Bases for Trust in Online Social Networks

by

Amre Shakimov

Department of Computer Science
Duke University

Date: ______________________

Approved:

____________________________________
Landon P. Cox, Supervisor

____________________________________
Jeffrey S. Chase

____________________________________
Bruce M. Maggs

____________________________________
Ramón Cáceres

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2012
ABSTRACT

Bases for Trust in Online Social Networks

by

Amre Shakimov

Department of Computer Science
Duke University

Date: __________________________
Approved:

_____________________________
Landon P. Cox, Supervisor

_____________________________
Jeffrey S. Chase

_____________________________
Bruce M. Maggs

_____________________________
Ramón Cáceres

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University 2012
Copyright © 2012 by Amre Shakimov
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence
Abstract

Online Social Network (OSN) services such as Facebook and Google+ are fun and useful. Hundreds of millions of users rely on these services and third-party applications to process and share personal data such as friends lists, photos, and geographic location histories. The primary drawback of today’s popular OSNs is that users must fully trust a centralized service provider to properly handle their data.

This dissertation explores the feasibility of building feature-rich, privacy-preserving OSNs by shifting the bases for trust away from centralized service providers and third-party application developers and toward infrastructure providers and OSN users themselves.

We propose limiting the trust users place in service providers through two decentralized OSNs: Vis-a-Vis and Confidant. In Vis-a-Vis, privacy-sensitive data is only accessed by user-controlled code executing on “infrastructure as a service” platforms such as EC2. In Confidant this data may only be accessed by code running on desktop PCs controlled by a user’s close friends. To reduce the risks posed by third-party OSN applications, we also developed a Multi-User Taint Tracker (MUTT). MUTT is a secure “platform as a service” that ensures that third-party applications adhere to access policies defined by service providers and users.

Vis-à-Vis is a decentralized framework for location-based OSN services based on the privacy-preserving notion of a Virtual Individual Server (VIS). A VIS is a personal virtual machine running within a paid compute utility. In Vis-à-Vis, a person stores her data on
her own VIS, which arbitrates access to that data by others. VISs self-organize into overlay networks corresponding to social groups with whom their owners wish to share location information. Vis-à-Vis uses distributed location trees to provide efficient and scalable operations for creating, joining, leaving, searching, and publishing location data to these groups.

Confidant is a decentralized OSN platform designed to support a scalable application framework for OSN data without compromising users’ privacy. Confidant replicates a user’s data on servers controlled by her friends. Because data is stored on trusted servers, Confidant allows application code to run directly on these storage servers. To manage access-control policies under weakly-consistent replication, Confidant eliminates write conflicts through a lightweight cloud-based state manager and through a simple mechanism for updating the bindings between access policies and replicated data.

For securing risks from third-party OSN applications, this thesis proposes a Multi-User Taint Tracker (MUTT) – a secure “platform as a service” designed to ensure that third-party applications adhere to access policies defined by service providers and users. MUTT’s design is informed by a careful analysis of 170 Facebook apps, which allows us to characterize the requirements and risks posed by several classes of apps. Our MUTT prototype has been integrated into the AppScale cloud system, and experiments show that the additional data-confidentiality guarantees of running an app on MUTT come at a reasonable performance cost.
To my family.
Contents

Abstract iv
List of Tables xi
List of Figures xii
Acknowledgements xiii

1 Introduction 1
1.1 Online social networks 1
1.2 Trust and privacy in online social networks 3
1.3 This thesis 4
1.4 Road map for the dissertation 4

2 Leveraging cloud for building a privacy-preserving OSN platform 6
2.1 Introduction 6
2.2 Location-based groups 9
2.3 Architecture 11
  2.3.1 Trust-and-threat model 13
  2.3.2 Membership service 15
  2.3.3 Location trees 15
  2.3.4 Operations 19
2.4 Implementation 20
  2.4.1 Vis-a-Vis and ZooKeeper 20
2.4.2 Group Finder mobile application .................................. 21
2.5 Evaluation ................................................................. 23
  2.5.1 Micro-benchmarks ............................................... 24
  2.5.2 Effect of geographic distribution of VISs ...................... 27
  2.5.3 End-to-end latency ............................................... 29
2.6 Conclusion ............................................................... 31

3 Building a privacy-preserving OSN platform using commodity computers 32
  3.1 Introduction ............................................................. 32
  3.2 Overview ............................................................... 35
    3.2.1 Architecture .................................................. 36
    3.2.2 Trust and threat model ...................................... 37
  3.3 Design ................................................................. 39
    3.3.1 Cryptographic state ......................................... 39
    3.3.2 Objects and access policies ................................. 41
    3.3.3 Name servers .................................................. 45
    3.3.4 Storage servers ................................................ 46
  3.4 Implementation ........................................................ 51
    3.4.1 Client ............................................................ 51
    3.4.2 Name Server .................................................... 52
    3.4.3 Storage Server .................................................. 53
    3.4.4 Data-Processing Scripts .................................... 53
    3.4.5 Untrusted Storage Server .................................... 54
  3.5 Evaluation .............................................................. 54
    3.5.1 Data-Processing Performance ............................... 55
    3.5.2 Access performance .......................................... 57
4 Enforcing third-party OSN applications to respect service provider’s policies

4.1 Introduction

4.2 Background

4.3 Trust and threat model

4.4 Status Quo

4.4.1 Classification of applications

4.4.2 Service policies

4.5 Design Challenges and Goals

4.5.1 Design Challenges and Limitations

4.5.2 Design Goals

4.6 Design

4.6.1 Strawman

4.6.2 MUTT Architecture Overview

4.6.3 OSN drivers

4.6.4 Language runtime

4.6.5 Platform

4.6.6 MUTT Policy Enforcement

4.7 Implementation

4.7.1 Python

4.7.2 AppScale

4.8 Evaluation

4.8.1 Storage overhead

4.8.2 Runtime overhead
List of Tables

2.1 Group Finder mean latency over WiFi in milliseconds, with standard deviations in parentheses. .......................... 27
2.2 Group Finder mean latency over 3G cellular in milliseconds, with standard deviations in parentheses. .......................... 28
3.1 Storage-server messages ................................................. 46
4.1 Facebook applications .................................................. 72
4.2 Categories of Facebook applications ................................. 72
4.3 Microbenchmarks results ................................................. 92
4.4 Latencies of selected tests from pybench suite ...................... 92
# List of Figures

1.1 Schematic representation of OSN Platform ........................................ 2

2.1 Vis-à-Vis architectural overview ..................................................... 14

2.2 User U1’s view of a group’s location tree ........................................ 16

2.3 Screenshot of the Group Finder application ....................................... 22

2.4 join and update performance .......................................................... 24

2.5 Local search performance .............................................................. 26

2.6 Group-wide search performance ....................................................... 27

2.7 Effect of geographic distribution of VISs .......................................... 29

3.1 Confidant architecture ................................................................. 36

3.2 Script performance ........................................................................ 56

3.3 Status-update performance .............................................................. 57

3.4 Photo performance ........................................................................ 59

3.5 Simulated read-success rates (Microsoft) .......................................... 60

3.6 Simulated read-success rates (Gnutella) ........................................... 61

3.7 Simulated write-success rates (Microsoft) ......................................... 62

3.8 Simulated write-success rates (Gnutella) .......................................... 63

4.1 Architecture overview ................................................................. 80
Acknowledgements

It would not have been possible to finish this dissertation without the guidance of my advisor, committee members, help from friends, colleagues, and support from my family. Here is my clumsy and certainly incomplete attempt to thank them for being with me throughout this journey.

Above all, I would like to express my deepest gratitude to my advisor, Landon Cox, for helping me to grow as an individual and professional. If some day I would be in a position to supervise someone, I think I know how my mentoring style would look like.

I cannot thank enough Ramón Cáceres, who I was truly fortunate to meet and work with for most of the years I have been at Duke. I also thank Jeff Chase and Bruce Maggs for being the greatest and most fun professors I will always admire.

Many thanks to Eduardo and Peter. They were always nearby willing to listen, help and support. I have heard from Amrip that he would be always at their service.

I would like to thank Harish, Harold, Hero, Jeff, Nedyalko, Michael, Pablo, Rishi, Ryan, Vamsi and so many more people I was lucky to meet at the department and beyond. Without them life as a grad student would never be so much fun.

I had the greatest roommates ever. Thanks to Stefan, Nikifor, Seth for being my friends and just awesome people.

