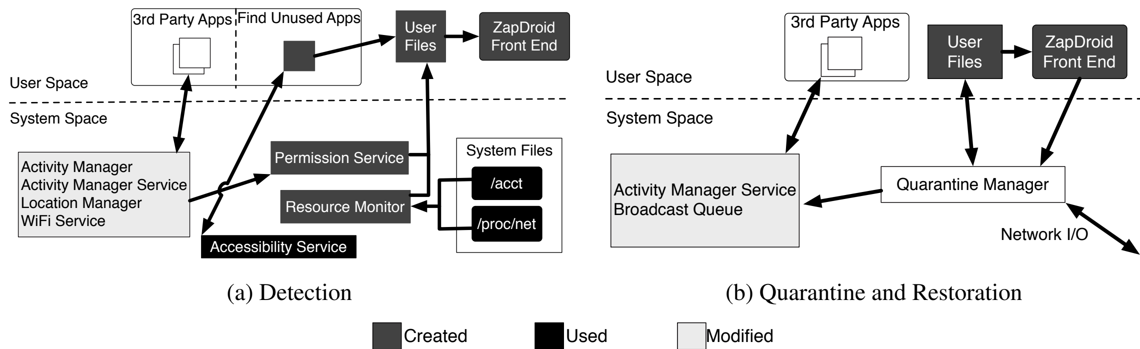


Figure 4.8: *ZapDroid*'s modules.

example, Android's activity manager can be polled). This unnecessarily consumes resources which is not in the user's best interest. We also do not use Android's logging system (one can query this using `logcat` but this requires superuser privileges in newer versions of Android) or implement a modification of Android for this purpose, since we wanted to use this component as an app for use in our previously described user study.

Measuring resource consumption: *ZapDroid* seeks to monitor the resources consumed by apps if they are unused by the user for a short duration (default of 2 days) to determine if they are candidates for being resource heavy zombie apps. It seeks to monitor app-level resource consumption in terms of CPU usage, network traffic, battery, and storage in a lightweight manner. The measured values should be periodically recorded and made available to the user via *ZapDroid*'s front end. Below, we discuss how *ZapDroid* achieves these monitoring goals in more detail.

Monitoring network usage: ZapDroid uses the Android-3.0 Linux kernel netfilter module `qtaguid` to track network traffic based on the UID of the owner process (which is the app's unique identifier). The output of this module is written to the file (`/proc/net/ xt_qtaguid/stats`). *ZapDroid*'s Resource Manager component (see Figure 4.8a) checks this file every 24 hours (it uses Android's Alarm Manager to set the timers) or whenever the device is rebooted (*ZapDroid* is notified

by the Broadcast Receiver) and copies the relevant content with regards to unused apps to a user file. Specifically, it records the total numbers of bytes sent and received by each unused app in that period.

Monitoring CPU usage: To monitor CPU usage, *ZapDroid* leverages Android's `cgroups` (Control Groups). To determine the CPU ticks consumed by a particular app *ZapDroid* reads the file `/acct/uid/<uid_of_app>/cpuacct.usage`. Note that there is a file per UID in this case; the single entry in each of these files is then copied to the user file discussed earlier.

Determining battery consumption: To compute the energy consumed by an unused app, the Resource Monitor uses the linear scaling model (based on CPU ticks consumed and network traffic transferred) used in Android's Fuel Gauge [12] but at coarser time periods of 24 hours. The results are then recorded in the user file.

Determining storage: The Resource Monitor in *ZapDroid* calls the `du` command to measure the disk usage of an unused app. It essentially measures the space consumed by three folders (recursively): 1) `/data/data/ <pkg_name>/` (the application's data on the flash), 2) `/data/data/<pkg_name>/lib` which is a soft link to the app library (the app libraries), and 3) `<External Media>/Android/<pkg_name>/` (any data on external storage such as an SD card).

Remarks: We do not use approaches that rely on `iptables` [15] or `cgroups` [37] for tracking network traffic since we found that these approaches resulted in higher energy overheads without really offering any additional benefits. We also did not use Fuel Gauge (or Power Tutor [132]) directly since we found that they did not account for UDP traffic that was transferred by some of the zombie apps (e.g., Pandora).

Listing 4.1: Notifying Permission Service

```
//ActivityManager.java
public static int checkComponentPermission(String permission, int
   uid, int owningUid, boolean exported) {
... //Cases where the permission can be rejected
NotifyPerms(permission,uid)
return PackageManager.PERMISSION_GRANTED;
...
```

Tracking apps' permission access patterns: The permission access patterns of an app are not exposed to any user-level tool (or even a system-level tool unless it is enforcing the permission). Thus, *ZapDroid* requires modifications to components of Android to track the permissions accessed by zombie apps. This information is then provided to the user via the front end.

Tracking permissions invoked by unused apps: To track the permissions accessed, *ZapDroid* includes a new system-level service in Android called the Permission Service. It also executes a modified version of Android's Service Manager to start the Permission Service at boot time. The Permission Service does not have privileges to make modifications to any other component(s) in the OS and executes as an isolated process. Our modifications essentially cause the Android's service manager to send an asynchronous message to the Permission Service whenever it grants a permission to an app. The messaging does not block the service manager process from acting on the permission, therefore it does not introduce any significant delays. The message consists of the UID of the app invoking the permission, the permission, and the resource that the app is trying to

Listing 4.2: Registering a callback

```
mCallBack = new IRemoteDataCallback.stub(){
      @Override
      public void addApp(String appN, int uid){
       mNewHandler.sendMessage( mNewHandler. obtainMessage(
          QR_ADD, appN, uid) );
       } ...
      mData.getOnChange(mCallback);
```

access (if any). Note that the messaging is essentially an IPC (inter-process communication) mechanism. It is implemented using AIDL (Android Interface Definition Language) [7]. Note also that the asynchronous messages are "one-way" (from the service manager to the Permission Service).