Finally, I want to thank my family. They were always supporting, encouraging and motivating me. I would never be in this position without them.
1

Introduction

1.1 Online social networks

Free online social networks (OSNs) such as Facebook, Twitter, and Google+ are central to the lives of millions of users. They help people share information with friends, stay in touch with families, and connect with new people based on common interests and geographical locations. Online social networks have already changed the landscape of the web \([119, 98, 99, 100]\), traditional media \([105, 65, 39]\), and Internet search \([19]\).

To illustrate the growth of social networking sites consider the following examples. Facebook alone has over 800 million active users with over 50% of them using the site every day \([50]\). Twitter has more than 100 million active users, and Google+ acquired 90 million users within 6 months of its inception \([8]\).

The volume of data handled by OSNs is staggering. Facebook receives more than 30 billion shared items every month \([50]\) and is the largest photo-sharing Internet service, outgrowing even dedicated photo services like Flickr and Photobucket \([2]\). Twitter receives about a billion tweets weekly \([13]\) handling up to almost 9,000 tweets per second \([15]\). Foursquare handled its 40-millionth check-in only five weeks after its 22-millionth \([76]\).
OSNs have also evolved into platforms for third-party websites and extensions (applications). Many websites use OSN APIs to augment their functionality. Similarly, third-party applications can extend an OSN by adding useful and interesting new features. More than 7 million applications and websites already integrate Facebook functionality via Facebook Platform API, and developers from over 190 countries have built applications for the Facebook Platform. Users install 20 million Facebook applications daily.

Figure 1.1 depicts a schematic overview of a typical OSN such as Facebook or Google+ and shows three main entities that OSN users interact with: (a) infrastructure provider/owner, (b) service provider, and (c) third-party applications.

Data that OSN users generate are handled by code running on hardware maintained by infrastructure providers. Service providers are responsible for managing the code that provides functionality to OSN users and mediates access to users’ data. Third-party applications interact with a service through remote APIs and often have access to a subset of users’ data. Typically, apps run on separate infrastructure such as on their own servers or third-party cloud solutions.

\[1\quad \text{In many cases infrastructure and service providers are running under the same administrative domain, i.e., service providers have their own datacenters.}\]
1.2 Trust and privacy in online social networks

In a survey of 2,253 adult OSN users, 65% had changed their privacy settings to limit what information they share with others, 36% had deleted comments from their profile, and 33% expressed concern over the amount of information about them online [75]. In a survey of young adults, 55% of 1,000 respondents reported being more concerned about privacy issues on the Internet than they were five years ago [60].

OSNs try to address these concerns by introducing tools to simplify management of online privacy [17, 22]. However, we argue that today’s OSN architectures pose inherent risks to users’ privacy.

Many recent incidents suggest that trusting large centralized services to safeguard sensitive OSN data may be unwise [23, 24, 25, 28, 29, 34, 35, 11]. First, concentrating the personal data of hundreds of millions of users under a single administrative domain leaves users vulnerable to large-scale privacy violations via security attacks [24] and inadvertent disclosures [23]. Furthermore, providers offering free services must generate revenue. OSN terms of service often reflect these incentives by giving the provider the right to reuse users’ data in any way the provider sees fit [47]. These rights include sharing data with third-party advertisers and websites without explicit consent from users [107].

OSNs also increasingly function as programming platforms through service extensions called applications. The server-side code for these applications often executes on hosting infrastructure such as Google’s App Engine that is outside the control of the service. As a result, a service has no way to guarantee that data shared with a third-party application will be protected according to users’ access policies and to the service’s terms of use. In order to run, apps have to get a user’s permission to access her data through the OSN-defined APIs. After permission has been granted, the app developers are in charge of properly handling this data. OSN platforms try to formalize their relationships with the third-party developers via Platform policies and guidances. Unfortunately, many apps have already
been caught leaking private data [14, 6, 12, 77, 54] and it is hard to even estimate the real number of such incidents and misbehaving applications.

Thus, existing OSNs provide inadequate bases for trusting service providers and application developers. If OSN users do not feel comfortable with this status quo their only option is to opt out.

In this dissertation, I seek to demonstrate that alternative OSN architectures can provide stronger bases for trust without compromising OSN features.

1.3 This thesis

Users’ privacy and trust in online social networks is the focus of my dissertation. My thesis statement is as follows:

It is feasible to design a privacy-preserving OSN that provides the same features as existing centralized OSNs and with stronger bases of trust.

1.4 Road map for the dissertation

This dissertation validates the thesis via the design, implementation, and experimental evaluation of three systems: Vis-à-Vis, Confidant, and MUTT.

Chapter 2 describes Vis-à-Vis, a decentralized platform for building location-based privacy-preserving OSN services. Vis-à-Vis is based on a privacy-preserving notion of a Virtual Individual Server. By shifting trust from service providers to infrastructure providers such as Amazon EC2 or Heroku, we provide different privacy and security guarantees to the OSN users. In contrast to free OSN services, paid compute utility providers allow users to retain full rights to the content and applications that users place on these services. Thus, trust in Vis-à-Vis is based on the cloud providers’ business model, since they charge for using their infrastructure and therefore do not have direct incentives to violate users’ privacy, for example, by selling it to the marketing companies.
In Chapter 3 we discuss Confidant, a decentralized OSN platform designed to support a scalable application framework for OSN data without compromising users’ privacy. In contrast to Vis-à-Vis, that uses highly available cloud services, Confidant uses commodity computers to host OSN data coupled with a novel socially-informed replication scheme. This scheme takes advantage of the trust embedded in the social network by replicating a user’s data on servers controlled by her friends. Because data is stored on trusted servers, Confidant allows application code to run directly on these storage servers and thus supports scalable data-processing framework.

We describe Multi-user Taint Tracker (MUTT) in Chapter 4. MUTT is a policy-enforcing platform as a service for third-party OSN extensions. It uses a taint-tracking mechanism to propagate taint labels within the untrusted application’s runtime to prevent information leaks and ensure proper management of users’ data. MUTT supports most existing OSN applications without requiring changes to service providers’ APIs and third-party applications’ design.

Related work is discussed in Chapter 5. Chapter 6 concludes the dissertation with a summary of the key contributions, future research directions generated by this dissertation, and closing remarks.
Leveraging cloud for building a privacy-preserving OSN platform

In this chapter we describe Vis-à-Vis [96] – a system that uses highly-available cloud infrastructure for building privacy-preserving location-based OSN services.

2.1 Introduction

Free online social networks (OSNs) such as Facebook, Twitter, and Foursquare are central to the lives of millions of users and still growing. Facebook alone has over 500 million active users and 250 million unique visitors each day [50]. The volume of data handled by OSNs is staggering: Facebook receives more than 30 billion shared items every month [50], Twitter receives more than 55 million tweets each day [62], and Foursquare handled its 40-millionth check-in only five weeks after handling its 22-millionth [76].

At the same time, many recent incidents suggest that trusting free centralized services to safeguard sensitive OSN data may be unwise [23, 24, 25, 34, 35]. These examples underscore the inherent risks of the prevailing centralized OSN model. First, concentrating the personal data of hundreds of millions of users under a single administrative domain
leaves users vulnerable to large-scale privacy violations via inadvertent disclosures [23] and malicious attacks [24]. Second, providers offering free services must generate revenue by other means. OSN terms of service often reflect these incentives by giving the provider the right to reuse users’ data in any way the provider sees fit [48]. These rights include sharing data with third-party advertisers and websites without explicit consent from users [107].

Unsurprisingly, many people have grown wary of the OSN providers they depend on to protect their private information. In a survey of 2,253 adult OSN users, 65% had changed their privacy settings to limit what information they share with others, 36% had deleted comments from their profile, and 33% expressed concern over the amount of information about them online [75]. In a survey of young adults, 55% of 1,000 respondents reported being more concerned about privacy issues on the Internet than they were five years ago [60].

Given the importance of OSNs in users’ lives and the sensitivity of the data users place in them, it is critical to limit the privacy risks posed by today’s OSNs while preserving their features. To address this challenge, we have developed a general framework for managing privacy-sensitive OSN data called Vis-à-Vis. Vis-à-Vis can interoperate with existing OSNs and is organized as a federation of independent, personal Virtual Individual Servers (VISs). A VIS is a virtual machine running in a paid cloud-computing utility such as Amazon Elastic Compute Cloud (EC2) or Rackspace Cloud Servers. Utilities provide better availability than desktop PCs and do not claim any rights to the content placed on their infrastructure [18]. Thus, just as cloud utilities are already trusted with many enterprises’ intellectual property, utility-based VISs store their owner’s sensitive data and arbitrate requests for that data by other parties.

In this chapter, we focus on the rapidly growing challenge of preserving the privacy of location information within an OSN. Location-based OSNs utilize privacy-sensitive information about users’ physical locations and are increasingly popular [73, 69, 55, 101]. For example, as of June, 2010, Foursquare had more than 1.5 million users and was expected
to grow to 2 million users by July [106].

Vis-à-Vis supports location-based OSNs through a group abstraction that gives members control over how they share their location and allows them to query the locations of other members. Groups are administered by the users that created them using a range of admissions policies. For example, groups can be open, as are most of Facebook’s “fan pages”, restricted by social relationships such as “Alice’s friends,” or restricted by a set of credentials such as “The Duke Alumni Club of New York.” Depending on a group’s admission policy, members may wish to share their location at finer or coarser granularities. Prior studies have shown that users will typically disclose their full location or no location at all with close friends [38], but will utilize vaguer location descriptions when sharing their location information on public sites [20].

In addition, we aim for Vis-à-Vis groups to scale to thousands of members. While we expect most groups with which people share private location information will be limited to hundreds of members (e.g., the average Facebook user has 130 friends [50]), we want the flexibility to scale to much larger groups if the need arises (e.g., 23% of Facebook’s fan pages have more than 1,000 members [94]).

To provide users with flexible control of their location information and to scale to groups with thousands of members, Vis-à-Vis organizes VISs into per-group overlay networks we call location trees. Within each tree, higher nodes represent coarser geographic regions such as countries while lower nodes represent finer regions such as city blocks. Interior nodes are chosen from the set of member VISs via a distributed consensus protocol. A user may publish her location to a group at an arbitrary granularity as long as her VIS becomes a leaf node within the subtree covering this location. Queries over a region are sent to the interior nodes covering that region and passed down the tree to lower nodes. Using this hierarchy, Vis-à-Vis guarantees that location queries complete in $O(\log(n))$ hops for groups of size $n$.

In summary, this chapter presents the following contributions:
• It presents the design of a privacy-preserving framework for location-based OSNs based on hierarchical overlay networks of Virtual Individual Servers (Sections 2.2 and 2.3).

• It describes an implementation of this framework, including a companion social application for mobile phones, that provides efficient and scalable OSN operations on distributed location data (Section 2.4).

• It demonstrates the feasibility of our approach through performance experiments involving up to 500 virtual machines running in Amazon EC2, including machines distributed across two continents. We found that the latency of our decentralized system is competitive with that of a centralized implementation of the same OSN operations (Section 2.5).

Despite this chapter’s focus on the sharing of geographic locations within large social groups, it should be clear that the Vis-à-Vis framework can be generalized to support other data types and social relationships, for example photographs shared among pairs of friends. In addition, Virtual Individual Servers can support many other applications besides online social networking, for example a personal synchronization service for mobile devices and a personal email service [32].

2.2 Location-based groups

The central abstraction supported by Vis-à-Vis is a group, as befits the central role played by social groups in OSNs. As a foundation, each principal in Vis-à-Vis is defined by a public-private key pair. Users are defined by a self-signed key pair. The private half of each user’s key pair is stored securely by her VIS, allowing a VIS to act on her behalf. Users distribute their public key and the IP address of their VIS out of band (e.g., via email or an existing OSN such as Facebook).
Each group consists of an owner, a set of users defining the group’s membership, and a mapping from group members to geographic regions. The group owner is the user who initiates and maintains the group. Each user within the group possesses a shared attribute such as an inter-personal relationship with the group owner or an interest in a particular topic. The geographic region associated with each group member is a geographic area the user wishes to share with other group members. Shared regions can be fine-grained or coarse-grained and can be set statically (e.g., hometown) or updated dynamically (e.g., current GPS coordinates).

Groups are named by a descriptor consisting of the group owner’s public key and a string used to convey the attribute shared among group members. Descriptors can be expressed as a \( \langle K^+_{owner}, \text{string} \rangle \) pair, where \( K^+_{owner} \) is the public key of the group owner. For example, a user, Alice, who is the president of the Duke Alumni Club of New York and works for AT&T might create groups \( \langle K^+_{Alice}, \text{DukeClubNYC} \rangle \), and \( \langle K^+_{Alice}, \text{AT&Tcoworkers} \rangle \). Including the group owner’s public key in each group descriptor allows users to manage the descriptor namespace independently and prevents naming conflicts. Descriptors do not contain privacy-sensitive information and, like a user’s public key and VIS IP address, are distributed out of band. We imagine that descriptors will be embedded in web sites and users’ existing OSN profiles.

Vis-à-Vis supports five operations on location-based groups under the following semantics:

- **create(string, policy)**: Creates a new, empty group using the public key of the caller and the passed-in string to construct the group descriptor. The policy argument allows group owners to define a call-back for implementing admission-policy plugins.

- **join(descriptor, region, credential)**: Adds the caller to the group with the specified descriptor, and, if successful, sets the region she shares with other group members. Success of the call depends on whether the credential passed in by the caller satisfies
the admission policy for the group. Credentials can be empty for open groups, an email address from a specific domain such as duke.edu, or a signed social attestation [109] specifying a relationship between users.

- **remove**(descriptor) : Removes the caller from the group.

- **update**(descriptor, region) : Creates a mapping from the caller to a geographic region within a group. The caller must already be a group member for this operation to succeed.

- **search**(descriptor, region) : Returns a list of all group members (and their associated geographic regions) whose regions are contained within the passed-in region.

The update and search operations are general enough to support a wide range of options. Depending on how much detail users want to reveal about their location to other group members, they can post different regions to different groups at arbitrary granularities. For example, if a user is usually not interested in being located by alumni of her alma mater, she could limit the region she shares with them to the city where she lives. However, in the fall, when she gathers with other fans to watch college football games, she may want to share the location of the bar where they usually meet. Such decisions can be made independently of how her location is represented in any other groups to which she might belong.

### 2.3 Architecture

To realize the Vis-à-Vis group abstraction, we organize users’ VISs into the hierarchy shown in Figure 2.3.1. The Vis-à-Vis architecture is similar in many respects to the distributed tree structures utilized by Census [40] and P2PR-Tree [78]. We choose to use a hierarchical structure rather than a distributed hash table (DHT) because DHTs’ simple get-set interface does not easily support the range queries required for the search operation.
We initially considered using a distributed skip graph [58] to manage location information, but were unhappy with the overhead of resorting distributed lists whenever a user changed her location.

Users access location-based groups through clients such as stand-alone mobile applications and web browsers. Vis-à-Vis is designed to interoperate with, rather than replace, established OSNs such as Facebook. A combination of embedded descriptors, web-browser extensions, and OSN APIs such as Facebook Connect allow users to adopt Vis-à-Vis while retaining their significant investments in existing platforms. Existing OSNs are treated as untrusted services that store only opaque pointers to sensitive content stored in VISs.

For example, users can integrate a Vis-à-Vis-managed location-based group into a Facebook group by embedding a group descriptor in the Facebook group’s information. When a user loads the group page in their web browser, a Vis-à-Vis browser extension interprets the document received from Facebook, identifies the group descriptor, and rewrites the page to include information downloaded from the appropriate VISs. Rendered pages can include a map displaying members’ current locations, or UI components for updating a user’s published location. These techniques for integrating externally managed information with existing OSNs are not new [109, 26, 56, 70]. We present an example of a standalone mobile client in Section 2.4.

Clients interact with groups via their user’s VIS. Each group supports a membership service responsible for implementing admission policies and for maintaining pointers to the group’s location tree. The root and interior nodes of the location tree are coordinator VISs. In addition, each group member’s VIS acts as a leaf node in the location tree.

The rest of this section describes Vis-à-Vis’s trust-and-threat model, the function of each architectural component, and how these components collectively implement the group operations described in Section 2.2.
2.3.1 Trust-and-threat model

Vis-à-Vis’s decentralized structure provides users with stronger privacy protections than centralized services such as Facebook and MySpace, because it gives them more direct control over who has access to their personal data. Of course, Vis-à-Vis cannot eliminate breaches completely, and this section describes the classes of attacks that Vis-à-Vis is and is not designed to address.