To enable permission tracking, we had to modify many files in the Android Framework[4] (wherever permissions were being enforced). Our modifications are best illustrated by the code snippet (Code 4.1) which corresponds to one such modification. Specifically, the new function `NotifyPerms` notifies Permission Service using a non-blocking asynchronous message about access to a permission, being granted by the Android OS.

Lightweight implementation: We do not use information flow tracking as in TaintDroid [90] since the overhead of doing so is quite large. We do not require information at that fine a granularity. When implementing IPC, we use AIDL since we found this to be the most efficient approach; details are provided in Section 4.7.

[4] Note that Android requires apps to create Java wrapper methods to enable native code (C/C++) to interact with its API. These wrapper methods enable Android to enforce the permissions as with regular Java apps and does not require special consideration for native code [93]. Thus, *ZapDroid* is able to track both Java and native apps.

Listing 4.3: Modifications to AMS

```
public int startActivityIntentSender(... IntentSender intent ...
   String resultWho ...) {
if(!isAllowed(resultWho,intent)) // our addition
      throw new SecurityException("Not allowed to run app " +
         resultWho);
...
```

4.5 Quarantining Zombie Apps

Zombie apps can belong to one of two categories depending on user input viz., "likely to restore" (Category L) or "unlikely to restore" (Category U). For Category L, quarantine involves revoking permissions from the zombie apps and preventing them from consuming resources; the apps are however, retained on the user's smartphone to ensure a quick restoration when needed. For Category U, primarily to save on storage in addition, *ZapDroid* moves the app binary and any associated data from the smartphone to remote storage. These functions of *ZapDroid* are performed by a Quarantine Manager which is implemented as an unprivileged Android system service.

User Input: When a user indicates that a zombie app should be quarantined using *ZapDroid*'s front end (therein, also indicating its category), *ZapDroid*'s quarantine module is invoked. The user, using the front end, can also indicate the external storage to which a zombie app is to be moved if it belongs to Category U.

Quarantining "likely to restore" zombie apps: The Quarantine Manager of *ZapDroid* essentially kills a Category L process that is executing in the background. It also prevents other

apps or processes from communicating (and thus re-initializing or waking up) this app.

Killing the zombie app: ZapDroid leverages Android's Activity Manager's "`am force -stop`" command to kill the chosen zombie app.

Keeping the zombie app in the inactive state: Inactive apps can potentially be activated by messages from deputy apps or cloud services like GCM (Google Cloud Message service). In an extreme case, an app may itself try to overcome being force stopped by signing up for such activation messages.

To prevent such messages from reaching a zombie app killed by the Quarantine Manager, we make modifications to the underlying Android system. Specifically, we create a hook in Android's Broadcast Queue's (BQ) source code and one in its Activity Manger Service (AMS). The hook in the BQ registers a callback from the Quarantine Manager (Code snippet shown in Code 4.2). The `mCallback` function executes in the Quarantine Manager but with the privileges of BQ; the manager sends the BQ a message whenever, a new Category L zombie app is identified, providing the details of that app. This information is stored locally in the BQ. The hook is again activated each time a new message is dequeued from the BQ, to be sent out. If the message is associated with a Category L zombie app (as indicated in the stored record), the hook suppresses the message.

The hook in AMS fulfills a similar purpose. It registers a callback from the Quarantine Manager which again, provides the AMS with a list of zombie apps. Whenever a new activity is initiated, *intents* may be used to invoke apps that are currently inactive. The hook checks the local record and simply suppresses intents meant for Category L zombie apps. The modifications to the AMS are depicted in Code 4.3; the hook simply throws a security exception whenever the target app is a zombie app.

Remarks: We point out that a user-level *swipe* away of an app (switching to another app) does not essentially kill the app [65]. Further, we emphasize that `am force-stop` by itself does not suffice, since the app can be woken up as discussed above.

Quarantining "unlikely to restore" zombie apps: For a Category U zombie app, the Quarantine Manager first kills the process associated with the zombie app; here, the process is identical to that used for Category L zombie apps. Next, it deletes the binary associated with the zombie app, as well as compresses and ships the data to an external storage of the user's choice (indicated using the front end). We discuss why the binary can be deleted later; based on user preferences, *ZapDroid* could also simply delete the user data (this is a clean uninstall of the zombie app).

Shipping data to external storage: For simplicity of discussion, let us assume that there is network connectivity to the external storage where the user wants to store the data associated with the quarantined zombie app. Note that the external storage can be a USB stick, an alternative computer belonging to the user, or even cloud storage (where the user has an account). The only two APIs that we require of this storage are: 1) a PUT primitive which either returns a URI (Uniform Resource Identifier) or allows the specification of a URI and, 2) a GET primitive that allows the retrieval of a previously stored object based on its URI. Using the PUT primitive, the Quarantine Manager, simply puts the user data associated with the app in the external storage, and logs the URI in the previously constructed user file (recall Section 4.4).

Why delete the app binary? We delete the binary since it can be retrieved, when need be, from the Google Play Store. If the zombie app is restored, the data is retrieved from the external storage. If the binary has not changed at the time of restoration, it can simply function with the re-

trieved data. A binary change is equivalent to an upgrade process, and can thus seamlessly function with the user's data. Thus, we see no point in storing the binary in the user's external store (We verify that this works correctly in Section 4.7).

4.6 Restoring Zombie Apps

The restoration of a previously-quarantined zombie app, is also primarily handled by the Quarantine Manager. The goal here is to return the quarantined zombie app to the same state it was in (or an upgraded version if a binary is restored in the case of Category U zombie apps), prior to the quarantine. In a nutshell, the Quarantine Manager simply reverses the procedures that were invoked during quarantine.

User Input: When a user seeks to restore an app, she visits *ZapDroid*'s front end, and explicitly removes the chosen app from the list of apps to be quarantined. This automatically triggers the restoration functions within *ZapDroid*.

Restoring "likely to restore" zombie apps: With Category L zombie apps, the Quarantine Manager, essentially invokes the callback function to inform both the hooks implemented in the BQ and in the AMS, that the zombie app to be restored must be removed from their local records. This essentially now causes the BQ and AMS to forward broadcast messages and intents respectively, to the restored app. Note that the user input, will automatically launch the app.

Restoring "unlikely to restore" zombie apps: With regards to Category U zombie apps, *ZapDroid* checks for connectivity to both the external store and the Google Play store. If the check passes, it retrieves the URI associated with the zombie app from the user file (where it had stored the information before). The URI is then passed to the external store, to retrieve the associated objects.

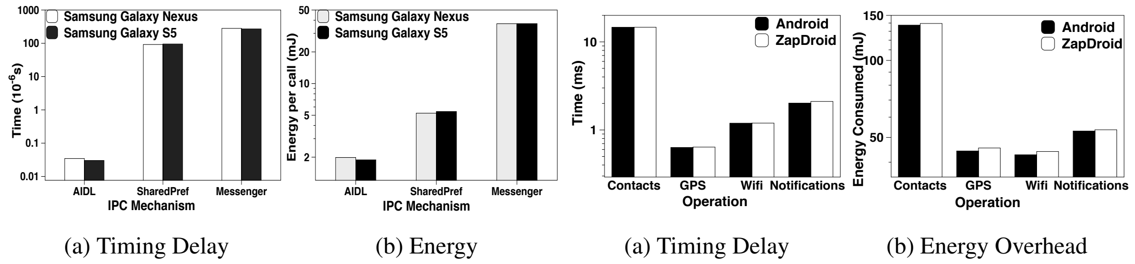
The binary is downloaded from the Google Play Store and installed, in parallel. After the install is completed, the user data is placed in the appropriate directories (this information was implicitly recorded during quarantine).

4.7 Evaluation

We have a complete implementation of *ZapDroid* and evaluate it along multiple dimensions. We first validate our choice of the IPC mechanism in the detection module. Next, we show that *ZapDroid* only introduces very light overhead. Finally, we compare its resource thriftiness compared to two apps on the Google Play store that kill background processes.

Validating the choice of AIDL to implement IPC: The Permission Service receives inputs from components such as the WiFi Service or the Location Manager (see Figure 4.8a). These inputs indicate the permissions accessed by an unused app with respect to that service. IPC (message passing) is used to deliver these inputs. The IPC mechanisms have to be lightweight; they must not introduce delays in the sending process nor result in high CPU usage. We find that using Android's Binder framework, which is based on AIDL (the Android Interface Description Language) best satisfies these requirements.

We compare the performance of the above approach (labeled AIDL), with the following alternative approaches: (i) using the Android's messenger framework (built on top of the Binder framework) [8] and (ii) using shared preferences [53], wherein the Permission Service is notified whenever one of the aforementioned services makes a change to an underlying shared file (stored on the SD card). In Figures 4.9a and 4.9b we depict the delays incurred in delivering the message (referred to as "timing delay"), and the energy overheads with the different approaches. We conduct



(a) Timing Delay (b) Energy

Figure 4.9: Comparison of IPC mechanisms.

(a) Timing Delay (b) Energy Overhead

Figure 4.10: Permission Service overhead.

these experiments on the Samsung Galaxy phones since with these phones, we were able to explicitly remove the battery and measure the energy using a Monsoon power meter [36]. The results demonstrate that the AIDL approach provides a three orders of magnitude advantage in terms of timing delay, and more than 60% less energy per call, compared to the alternatives. This is because, with the alternatives, there is either the overhead of using an abstraction built on top of AIDL or the overhead of using the shared media (SD card) as the medium for implementing message passing.

Remarks: One could potentially use named pipes [4] for IPC, but this depends on the command `mkfifo` which is not compatible with the FAT32 file system (the SD card will have to be formatted using an alternative file system). In addition this would have required us to use JNI (Java Native Interface).

Overheads with *ZapDroid*: To demonstrate that *ZapDroid* is lightweight, we quantify the timing delay and energy overheads incurred due to the different components of *ZapDroid*. We compare the timing delay and energy expenditure with *ZapDroid*, built on the Android version android-4.3_r1, with that in an unmodified install of the same version (no *ZapDroid*). In the following, each experiment was performed 100 times and we ensured consistency.