Vis-à-Vis’s trust model is based on the business interests of compute utilities and the inter-personal relationships of users; both the compute utility and a user’s social relations are trusted to not leak any sensitive information to which they have access. Unlike decentralized OSNs such as Persona [26], in which users do not trust their storage services, Vis-à-Vis exposes unencrypted data to the utilities hosting VISs.

There are three key benefits of entrusting compute utilities with access to cleartext data: 1) it simplifies key management, 2) it reduces the risk of large-scale data loss due to lost or corrupted state, and 3) most important for this chapter, it allows computations such as range queries to be executed on remote storage servers, which is a key requirement for mobile services such as Loopt [73] and Google Latitude [69].

Since compute utilities control the hardware on which VISs execute, utility administrators can access all of a user’s personal data. Despite this, Vis-à-Vis’s assumption that compute utilities will not exercise this power is reasonable. Compute utilities’ business model is based on selling computational resources to users rather than advertising to third parties. The legal language of utilities’ user agreements formalize these limitations [18], providing economic and legal consequences if personal data is abused. In contrast, OSN terms of use usually grant the service provider a “non-exclusive, transferable, sub-licensable, royalty-free, worldwide license to use any IP content that you post” [47]. We note that many companies already trust compute utilities with their intellectual property by hosting their computing infrastructure there.
In addition, Vis-à-Vis makes several assumptions about the guarantees provided by a compute utility and the software executing within each user’s VIS. First, we assume that any compute utility that hosts a VIS supports a Trusted Platform Module (TPM) capable of attesting to the software stack in use by the VIS [91]. A TPM infrastructure within the cloud allows compute utilities to prove to their customers what software is executing under their account.

While such capabilities are rarely utilized at the moment, we believe that TPMs will be a commonly supported feature for utilities in the future. Vis-à-Vis may still function in the absence of TPMs, but can lead to a wide range of security problems that are inherent to open systems [33]. For Vis-à-Vis, an attested software stack will allow nodes to prove to each other that they are executing correct protocol implementations.

Vis-à-Vis also assumes that users’ VISs are well administered and free of malware. Users, or preferably providers of managed VIS services, must properly configure the access-control policies of their software, and install the latest available security patches.
As with other networked systems, software misconfiguration and malware are serious threats to Vis-à-Vis, but are orthogonal to the focus of our design. If an adversary gains control of a user’s VIS it would not only gain access to the user’s own personal data, but could potentially access others as well by masquerading as the victim.

With this trust-and-threat model in mind, we now discuss the design of Vis-à-Vis in greater detail.

2.3.2 Membership service

A group’s membership service is initially implemented by the group founder’s VIS, although multiple VISs can participate in the membership service if the group grows. The primary function of the membership service is to provide a gateway to the location tree, in which group members’ location information is maintained. New group members attempt to join a group through the membership service executing on the group owner’s VIS. The owner’s IP address is distributed out of band in the same manner as the owner’s public key.

Access to the location tree can either be open to all requests or guarded, depending on the admission policy of the group. For example, mimicking the admission policies of Facebook’s university networks, a group’s membership service could require evidence of access to an email address within a specific domain. If the membership service receives sufficient evidence, it can issue a group capability in the form of a secret key. Subsequent requests among group members could be authenticated using this capability.

2.3.3 Location trees

Vis-à-Vis applies the distributed-systems techniques of consensus, leases, and hierarchy [68] to provide efficient, fault-tolerant, and scalable range queries over group members’ location information. Group-specific location information in Vis-à-Vis is accessed through each group’s location tree. A location tree is a hierarchical routing structure designed to efficiently and scalably support range queries over user locations. Unlike other
Figure 2.2: User $U_1$’s view of a group’s location tree.

data structures which use an arbitrary location region to partition a map, location trees use hierarchical political divisions. Since the divisions of a map are already known, the levels of the tree can be statically computed. Figure 2.3.3 shows an example tree. The top level represents countries, followed by states, counties, cities, and places. Leaf nodes represent users. In this example, user $U_1$ is in place $P_1$, in the city of Durham, within Durham County, in the state of North Carolina (NC), in the United States (US).

Each member’s VIS occupies exactly one leaf node, regardless of the granularity at which she shares her location. The only constraint users face when inserting their VIS is that it must be a leaf node within the subtree covering its shared location. If a user’s shared location is coarse, it may become a leaf node randomly below the corresponding interior node.

A potential danger of using static political divisions is that the finest-grained regions could be vulnerable to flash crowds. For example, places defining a sports arena would be unlikely to scale up to venue capacities of tens of thousands of users. To avoid such problems, Vis-à-Vis could easily apply techniques described by Census [40] and P2PR-Tree [78] for dynamically re-partitioning geographic regions into smaller subregions. Because of its hierarchical structure, any dynamic decomposition of geographic space would be hidden from higher levels of the tree.
Routing state

Each node maintains a subgraph of the tree, giving it partial knowledge of the group membership and members’ locations. We have optimized for the expected common case of queries about nearby locations by having nodes store more information about closer VISs than farther VISs, thereby ensuring that queries complete in fewer hops on closer regions than farther regions. For example, in Figure 2.3.3, user $U_1$ maintains a list of its siblings in $P_1$ (i.e., $U_2$ to $U_m$) and their shared locations. Nodes also maintain a list of regional coordinators for each level of the tree along their path to the root. A coordinator is a VIS whose shared location is within the corresponding region.

Coordinators are identified through a distributed consensus protocol such as Paxos [66]. The coordinators are elected by and from the pool of immediate child VISs. For example, in Figure 2.3.3, the coordinator for $P_1$ is elected from the pool of $U_1, U_2, \ldots, U_m$. Similarly, the coordinator for the city of Durham is elected by and from the coordinators of $P_1, P_2, \ldots P_n$, the coordinator for Durham county is elected by and from the coordinators of cities in Durham county, and so forth. A top-level coordinator serves at all levels of the tree.

In Figure 2.3.3, user $U_1$ maintains a list of coordinator VISs for each of the following: places $P_1$ to $P_n$ in the city of Durham, all cities in Durham County, all counties in North Carolina, all states in the US, and all countries. To retrieve a list of user locations in Monmouth County, New Jersey (NJ), US, user $U_1$ forwards a search request to the NJ coordinator. Because this VIS is sharing a location in NJ, it will have a pointer to the Monmouth County coordinator (if there are any users in this region), and forwards the search request to this VIS.

It is the responsibility of the Monmouth County coordinator to collect the list of users sharing their location within its county. The Monmouth coordinator has pointers to the coordinators for all cities in the county, which in turn have pointers to all places within those
cities. Thus, the Monmouth coordinator forwards the search request to the coordinators for all cities in Monmouth, each of which forwards the request to the coordinators of all places in their cities. The place coordinators finally return lists of their leaf VISs and their shared locations to the Monmouth coordinator.

Unless the Monmouth coordinator knows the number of places in the tree that are populated, it cannot know when all results have been returned. One way to address this would be to use a time-out, such that the coordinator waits for a fixed period of time before returning an answer to a query. However, this would require coordinators to wait for that period on every request, even if all answers had been received. Instead, coordinators maintain extra information with their parents about the number of populated places below them in the tree. In our example, this would allow the Monmouth coordinator to know how many query responses to expect from the place coordinators. Once all of the responses are received, the Monmouth coordinator combines the search results and returns them to U1.

Tree maintenance

Because coordinators’ identities are replicated within the routing state of all group members, it is important for members to have a consistent view of which VIS is a coordinator for which region, even as VISs leave the group or change locations. Vis-à-Vis maintains the consistency of this state using leases. VISs obtain a lease for their role as coordinator and periodically multicast lease-renewal messages down the tree as long as they continue to serve.

Coordinator failures are detected through explicit withdrawals (in the case of a user changing locations) or expiring leases (in the case of unplanned disconnections). Thus, in addition to maintaining a list of coordinators along the path to the root, each VIS must also maintain the lease expiry time for any coordinator they would be a candidate to replace. If a coordinator fails to renew their lease, a new election is held and election results are multicast down the tree. For example, in Figure 2.3.3, leaf nodes U1, U2, …, Um would main-
tain leasing state for the coordinator of $P_1$, the coordinators for $P_1, P_2, \ldots, P_n$ would maintain leasing state for the coordinator of the city of Durham, and so forth.

Similarly, it is important for VISs sharing the same place coordinator to have a consistent view of their siblings. Thus, VISs also maintain expiry state for their siblings. Nodes periodically renew their lease with their siblings via multicast. If a sibling’s lease expires without renewal, the sibling is assumed to have failed.