Detection overhead: First, we examine the detection module of *ZapDroid*. Specifically, we examine the overheads incurred due to notifications sent to the Permission Service. We consider

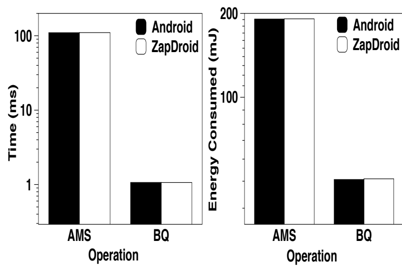


Figure 4.11: Quarantine overhead

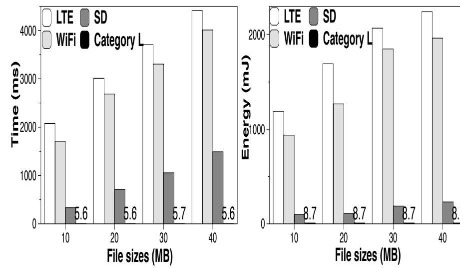


Figure 4.12: Restoration overhead

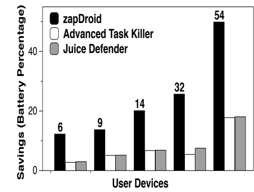


Figure 4.13: Comparing *ZapDroid* with other solutions

permission access by unused apps, to four different services viz., the contact list, the GPS, WiFi and the notification manager. We see in Figure 4.10 that the increase in overhead is extremely low (¡ 4% with regards to energy and ¡ 1% with regards to timing delay).

Quarantine overhead: Next, we quantify the overhead due to the modifications made to the BQ and AMS to enable quarantine. Specifically, (i) we launch an activity, *and separately* (ii) send a broadcast with the two systems. We present the observed timing delays and energies in Figure 4.11. We observe that the increase in timing delay and energy due to the Quarantine Manager is $< 1\%$ for both operations.

Note here that when quarantining Category U apps, in our implementation, we ship the user data either to Dropbox or to an SD card; in the former case, we ship the data when the user is on a WiFi connection and his device is plugged into a power outlet to eliminate bandwidth/energy costs.

Restoration overhead: Finally, we measure the overhead incurred in restoring a previously-quarantined zombie app. We examine the overheads due to the restoration of both Category L and Category U zombie apps. Specifically, we look at the restoration delay, and the energy expended

due to restoration. The results are shown in Figures 4.12a and 4.12b. In these figures we plot the overhead with Category L zombie apps, and with three cases associated with Category U zombie apps. We observe that in comparison with Category U zombie apps, as one might expect, the overheads with Category L apps are minuscule; the timing delay is on the order of 5 milliseconds, and the energy overhead is about 8.7 mJ (these bars are almost not visible).

With Category U apps, where this overhead is more pronounced, we consider three cases. In one case, both the binary and the user data are retrieved from the device's external SD card. In the other two cases, the binary is retrieved from the Google Play store, while the data is restored from the user's Dropbox, via WiFi and LTE, respectively. We observe that both the timeliness and the energy are affected because of network transfers. In comparison with restoring the zombie app from the SD card, the overheads are approximately two orders of magnitude higher. However, even with LTE downloads, the delay incurred is less than 5 seconds with a binary plus data size of 40 MB. The energy consumed in the worst case is below 2.5 Joules (which translates to 0.007 % of the battery capacity on the Samsung Galaxy S4 phone).

Verifying that restoration results in a functional app: Next, we verify that the restoration of a Category U zombie app (after a prolonged period of a month) does not hamper the app's ability to execute on a user's phone. For this experiment, we choose a random set of 50 apps (among the zombie apps from our large scale study). We created accounts for apps and interacted with them, if needed. (For such apps, we either inserted fake personalized information or played and achieved high scores and unlocked levels, prior to quarantine). We installed these apps on two phones (both Samsung Galaxy S5) and had the same user data on both of them. *ZapDroid* removed all the binaries of the zombie apps from one device (these were quarantined), while we let them remain on the

other device (there was no quarantine of any sort).

We ran this experiment for a month and then tried to restore the zombie apps on the second phone (where quarantine was performed). We found that 19 of these apps were updated (later versions). On the first phone, we simply allowed these updates. On the second phone, *ZapDroid* downloaded a new binary from the Google Play store and did a fresh install. Upon launching these apps, we found that for 4 of them, the data was changed on the first phone as compared to the original data. One was due to a database upgrade and the other 3 were just additions to a file (incremental state). The apps were able to execute with the user's old information. We observed that on the second phone, the same changes were seen, and the apps were able to execute seamlessly with the user's old data. This essentially demonstrated that the processes of quarantine and restoration had no impact on the operation of the app.

Evaluating the effectiveness of *ZapDroid* in constraining zombie apps from consuming resources: To showcase the efficacy of *ZapDroid* in ensuring that zombie apps do not consume significant resources, we compare its performance with that of two popular apps from the Google play store, viz., Advanced Task Killer [1] and Juice Defender [34]. These apps are specifically designed to kill undesired background processes. For this experiment, from the 15 random user profiles that we selected from our user study in Section 4.2, we choose 5 profiles where there is a large variation in the number of unused apps. We select 4 *identical* Nexus 4 devices and install the following: (a) *ZapDroid*, (b) an unmodified version of Android with Advanced Task Killer, (c) an unmodified version of Android with Juice Defender, and (d) a version of Android with none of the above installed. The last case is a baseline case and we compare the performance of (a), (b), and (c) with this case; specifically, we measure the savings in terms of energy (since this implictly

subsumes activities that correspond to CPU and network usage).

We install all the apps from the selected profiles sequentially on our 4 phone setup. We initialize all the potential zombie apps with the same state on all devices. We measure the energy saved by each system viz., (a), (b) or (c), for a 24 hour period (for each profile). Figure 4.13 depicts the energy savings in each case. The number on top of each cluster of bars, indicates the number of zombie apps in the particular user profile. We see that *ZapDroid* saves more than 2X energy compared to the other solutions for all user profiles. This is primarily because, with Advanced Task Killer and Juice Defender (using default settings) the killed background processes restart fairly often. Then, they resume their activities as before. In fact, we notice that the savings with both Advanced Task Killer and Juice Defender are slightly lower with user profile 4 with 32 zombie apps as compared to that with user profile 3 with just 14 zombie apps. This is simply an artifact of the latter's zombie apps being more active and resource intensive upon being awake. *ZapDroid* essentially prevents these zombie apps from waking up once quarantined and thus, a much higher energy savings is achieved.