2.3.4 Operations

Vis-à-Vis groups implement each group operation—create, join, remove, update, and search—using the leasing and routing state maintained by VISs.

- **create**: To create a new group, a user distributes the group descriptor and the IP address of her VIS as needed. The owner’s VIS provides the membership service for the group, which is similar to Facebook group founders being automatically made group administrators.

- **join**: To join a group, a VIS contacts the group’s membership service and asks for the addresses of the top-level coordinators. If admitted into the group, the VIS then uses these top-level hosts to recursively identify the coordinator of the place where it wishes to become a leaf node.

  If the coordinator for the place exists, the new VIS notifies the coordinator of its presence. The coordinator then multicasts the IP address and shared location of the new VIS to other VISs in the region, who forward their information to the joining node. On the other hand, if the place coordinator or any coordinators along the path to the root do not exist, the joining VIS elects itself the coordinator of these previously unpopulated regions. Notification of these updates are forwarded to the appropriate regions.
• remove: To remove itself from a group, a VIS sends a purge message to its sibling VISs, removing it from their routing state. A VIS can also remove itself by allowing its leases to expire.

• update: If a location update does not change a VIS’s place, the VIS simply multicasts its new location to its siblings as part of its lease-renewal messages. If a location update requires moving to a new region, the node purges itself from its siblings’ routing state (either explicitly or via an expired lease), looks up the coordinator for the new region, and notifies the new coordinator of its location.

• search: Search is performed in two stages. First, a VIS decomposes the requested bounding-box region into the smallest set of covering subregions. For each subregion in this set, the VIS looks up the coordinator, and if the coordinator exists, sends it a search request. Search requests received by coordinators are satisfied using the recursive procedure described in Section 2.3.3.

2.4 Implementation

We built a Vis-à-Vis prototype based on the design described in Section 2.3 and deployed it on Amazon EC2. We also modified Apache ZooKeeper to support namespace partitioning. Finally, we created a mobile phone application called Group Finder to test and evaluate our Vis-à-Vis prototype. This section describes these software implementations.

2.4.1 Vis-a-Vis and ZooKeeper

Our prototype Vis-à-Vis implementation is written in Java and consists of over 3,300 semi-colon terminated lines of code and 50 class files.

Software for maintaining location-tree state is a multithreaded application that maintains its position in the tree by exchanging heartbeats with other nodes and participating in coordinator elections. This software also serves incoming requests from the client, e.g.,
update the shared location, leave or join the tree. Our implementation supports both IP- and application-level multicast. However, since Amazon EC2 does not support IP multicast, we use application-level multicast for the experiments reported in Section 2.5.

We use ZooKeeper [61] for our coordinator election service. In our Vis-à-Vis prototype, the ZooKeeper server runs on the group founder’s VIS. However, ZooKeeper supports clustering and can be run on multiple VISs. ZooKeeper includes a leader election function based on Paxos [66], which we use for our coordinator election. We have modified ZooKeeper to support optional namespace partitioning. This allows us to span the coordinator election service across multiple datacenters, improving performance and reducing the number of expensive roundtrips across long distances. For example, if the tree consists of a number of VISs in Europe and North America, then it is beneficial to use two ZooKeeper instances: one in Europe and another in North America. Each ZooKeeper instance is responsible for a particular namespace, e.g., EU or US, and serves requests from the same continent. However, if a European VIS wants to join a location in the US, the EU-based ZooKeeper would redirect this VIS to a US-based ZooKeeper instance.

Finally, each VIS runs a Tomcat server. Low-level Java Servlet APIs provide an interface that supports the group operations described in Section 2.2. High-level APIs provide application-specific operations. These APIs only accept requests from the owner of the VIS.

### 2.4.2 Group Finder mobile application

We also developed a companion iPhone application for Vis-à-Vis called Group Finder. The application allows a user to submit her location along with a status message to a group she belongs to, and retrieve the last known locations and status messages of the other group members. We refer to the operation of submitting a location and status message to a server as a “check-in”.

A screenshot of Group Finder is shown in Figure 2.3. The current location of the
user is shown as a blue dot and the most recent check-in locations of the group members as pins. Selecting a pin shows the corresponding group member’s photo, his last status message, and the time since his last update. The buttons at the top of the screen allow the user to check in or retrieve information about the latest check-ins from group members. Just below the buttons, Group Finder displays the name of the currently selected group.

In our implementation, each user’s mobile phone communicates only with that user’s VIS. Location updates and status messages are shared only within the current group. Retrieving the latest check-in locations of group members is implemented in the VIS as a call to the search operation, with the location bounds equal to the map area shown on the screen. Check-ins invoke the update operation.

We used Group Finder to debug the Vis-à-Vis framework and measure end-to-end latencies over 3G and WiFi networks. We report on these findings in Section 2.5.
2.5 Evaluation

In our experimental evaluation of Vis-à-Vis, we sought answers to the following three questions:

- How well does our Vis-à-Vis prototype perform the core group operations of join, update, and search?

- How well does our Vis-à-Vis prototype perform when the nodes are geographically spread across continents?

- How well does our Group Finder application perform when communicating with Vis-à-Vis over WiFi and 3G networks?

We wanted to characterize the performance of our decentralized approach to managing information in location-based OSNs. For comparison, we also implemented the Vis-à-Vis group abstraction using a centralized approach. The centralized service is a conventional multi-tiered architecture, consisting of a front-end web-application server (Tomcat server) and a back-end database server (MySQL server). We expected the centralized server to provide lower latency than our decentralized Vis-à-Vis prototype, but wanted to determine whether our prototype’s performance was competitive.

We ran micro-benchmarks, studied the effect of geographic distribution of VISs, and measured end-to-end latencies using our Group Finder application. All experiments used EC2 virtual machines as VISs. Each virtual machine was configured as an EC2 Small Instance, with one virtual core and 1.7 GB of memory. Our centralized service ran on the same type of virtual machine in the same data center.

The location tree for all experiments was based on our Group Finder app, which uses an $8 \times 8$ grid for its finest-grained locations. The resulting location tree had four levels (including leaf nodes) and four coordinators per level.
2.5.1 Micro-benchmarks

For our micro-benchmarks, we wanted to measure the latency from an external host to complete the join, update, and search operations in both Vis-à-Vis and our centralized implementation. We measured latency at a wired host at Duke from the time an operation request was issued to the time it completed. VISs were hosted within the same Amazon data center on the east coast of the US, where the round-trip latency between machines varied between 0.1 and 5ms.

For these experiments, we varied the group size from 10 to 500 members, and assigned each member a separate VIS. VISs inserted themselves into the tree as randomly-placed leaf nodes. For each experiment, we report the mean time over 20 trials to complete each operation, including network, DNS, and server latency. In all figures, error bars denote standard deviations. Due to occasional, transient spikes in the time to complete DNS lookups, we did not include some outliers in our micro-benchmark results. The vast majority of DNS lookups completed within tens of milliseconds, but occasionally lookups
took hundreds of milliseconds or timed out altogether. We attribute these spikes to well
documented problems specific to Amazon EC2 [112], but since the spikes were rare we
did not thoroughly examine them. Thus, when network latency was more than an order of
magnitude higher than normal, we removed 1) these high-latency trials, and 2) an equal
number of the lowest-latency trials.

Figure 2.4 shows the average latency to complete join and update operations for our
Vis-à-Vis prototype and our centralized implementation. In the centralized case, join and
update are identical: both essentially insert a new value into the MySQL database. For
Vis-à-Vis, join is more expensive than update because the VIS must initialize its routing
state before registering its own location with a coordinator.

As expected, the centralized implementation had lower latencies than Vis-à-Vis, with
update operations completing in approximately 20ms for all group sizes. Nonetheless,
our decentralized Vis-à-Vis implementation performs reasonably well, with join operations
completing in approximately 400ms and update operations completing in under 100ms for
all group sizes. These results demonstrate two important properties of Vis-à-Vis: 1) that
join and update provide reasonable efficiency, even though the centralized case is faster,
and 2) that join and update scale well to large group sizes.

To understand search performance, we investigated two cases. In the first case, we
performed local search operations within a single fine-grained region. These searches
only required communication with one coordinator VIS. In the second case, we performed
search operations across the entire group so that the locations of all group members were
retrieved. These searches required contacting all coordinators in the tree and represent a
worst case for Vis-à-Vis.