Listing 4.4: Restricting a broadcast

```
 private final void deliverToRegisteredReceiverLocked(
     BroadcastRecord r, BroadcastFilter filter, boolean
     ordered) { ...
if(r.callingUid>=Process.FIRST_APPLICATION_UID &&
    canSendTo(filter.receiverList.uid)
            skip=true;
...
```

Finally, we point out here that neither Advanced Task Killer nor Juice Defender helps in preventing these apps from using undesired permissions when they wake up.

Security: We verify that *ZapDroid* in no case elevated any other apps' privileges for 50 apps. This is because it essentially imposes an *additional* constraint on the permissions granted to an app; it does not grant new permissions. This makes it more restrictive. In Code 4.4, we show a code snippet where this additional requirement (`canSendTo`) is imposed.

4.8 Discussion

Empowering applications with *ZapDroid* functions does not yield any benefits: One can conceivably empower certain applications with the features of Android. However, this not only still requires changes to the Android ecosphere, but also escalates the privileges of such applications which may lead to unforeseen vulnerabilities that we seek to avoid.

To illustrate, let us consider an implementation that modifies the `Launcher` app to allow it to quarantine and restore zombie apps. First, the user can simply request a clean unistall of the desired zombie app. The uninstall process will need to be modified so as to allow the `Launcher` app to back up the app data; in addition, the `Launcher` app will need to be provided with the privileges (root) to do such a back up. Second, one has to make changes to the `Launcher` app's `UninstallShortcutReceiver` class file to ensure that a quarantined app (while uninstalled) does not stop displaying its icon; this is essential for future restoration. In addition, here we will need to modify the intent to launch the app so as to invoke the install process instead of simply launching the app, when the icon is clicked. Finally, for the `Launcher` app to backup and restore user data, one would need to give the app the permission to connect to the Internet.

Problems: First, the above process does not distinguish between Category U and L zombie apps; all apps are treated as Category U apps. Thus, if the network connectivity is poor, a user may not be able to restore an app quickly; this may discourage him from quarantining some of the zombie apps. This reduces some of the benefits (e.g., energy savings) that *ZapDroid* can provide for Category L apps.

Second, the `Launcher` app, which is a user space app, now has privileges for accessing data that belongs to other apps, and potentially sending them over the Internet to unwanted entities. A modified version of Android that enables the placement of such a `Launcher` must therefore disallow the users from installing other third party `Launcher` apps from the market (e.g. [40]) to prevent such possibilities.

Coping with delays experienced due to quarantine of interactive apps: As alluded to earlier, users may have interactive or social apps like Yahoo! Mail or Skype which are seldom used. However, such apps consume heavy resources for the purposes of periodically checking for updates. While allowing them to run unperturbed continuously could hurt the smartphone resources (e.g., battery), quarantining them completely is clearly not the best idea either. In such cases, a user may be willing to trade-off slight delays in retrieving content, for battery savings.

To facilitate this, *ZapDroid* can be modified to reactivate such quarantined (possibly Category L) zombie apps periodically in order to enable them to check for updates. Thus restored, the app can retrieve updates, notify the user and remain active for a preset short period. It is then quarantined again. The frequency of such temporary restorations can be specified by the user; the lower this frequency, the higher the resource savings, but also the higher the latency in getting notifications. To implement this, an intent will have to be generated periodically to explicitly restore

and launch the app; a timeout can be set, upon the expiry of which the app is again quarantined. We will consider the implemention of this feature in future work.

Automating quarantine of zombie apps: In our current design of *ZapDroid*, we let the user select each candidate zombie app individually for quarantine. Our intent was to provide the user with fine-grained control over which unused apps can be put away. We could modify *ZapDroid* to enable the user to set thresholds on resource usage (or incidence of access to permissions) which can be used to determine if a candidate zombie app should be automatically quarantined. resources, and the invocations of permissions by zombie apps. Again, we defer such a possibility to future work.

4.9 Related Work

There are no works that share our vision of identifying zombie apps, quarantining them, and restoring them if the user desires. However, there are efforts related to some of the modules that we build within *ZapDroid*.

Profiling applications: Most prior work on profiling Android applications has been either geared towards helping developers build better apps [118], or detecting malicious apps [73]. ProfileDroid [127] undertakes offline profiling of a number of Android apps at several layers and identifies discrepancies between app specification and execution. While Ravindranath et al. [118] perform online profiling, they primarily characterize performance bottlenecks of apps. Yoon et al. [129] and Ding et al. [84] propose mechanisms to monitor app-specific battery consumption. However, these mechanisms are implemented in the user space, and consume high energy themselves. PowerTutor [132] does online analysis to estimate the energy consumption behavior of each

app on a smartphone. The Android Fuel Gauge also does something similar [12].

In our work, we leverage information with regards to CPU and network that is readily provided by the Android OS (with super user privileges). For monitoring battery consumption, we design and implement our own tool; it only functions at a coarse granularity (unlike PowerTutor) to measure the aggregate energy consumed by apps over long periods. It also accounts for energy due to UDP traffic transfers (unlike Fuel Gauge).

To the best of our knowledge, no prior work has looked into identifying infrequently-used apps and profiling their resource usage or their access to user information. Further, unlike prior work, we only profile apps that go unused for a pre-specified time.