Figure 2.5 shows the average latency to perform a local search for the centralized
implementation and Vis-à-Vis. As expected, both perform well, returning results in under
100ms and easily scaling to 500-member groups. Recall that VISs inserted themselves
randomly into the tree, so that in the case of a 500-member group, low-level coordinators
Figure 2.5: Local search performance.

Figure 2.6 shows the average latency to perform a group-wide search for the centralized implementation and Vis-à-Vis. Again as expected, Vis-à-Vis’s decentralized approach has higher latency than the centralized case for group-wide searches since queries require communicating with multiple coordinators. However, Vis-à-Vis’s latency plateaus at around 200ms for all groups larger than 100, while the centralized approach experiences increased latency for 500-member groups. Like the join and update micro-benchmarks, these results demonstrate 1) that Vis-à-Vis’s performance is reasonable, even when compared to a fast centralized implementation, and 2) that Vis-à-Vis scales well, even in the worst case when all coordinators must be contacted.

Note that our maximum group size of 500 corresponds to the majority of Facebook groups, even though many are much larger [94]. We would have liked to generate results for groups of 1,000 or more VISs, but could not due to EC2 limitations. Nonetheless, given the results of our micro-benchmarks, we see no reason why even our unoptimized
Figure 2.6: Group-wide search performance.

Table 2.1: Group Finder mean latency over WiFi in milliseconds, with standard deviations in parentheses.

<table>
<thead>
<tr>
<th>Group Size</th>
<th>Latency Component</th>
<th>Decentralized search</th>
<th>Decentralized update (far away)</th>
<th>Decentralized update (nearby)</th>
<th>Centralized search</th>
<th>Centralized update</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 Network</td>
<td>169 (98)</td>
<td>365 (97)</td>
<td>295 (225)</td>
<td>211 (57)</td>
<td>263 (177)</td>
<td></td>
</tr>
<tr>
<td>10 Server</td>
<td>36 (14)</td>
<td>297 (37)</td>
<td>14 (15)</td>
<td>10 (3)</td>
<td>8 (1)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>205</td>
<td>662</td>
<td>309</td>
<td>221</td>
<td>271</td>
<td></td>
</tr>
<tr>
<td>100 Network</td>
<td>376 (166)</td>
<td>322 (57)</td>
<td>362 (89)</td>
<td>311 (87)</td>
<td>287 (37)</td>
<td></td>
</tr>
<tr>
<td>100 Server</td>
<td>160 (28)</td>
<td>295 (40)</td>
<td>10 (9)</td>
<td>68 (50)</td>
<td>9 (3)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>536</td>
<td>617</td>
<td>372</td>
<td>379</td>
<td>296</td>
<td></td>
</tr>
</tbody>
</table>

Prototype would not scale to tens of thousands of nodes.

2.5.2 Effect of geographic distribution of VISs

To study the effect of geographic distribution of VISs on Vis-à-Vis, we built a location tree with nodes hosted at two Amazon EC2 data centers located in distant geographic locations: 50 nodes in the US and 50 nodes in the European Union (EU), specifically Ireland. We left the group’s membership service in the US zone, and ran one ZooKeeper instance in each geographic zone with a partitioned namespace. We used the same client machine at Duke.
Table 2.2: Group Finder mean latency over 3G cellular in milliseconds, with standard deviations in parentheses.

<table>
<thead>
<tr>
<th>Group Size</th>
<th>Latency Component</th>
<th>Decentralized search</th>
<th>update (far away)</th>
<th>update (nearby)</th>
<th>Centralized search</th>
<th>update</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Network</td>
<td>870 (918)</td>
<td>3004 (3678)</td>
<td>1673 (1104)</td>
<td>1103 (905)</td>
<td>1280 (1845)</td>
</tr>
<tr>
<td></td>
<td>Server</td>
<td>44 (12)</td>
<td>438 (184)</td>
<td>27 (15)</td>
<td>7 (1)</td>
<td>7 (1)</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>914</td>
<td>3442</td>
<td>1700</td>
<td>1110</td>
<td>1287</td>
</tr>
<tr>
<td>100</td>
<td>Network</td>
<td>2812 (3428)</td>
<td>3873 (6003)</td>
<td>2294 (3098)</td>
<td>1940 (1263)</td>
<td>1363 (1160)</td>
</tr>
<tr>
<td></td>
<td>Server</td>
<td>155 (41)</td>
<td>385 (71)</td>
<td>28 (17)</td>
<td>8 (6)</td>
<td>31 (33)</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>2967</td>
<td>4258</td>
<td>2322</td>
<td>1948</td>
<td>1394</td>
</tr>
</tbody>
</table>

as in Section 2.5.1. The round-trip latency between the US- and EU-based nodes varied between 85 and 95ms.

We compared two different methods of constructing the tree. The first is a random assignment, where a VIS joins a random location. This method is expected to have poor performance since an EU-based VIS has to contact a US-based ZooKeeper instance when joining a US-based location, and possibly a number of US-based coordinators. The second method is proximity-aware assignment. This scenario assumes that US- and EU-based VISs are more likely to join a ZooKeeper instance near their respective published locations. This assignment should have better performance since a EU-based VIS will mostly interact with EU-based servers. However, both of these methods incur unavoidable overhead from the network latencies between the US-based client machine and the EU-based VISs, and between the EU-based VISs and the US-based group’s membership service.

Figure 2.7 shows the latencies of the join and search operations on a cross-continental 100-node tree with 50 nodes in Europe and 50 nodes in North America. In the case of random assignment, we measured the latencies of an EU-based VIS joining a random (EU or US) location of the tree. For the proximity-aware assignment, we measured the latency of a EU-based VIS joining an EU location of the tree. For both scenarios we measured the local search operation performed by an EU-based VIS.

As expected, latencies using the random assignment method are longer than those using the proximity-aware method. However, even the shorter latencies are longer than...
Figure 2.7: Effect of geographic distribution of VISs.

2.5.3 End-to-end latency

The goal of our final set of experiments was to measure the end-to-end latency of a mobile application using Vis-à-Vis. These experiments were performed at Duke using our Group Finder iPhone application. We instrumented the Group Finder and server code to measure the latencies of checking in and retrieving group members’ locations. We measured the network and server latency to complete these tasks while varying the following parameters: network type (WiFi or 3G cellular), group size (10 or 100 members), and architecture (Vis-à-Vis or a centralized server).

During a check-in, the user’s client uploaded a location via an update call to a server: her VIS in the Vis-à-Vis case and the centralized server in the other. For Vis-à-Vis experiments, the user’s VIS synchronously translated the user’s location to the correct coordinator, notified the coordinator of the user’s new location, then returned control to the mobile device. A user retrieved group members’ locations through a search call. In Vis-à-Vis
this call propagated queries down the location tree, and in the centralized case it led to a MySQL query.

As with the micro-benchmarks, we report the mean and standard deviation of latencies across 20 trials. However, unlike with the micro-benchmarks, we do not remove outliers for our end-to-end results. This is because spikes in 3G network latency are common and removing outliers from the wireless experiments would have given an inaccurate view of Vis-à-Vis’s performance.

Table 2.1 shows the time for Group Finder to retrieve the locations of a group’s members (i.e., to complete a search operation) over WiFi, broken down by network and server latency. The increased server latency for 100 group members in the decentralized case reflects the need to communicate with more coordinators. The server latency is comparable to our search micro-benchmarks for groups with 100 members.

Table 2.1 also shows the time for a Group Finder user to check-in (i.e., to complete an update operation) over WiFi. For the decentralized setup, we measured the time to check-in to a region that is far away, which required contacting several coordinators, and the time to check-in to a nearby region, which required contacting a single coordinator. As with our micro-benchmarks, group size had little effect on search latency. Also as expected, checking in to a nearby location was faster than checking in to a far-away location.

These results demonstrate that the performance of common tasks in Group Finder, such as retrieving members’ locations and checking in to a nearby location, are dominated by network latency rather than server latency. Only in the uncommon case when a user checks in to a far away region would server latency approach network latency under WiFi.

Table 2.2 shows the latency of the same Group Finder tasks using the iPhone’s 3G cellular connection. These results show that 3G network latency was often an order of magnitude greater than that of WiFi. In addition, standard deviations for 3G latency were often greater than the means themselves. As a result, server latency was small relative to network latency in all our 3G experiments.
Overall, our end-to-end experiments demonstrate that for mobile applications such as Group Finder, performance is often dominated by the latency of wireless networks rather than that of Vis-à-Vis’s back-end services. This effect reduces the perceived performance differences between Vis-à-Vis and centralized OSNs.

2.6 Conclusion

We have presented the design, implementation, and evaluation of Vis-à-Vis, a decentralized framework for online social networks. Vis-à-Vis is based on a federation of independent Virtual Individual Servers, machines owned by individuals and preferably running in a paid, virtualized cloud-computing infrastructure. Vis-à-Vis preserves privacy by avoiding the pitfalls of the prevailing OSN model based on centralized free services. We focused on Vis-à-Vis’s use of distributed hierarchies to provide efficient and scalable location-based operations on social groups. We deployed a Vis-à-Vis prototype in Amazon EC2 and measured its performance against a centralized implementation of the same OSN operations. Our results show that the latency of our decentralized system is competitive with that of its centralized counterpart.
3

Building a privacy-preserving OSN platform using commodity computers

In this chapter we describe Confidant [71] – a system that takes advantage of the trust embedded in the social network to replicate cleartext data on the machines of close friends for building a general-purpose OSN platform.

3.1 Introduction

Online social networks (OSNs) such as Facebook, MySpace, and Twitter have enhanced the lives of millions of users worldwide. Facebook has surpassed 500 million active users per month and already attracts 32% of global Internet users every day, with the average user spending 32 minutes each day on the site [49, 50]. The aggregate volume of personal data shared through Facebook is staggering: across all users, Facebook receives nearly 25 billion new items each month. Such high levels of participation should not be surprising; these services are fun, useful, and free.

OSN users also trust providers to manage their data responsibly, mining it internally for targeted advertising, and otherwise enforcing user-specified access policies to protect
their profiles, messages, and photos from unwanted viewing. Unfortunately, behavior by OSN providers has not met these expectations. In late 2009, Facebook unilaterally eliminated existing restrictions on users’ friend lists and other information by making them world-readable. In addition, the site’s privacy “transition tool” actively encouraged users to replace restrictions limiting access to “Networks and Friends” with the more permissive “Everyone” option. Similarly, Google revealed many users’ most-emailed Gmail contacts by making their Buzz friend list world-readable. Both services eventually reinstated previous access policies after heavy criticism, but many users’ sensitive information was exposed for days or weeks.

With OSNs now central to many people’s lives, it is critical to address the rising tension between the value of participation and the erosion of privacy exhibited by existing services. Users want to continue enjoying OSNs, but they also want to control their data and limit the trust placed in service providers. Decentralized OSNs offer the hope that these goals can be met while potentially improving the functionality and scalability offered by today’s successful centralized services.

Several research groups have recently proposed decentralized OSN architectures [26, 109, 85]. All of these systems are based on the assumption that the storage servers where OSN data resides are untrusted: data is encrypted before it is stored and plaintext is only viewable by clients with the appropriate decryption keys. This approach is appealing because encrypted data can be stored anywhere, including open, peer-to-peer systems such as OpenDHT, cloud services such Amazon S3, or even existing OSNs such as Facebook.

However, we observe that reliance on untrusted storage comes at a price. Scalable data-processing of large data sets such as OSN data is the basis for a growing number of cloud services [42, 83, 46], but if remote servers cannot be trusted to view plaintext data, all data must be retrieved and decrypted by a client itself before it can be operated on. For mobile and desktop clients alike, this can lead to bandwidth, storage, and compute scalability problems. A partial solution is to apply techniques for performing queries over
encrypted data [45, 97, 103, 85], which rely on specialized indexes based on pre-defined features such as key words. Unfortunately, these techniques are limited to feature-based search, and cannot scalably support more advanced data-processing applications such as the trending-topics feature of Twitter, face-recognition analysis of Facebook photos [46], and photo-mosaics generated from flickr and Facebook photos [83].

The central question of this chapter is: how can decentralized OSNs support scalable data-processing? To answer this question we present the design and implementation of Confidant. Confidant’s approach to decentralized OSNs is to use information from the social graph so that users’ data can be stored in plaintext by machines that they trust. The intuition behind our approach is that since users’ friends already have read access to their OSN data, why not allow them to serve it as well?

Confidant participants use commodity machines such as personal desktop PCs or enterprise workstations as storage servers, and create replicas on machines controlled by a subset of their friends. Trust relationships between users can be exposed to Confidant in many ways, such as by mining features of existing OSNs (e.g., the level of interaction between users or the number of overlapping friends), or by asking users to identify trusted friends by hand.

The key technical challenge for Confidant is managing OSN data and access-control policies under weakly-consistent replication. As in prior work [87, 115], Confidant ensures eventual consistency among replicas and treats objects and access policies as immutable first-class data items. However, managing data shared within an OSN presents a different set of usability and access-control requirements than those addressed previously. First, OSN users commonly create new OSN data from multiple clients and should not have to deal with the inconvenience of manually resolving conflicts. Confidant eliminates conflicts without compromising data confidentiality or integrity by serializing a user’s updates through a highly available and lightweight state manager hosted in the cloud. In addition, OSN users who inadvertently mis-share an item must be allowed to recover from their
mistake without perturbing the policies protecting other data items. Confidant addresses this issue through flexible rebinding of access policies to data items.

To summarize, this chapter makes the following contributions. Confidant’s design represents the first decentralized OSN architecture to leverage trustworthy storage servers based on inter-personal relationships. This trust enhances decentralized OSNs by enabling a scalable data-processing framework for third-party applications. Confidant also provides an access-control scheme for weakly-consistent, replicated data that is tailored to the needs of OSN users. In particular, Confidant eliminates write conflicts and allows participants to recover from access-control mistakes by binding access policies to individual data items rather than to groups of items with the same label.

Finally, we have evaluated Confidant using trace-driven simulations and experiments with a prototype. Our simulation results show that typical OSN users should expect read and write success rates of between 99 and 100%. Experiments with our prototype show that data-processing tasks such as remote keyword search, trending topics, and face detection will scale well; Confidant scripts for processing status updates and photos from 100 friends are between 3 and 30 times faster than an approach relying on untrusted remote storage.

The rest of this chapter is organized as follows: Section 3.2 gives a high-level overview of the Confidant architecture, Section 3.3 describes Confidant’s design, Section 3.4 describes our prototype implementation, Section 3.5 presents an evaluation of our design and prototype implementation, and Section 3.6 provides our conclusions.

3.2 Overview

This Section provides a high-level overview of the Confidant architecture and trust model.
3.2.1 Architecture

Figure 3.1 shows the Confidant architecture, including paths for posting new data (steps 1-2), retrieving new data (steps 3-5), and performing remote data processing (steps 6-8). Keeping costs low is a critical design goal for Confidant, since existing OSNs such as Facebook are popular in large part because they are free. As a result, Confidant relies on two low-cost forms of infrastructure: desktop and enterprise PC storage servers, and lightweight cloud-based name servers. PCs and workstations are a sunk cost for most users and free cloud services such as Google AppEngine and Heroku allow a user to execute a single-threaded server process while maintaining a small amount of persistent state. We assume that every Confidant user controls both a storage server and name server.
A storage server hosts a user’s OSN data and authenticates client read and write requests. Because storage servers can experience transient failures, a user may select a small number of her friends’ servers to host *replicas* of her data. Each replica manages a copy of the user’s data and participates in anti-entropy protocols to ensure eventual consistency. Storage servers also allow authorized users to run sandboxed data-processing *scripts* directly on their hardware.

Name servers are assumed to be highly available, but due to resource and trust constraints have limited functionality. A name server is only responsible for maintaining two pieces of state: its owner’s *logical clock*, and a list of available replicas. Maintaining this state in a highly-available, centralized location is appealing for two reasons. First, placing the list of available replicas in a stable, well-known location simplifies data retrieval. Second, maintaining logical clocks in centralized locations allows Confidant to serialize a user’s updates and eliminate write conflicts.

Eliminating conflicts is important because users are likely to access Confidant data via multiple *clients*. A single physical machine such as a laptop can serve as both a storage server and a client, but we explicitly separate client and server roles due to the resource constraints of clients such as mobile phones. Client functionality is limited to uploading and retrieving content and injecting data-processing scripts.

### 3.2.2 Trust and threat model

Trust in Confidant is based on physical control of hardware and inter-personal relationships.

Users trust their clients to read their data, create new data on their behalf, and update the access policies protecting their data. A user’s clients are not allowed to create objects on behalf of other users or alter the access policies protecting other users’ data.