Managing permissions of applications: Permission Denied [43] and XPrivacy [70] allow a superuser to change the permissions granted to apps; however, when a permission is changed, the phone must be rebooted. In addition, improper user-initiated revocations can end up crashing the app or the device (due to revocation of permissions to a system app) [69]. An undocumented feature in Android, App Ops, allowed user-level revocation of permissions from apps, but has been removed from the kernel in the latest versions [31]. TISSA [133] allows users to dynamically change permissions of apps; however negative consequences from such changes (e.g., the app crashing, or error pop ups [69]) are not well understood. To prevent apps from crashing when they are not given access to a permission, MockDroid [75] sends "mock" information to apps. Hornyack et al. [97] attempt using fake information as well; they hypothesize that if the screenshot of the app does not change upon a permission revocation, the app continues to execute normally. Again, the negative ramifications of isolated permission revocations are not studied. The work in Pearce et al. [115] develops a new framework to manage advertisements rather than providing apps with privileges to

receive advertisements. Blackphone which runs on PrivatOS [14] forks a version of the Android OS and modifies it to revoke permissions towards keeping information private; again, there could be negative effects of revoking isolated permissions (not studied). Unlike these efforts, once a zombie app is chosen for quarantine, we revoke all permissions from the zombie app as well as prevent other apps from communicating with it. This essentially *freezes* the state of the zombie app and does not adversely affect user experience.

TaintDroid [90] enables information flow tracking to detect permissions invoked by apps. It could potentially be used to monitor permission access patterns of candidate zombie apps. However, full-fledged flow tracking is an overkill to identify permission access, and this approach would impose unnecessary performance penalties. Instead, we use a much more coarse-grained approach to track permissions invoked by apps.

Killing processes: Applications like Advanced Task Killer [1], BatteryDoctor[13], and Juice Defender[34] are meant to allow users to kill background processes associated with either services or apps to save battery. Apart from the high resources consumed by these apps themselves (they are always running and monitoring for opportunities when tasks should be killed), killing background apps using these can have other detrimental effects [67]. The killed background processes restart due to either wakeup triggers or timeouts (some start right back up [11]) and resume consuming resources and/or accessing the user's private information. Since this process of killing and restarting could happen often with these applications, even more resources may be consumed; the restarted processes will each time rebuild their state prior to being killed [68]. The process of quarantine and restoration are done infrequently in a lightweight manner with *ZapDroid*, and thus, such resource consumption penalties are minimal.

Resource management: There are prior efforts that try to reduce resource consumption on smartphones. For example, Calder et al. [80] attempt to reduce network traffic, Paul and Kundu [114] attempt to reduce energy consumption, and Kemp et al. [101] shift the processing to the cloud to reduce CPU load. However, these efforts are orthogonal to our work. Note that unlike Cuervo et al. [82], we do not offload apps to the cloud for execution; rather, we simply store quarantined apps in the cloud when appropriate.

We note that Android kills background apps under low memory conditions, to reclaim memory; however, these apps can restart when the situation improves, by setting the START_STICKY bit [10]. *ZapDroid*'s goal is not just to kill apps when the memory runs low, but to curb activities of unused apps throughout for both resource savings and privacy.

4.10 Conclusions

Typical smartphone users install third-party applications on their smartphones, but end up not using them in the long run. Many such apps execute in the background consuming smartphone resources (e.g., network bandwidth, energy) and/or accessing private information. We conduct an IRB-approved user study on Amazon's Mechanical Turk to identify such apps, and showcase their negative impact. As our main contribution, we design and implement *ZapDroid*, which detects these unused apps, and identifies those that exhibit the above undesired behaviors. These so-called zombie apps are then quarantined in order to curb these behaviors. If the user so desires, *ZapDroid* can later restore the quarantined app quickly and efficiently, to the same state that it was in prior to the quarantine. Evaluations on our prototype of *ZapDroid* demonstrate that it is lightweight, and can more efficiently thwart zombie app activity as compared to state of the art solutions that target

killing undesired background processes.

Bibliography

[1] Advanced Task Killer. https://play.google.com/store/apps/details?id=com.rechild.advancedtaskkiller.

[2] Amazon EC2 micro-instance. http://amzn.to/14fxKbM.

[3] Amazon S3 pricing. http://amzn.to/1dRGFuz.

[4] An Introduction to Named Pipes— Linux Journal. http://www.linuxjournal.com/article/2156.

[5] Anatomy of Facebook. http://on.fb.me/1az2axi.

[6] Android: Accessibility Service. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService.html.

[7] Android Interface Definition Language (AIDL). http://developer.android.com/guide/components/aidl.html.

[8] Android Messenger. http://developer.android.com/guide/components/bound-services.html#Messenger.

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

[10] Android Service Lifecycle. http://developer.android.com/guide/components/services.html.

[11] Android Task Killers Explained: What They Do and Why You Shouldn't Use Them. http://lifehacker.com/5650894/android-task-killers-explained-what-they-do-and-why-you-shouldnt-use-them.

[12] Android's FuelGuage. https://android.googlesource.com/platform/packages/apps/Settings/+/android-4.3.1_r1/src/com/android/settings/fuelgauge/PowerUsageSummary.java.

[13] Battery Doctor. https://play.google.com/store/apps/details?id=com.ijinshan.kbatterydoctor_en.

[14] blackphone. https://www.blackphone.ch.

[15] Block Outgoing Network Access For a Single User Using Iptables. http://www.cyberciti.biz/tips/block-outgoing-network-access-for-a-single-user-from-my-server-using-iptables.html.

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

[17] The Diaspora project. http://diasporaproject.org/.

[18] Digital Diary: Are We Suffering From Mobile App Burnout? http://bits.blogs.nytimes.com/2013/02/15/digital-diary-are-we-suffering-from-mobile-app-burnout/?_r=0.

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

[20] Facebook fixes security glitch after leak of Mark Zuckerberg photos. http://lat.ms/14fx4mC.

[21] Facebook says it fixed leak that opened info to third-parties. http://wapo.st/12UidOW.

[22] Facebook: Statement of rights and responsibilities. https://www.facebook.com/legal/terms.

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

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

[25] Ganglia. http://ganglia.sourceforge.net/.

[26] Google App Engine. http://bit.ly/117kPXo.

[27] Google App Engine Pricing. http://bit.ly/1cclPzm.

[28] Google Cloud Messaging for Android. http://developer.android.com/google/gcm/index.html.

[29] Google Play: number of downloads 2010-2013. http://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/.

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

[31] Google Removes App Ops Privacy Control Feature from Android 4.4.2. http://www.dailytech.com/Google+Removes+App+Ops+Privacy+Control+Feature+from+Android+442/article33936.htm.

[32] Grouptweet. http://www.grouptweet.com/.

[33] Heroku. http://www.heroku.com/.

[34] Juice Defender. https://play.google.com/store/apps/details?id=com.latedroid.juicedefender.

[35] Launch Checklist for Android. http://developer.android.com/distribute/tools/launch-checklist.html.

[36] Monsoon Power Meter. https://www.msoon.com/LabEquipment/PowerMonitor/.

[37] Network classifier cgroup. https://android.googlesource.com/kernel/common/+/android-3.10/Documentation/cgroups/net_cls.txtt.

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

[39] Nielsen: US smartphones have an average of 41 apps installed, up from 32 last year. http://thenextweb.com/insider/2012/05/16/nielsen-us-smartphones-have-an-average-of-41-apps-installed-up-from-32-last-year/.

[40] Nova Launcher Prime. https://play.google.com/store/apps/details?id=com.teslacoilsw.launcher.prime.

[41] Number of available Android applications. http://www.appbrain.com/stats/number-of-android-apps.

[42] Oauth 2.0. http://oauth.net/2/.

[43] Permission Denied. https://play.google.com/store/apps/details?id=com.stericson.permissions.donate.

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

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

[46] Powertutor. http://bit.ly/hVaXh1.

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

[48] Recommended elliptic curves for federal government use. http://1.usa.gov/14fwPYI.

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

[50] RFC6278: Use of static-static elliptic curve Diffie-Hellman key agreement in cryptographic message syntax. http://tools.ietf.org/html/rfc6278.

[51] Samsung Galaxy S4. http://www.samsung.com/global/microsite/galaxys4/.

[52] Secure hash standard. 1.usa.gov/cISXx3.

[53] SharedPreferences. http://developer.android.com/reference/android/content/SharedPreferences.html.

[54] Smartphone users waste '£6m a year on one hit wonder apps'. http://metro.co.uk/2013/05/30/smartphone-users-waste-6m-a-year-on-one-hit-wonder-apps-3814805/.

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

[56] Some quitting Facebook as privacy concerns escalate. http://bit.ly/15pqQmK.

[57] strace Linux- man page. http://linux.die.net/man/1/strace.

[58] Syme. https://getsyme.com.

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

[60] Tweetcaster. http://tweetcaster.com/.

[61] Tweetworks. http://www.tweetworks.com.

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

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

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

[65] What actually happens when you swipe an app out of the recent apps list? http://android.stackexchange.com/questions/19987/what-actually-happens-when-you-swipe-an-app-out-of-the-recent-apps-list.

[66] What do the permissions that applications require mean? http://android.stackexchange.com/questions/38388/what-do-the-permissions-that-applications-require-mean.

[67] Why RAM Boosters And Task Killers Are Bad For Your Android. http://www.makeuseof.com/tag/ram-boosters-task-killers-bad-android/.

[68] Why You Shouldn't Be Using a Task Killer with Android. http://forum.xda-developers.com/showthread.php?t=678205.

[69] XDA Community Support for XPrivacy. http://forum.xda-developers.com/xposed/modules/xprivacy-ultimate-android-privacy-app-t2320783/page39.

[70] XPrivacy. http://www.xprivacy.eu.

[71] Your Facebook friends have more friends than you. http://wapo.st/11I58Mj.

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

[73] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, MALWARE '11, 2011.

[74] Filipe Beato, Markulf Kohlweiss, and Karel Wouters. Scramble! your social network data. In Simone Fischer-Hbner and Nicholas Hopper, editors, *PETS*, volume 6794 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2011.

[75] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, 2011.

[76] Michael S. Bernstein, Eytan Bakshy, Moira Burke, and Brian Karrer. Quantifying the invisible audience in social networks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 21–30, New York, NY, USA, 2013. ACM.

[77] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 321–334, Washington, DC, USA, 2007. IEEE Computer Society.

[78] Dan Boneh and Michael Hamburg. Generalized identity based and broadcast encryption schemes. In *ASIACRYPT*, 2008.

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

[80] M. Calder and M.K. Marina. Batch scheduling of recurrent applications for energy savings on mobile phones. In *Sensor Mesh and Ad Hoc Communications and Networks (SECON), 2010 7th Annual IEEE Communications Society Conference on*.

[81] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, 2010.

[82] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, 2010.

[83] George Danezis and Andrei Serjantov. Statistical disclosure or intersection attacks on anonymity systems. In *Proceedings of the 6th International Conference on Information Hiding*, IH'04, pages 293–308, Berlin, Heidelberg, 2004. Springer-Verlag.

[84] Fangwei Ding, Feng Xia, Wei Zhang, Xuhai Zhao, and Chengchuan Ma. Monitoring energy consumption of smartphones. In *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, 2011.

[85] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

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

[87] Robert Dunbar. The ultimate brain teaser. `http://bit.ly/17Floky`.

[88] Gene Tsudik Emiliano De Cristofaro, Claudio Soriente and Andrew Williams. Hummingbird: Privacy at the time of twitter. Cryptology ePrint Archive, Report 2011/640, 2011. `bit.ly/SYBEzK`.

[89] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[90] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[91] Ariel J. Feldman, Aaron Blankstein, Michael J. Freedman, and Edward W. Felten. Social networking with frientegrity: Privacy and integrity with an untrusted provider. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 31–31, Berkeley, CA, USA, 2012. USENIX Association.

[92] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[93] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *ACM Conference on Computer and Communications Security*, 2011.

[94] David Alan Grier and Mary Campbell. A social history of bitnet and listserv, 1985-1991. *IEEE Annals of the History of Computing*, 2000.

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

[96] D.R. Hankerson, S.A. Vanstone, and A.J. Menezes. *Guide to Elliptic Curve Cryptography*. Springer Professional Computing. 2004.

[97] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[98] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, 2012.

[99] Jung Yeon Hwang, Dong Hoon Lee, and Jongin Lim. Generic transformation for scalable broadcast encryption scheme. In *CRYPTO*, 2005.

[100] Sonia Jahid, Shirin Nilizadeh, Prateek Mittal, Nikita Borisov, and Apu Kapadia. DECENT: A decentralized architecture for enforcing privacy in online social networks. In *IEEE SESOC*, 2012.

[101] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Energy efficient information monitoring applications on smartphones through communication offloading. In *Mobile Computing, Applications, and Services*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2012.

[102] Neal Koblitz, Alfred Menezes, and Scott Vanstone. The state of elliptic curve cryptography. *Designs, Codes and Cryptography*, 2000.

[103] Balachander Krishnamurthy and Craig Willis. Characterizing privacy in online social networks. In *WOSN*, 2008.

[104] Balachander Krishnamurthy and Craig Willis. On the leakage of personally identifiable information via online social networks. In *WOSN*, 2009.

[105] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 364–373, Oct 1997.

[106] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

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

[108] Dongtao Liu, Amre Shakimov, Ramon Caceres, Alexander Varshavsky, and Landon P. Cox. Confidant: Protecting OSN Data without Locking it Up. In *Middleware*, 2011.

[109] David Lubicz and Thomas Sirvent. Attribute-based broadcast encryption scheme made efficient. In *AFRICACRYPT*, 2008.

[110] Matthew M. Lucas and Nikita Borisov. FlyByNight: Mitigating the privacy risks of social networking. In *WPES*, 2008.

[111] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 448–457, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

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

[113] Shirin Nilizadeh, Sonia Jahid, Prateek Mittal, Nikita Borisov, and Apu Kapadia. Cachet: A decentralized architecture for privacy preserving social networking with caching. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 337–348, New York, NY, USA, 2012. ACM.

[114] K. Paul and T.K. Kundu. Android on mobile devices: An energy perspective. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, 2010.

[115] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, 2012.

[116] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of HotNets–I*, Princeton, New Jersey, October 2002.

[117] Raluca Ada Popa, Hari Balakrishnan, and Andrew J. Blumberg. VPriv: Protecting privacy in location-based vehicular services. In *USENIX Security Symposium*, 2009.

[118] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[119] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for Web transactions. *ACM TISSEC*, 1998.

[120] Amre Shakimov, Harold Lim, Ramon Cceres, Landon P. Cox, Kevin Li, Dongtao Liu, and Alexander Varshavsky. Vis-ã-Vis: Privacy-preserving online social networks via virtual individual servers. In *COMSNETS*, 2011.

[121] Indrajeet Singh, Michael Butkiewicz, Harsha V. Madhyastha, Srikanth V. Krishnamurthy, and Sateesh Addepalli. Building a wrapper for fine-grained private group messaging on Twitter. In *HotPETS*, 2012.

[122] J. H. Song, R. Poovendran, J. Lee, and T. Iwata. RFC 4493 - The AES-CMAC Algorithm, 2006.

[123] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious ram. In *NDSS*. The Internet Society, 2012.

[124] Latanya Sweeney. K-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, October 2002.

[125] Amin Tootoonchian, Stefan Saroiu, Yashar Ganjali, and Alec Wolman. Lockr: Better privacy for social networks. In *CoNEXT*, 2009.

[126] Bryan Urban, Verena Tiefenbeck, and Kurt Roth. Energy consumption of consumer electronics in US homes in 2010. http://bit.ly/10NMqOn.

[127] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Profiledroid: Multi-layer profiling of android applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, 2012.

[128] Christo Wilson, Troy Steinbauer, Gang Wang, Alessandra Sala, Haitao Zheng, and Ben Y. Zhao. Privacy, availability and economics in the Polaris mobile social network. In *HotMobile*, 2011.

[129] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Presented as part of the 2012 USENIX Annual Technical Conference*, 2012.

[130] Nickolai Zeldovich, Silas Boyd Wickizer, and David Mazires. Securing distributed systems with information flow control. In *NSDI*, 2008.

[131] Liang Zhang and Alan Mislove. Building confederated web-based services with priv.io. In *Proceedings of the First ACM Conference on Online Social Networks*, COSN '13, pages 189–200, New York, NY, USA, 2013. ACM.

[132] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, 2010.

[133] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and VincentW. Freeh. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011.