We assume that a user’s storage server and its replicas will participate in anti-entropy protocols, enforce access policies, and correctly run data-processing scripts. Correct exe-
cution of scripts requires storage servers to access plaintext copies of data and to preserve the integrity of a script’s code, runtime state, and input data. To ensure that replicas meet these trust requirements, Confidant users only place their data on servers controlled by trusted friends.

Serving a user’s data from her friends’ PCs rather than from third-party servers creates no additional threats to data confidentiality than existing centralized and decentralized OSNs. Users already share their OSN data with their friends and, as with all other OSNs, we assume that users do not collude or share data with unauthorized entities. Software misconfiguration and malware are serious problems for user-managed machines, but these vulnerabilities are not unique to Confidant. If an attacker compromises a Facebook user’s PC or mobile device, the attacker can use the Facebook credentials stored on the machine to access any data the owner is authorized to view. Decentralized OSNs such as Persona are also vulnerable to attacks on client machines.

However, even if a user is trusted to preserve the confidentiality of their friend’s data, their storage server might not be trusted to preserve the integrity of data-processing scripts. An adversarial storage server can corrupt script results by modifying a script’s execution, injecting false data, or removing legitimate data. To reduce the likelihood of corrupted script results, we assume that users can identify a small number of friends whose storage servers will behave as expected. Based on several proposals to gauge the strength of social ties using the level of interaction between OSN users [51, 37, 113], it is reasonable to assume that users will find enough storage servers to act as replicas. For example, Facebook has reported that male users regularly interact with an average of seven friends, while women regularly interact with an average of ten friends [43].

Limiting the trust users must place in cloud-based services is an important design goal for Confidant. Thus, our cloud-based name servers are trusted to correctly maintain information about storage servers’ availability and a user’s logical clock, but are not trusted to access plaintext data. Confidant is agnostic to the mechanism by which users become
aware of new data, though for convenience and incremental deployability our prototype implementation uses Facebook. Services like Twitter or open alternatives built on top of OpenDHT could also be plugged in. Regardless of what notification service is used, Confidant only trusts machines controlled by friends to access plaintext data.

3.3 Design

In this section, we describe the Confidant design.

3.3.1 Cryptographic state

Confidant encodes component roles and trust relationships using techniques described by work on Attribute-Based Access Control (ABAC) [114]. We do not support the full power of an ABAC system, and have adopted a subset of techniques (e.g., independently rooted certificate chains and signed attribute-assignments) that are appropriate to our decentralized OSN.

Principals in Confidant are defined by a public-key pair, and users are defined by a public-key pair called a root key pair. Users generate their own root keys, and distribute their root public key out of band (e.g., through their Facebook profile or via email). A user’s root public key is distributed as a self-signed certificate called a root certificate; the root private key is kept in a secure, offline location. Through her root key-pair, a user also issues certificates for her storage server (storage certificate), name server (name certificate), and any clients under her control (client certificate): these certificates describe the principal’s role (i.e., storage server, name server, or client) and its public key. For each certificate signed by a user’s root key pair, the matching private key is only stored on the component, and expiration dates are set to an appropriate period of time. Users also generate replica certificates with their root key pair for any storage servers controlled by others who are authorized to serve their objects. All certificates are distributed out of band through a service such as Facebook or via email.
Users encode their inter-personal relationships through groups. A group is defined by four pieces of state: 1) a unique user who owns the group, 2) a list of users making up the group’s membership, 3) an attribute string, and 4) a secret key. Group owners are the only users who can update a group’s state. Group memberships grow monotonically; “removing” a member requires an owner to create a new group with the previous membership minus the evicted member.

A group’s string provides a convenient mechanism for assigning attributes to sets of users, which can in turn be used to express access-control policies over sensitive data. For example, user Alice may wish to define groups with attributes such as “New York friends,” “college friends,” and “family.” Group membership does not need to be symmetric. Bob may be included in Alice’s group “New York friends,” but Bob is not obligated to include Alice in any of the groups he owns.

Group keys are generated on the group owner’s storage server and distributed as social attestations [109]; attestations are signed using the key pair of the owner’s storage server. Social attestations in Confidant are nearly identical to those in Lockr, except that Confidant attestations also enumerate their group’s membership. Within a social attestation, group members are represented by the string description and public key found in their root certificate. We include this information so that clients can reason about the level of anonymity provided by an attestation. When authorizing herself with an attestation, a user can determine if the attestation’s group size will sufficiently hide her identity to the storage server handling her request. If new members are added to the group, the group owner distributes a new social attestation to members reflecting the larger group size.

New and updated attestations are distributed epidemically among storage servers; clients periodically synchronize their set of attestations with their owner’s storage server. Although there is no bound on the time for a client to receive an attestation, the key embedded in an existing attestation can remain valid even if the membership enumerated in the attestation becomes stale. Importantly, an attestation containing a valid key and stale
membership will not give users’ a false sense of anonymity since groups grow monotonically.

Access policies specify the groups that are allowed to view an object, and are represented by logical expressions in disjunctive normal form (e.g., \((g_0 \land g_1) \lor (g_2 \land g_3)\) ), where each literal describes a group, and each conjunction indicates a set of group keys that could be used for authorization. For anonymity reasons, our policy expressions do not support a negation operator since users would have to provide their full identity to a storage server to prove when they are not a member of a group.

Finally, because social attestations contain a secret key, they must be handled carefully. Name servers cannot access social attestations since the machines on which name servers execute are physically controlled by a cloud provider rather than a user. Storage servers store copies of any attestations needed to authenticate access requests. Clients store any attestations required to upload new objects or access friends’ objects.

Key revocation is known to be a difficult problem, and we use a reasonable set of techniques to address it. First, certificates and social attestations include an expiration date. If a private or secret key leaks, the certificate expiration date provides an upper bound on the key’s usefulness. For group keys, users can generate new keys and attestations to protect any new objects they create. Storage servers can also be asked to ignore group keys that become compromised. This approach should scale well since the number of storage servers hosting a user’s data is expected to be on the order of ten machines. We discuss how new policies can be assigned to old objects in the next section.

3.3.2 Objects and access policies

Data in Confidant is managed as items. Confidant supports two kinds of items: objects and access policies.

The unit of sharing in Confidant is an object. Like Facebook wall posts, comments, and photos, or Twitter tweets, objects are immutable. Every object has a unique descriptor
with the following format:

\{owner, seq, acl\}

Descriptors function strictly as names (i.e., not capabilities) and can be embedded in feeds from untrusted services such as Facebook and Twitter without compromising data confidentiality.

The owner and seq fields uniquely identify the object. The owner field of a descriptor is set to the root public key of the user whose client created the object. The seq field is the object’s sequence number. Each number is generated by a user’s name server when an item is created and is unique for all items created by a user. The acl field is an expression in Confidant’s access-policy language.

Objects consist of a meta-data header, followed by the object’s content:

\{owner, seq, typ, t, len, \textlangle data\textrangle\}

The owner and seq fields are identical to those present in the object’s descriptor. The typ field indicates the object’s format (e.g., text, or JPEG image), t field is a wall-clock timestamp. The end of the object is opaque data of length len.

The unit of protection in Confidant is an access policy. Access policies are treated as distinct data items with the following representation: \{owner, seq_{ap}, acl, seq_{obj}\}. As before, the acl field is an expression in Confidant’s access-policy language. seq_{ap} is the sequence number associated with the access policy; this number is unique across all items (objects and policies) created by a user. The owner and seq_{obj} fields of an access policy refer to the object to which the expression applies. To ensure that clients do not attempt to assign access policies to objects they do not own, storage servers must check that a new policy’s owner field matches the identity of the issuing client’s certificate signer.

Like objects, access policies are immutable, although the binding between objects and polices can change according to the following rule: an object is protected by the policy with the greatest sequence number that refers to the object. Application of this rule allows
clients to add and remove permissions using a single, simple mechanism. If a client wants to bind a new access policy to an old object, it increments its user’s logical clock and creates a new policy using the new sequence number. To invalidate an object, a client can issue a new access policy with a null expression. If two objects are protected by the same logical expression, they will require separate access policies. Note that because the binding between objects and policies can change the acl included in an object descriptor is meant only as a hint [67], and may not reflect the current protection scheme for the object.

There are two potential drawbacks of not aggregating policies across objects: the overhead of storing and transferring extra policy items, and the added complexity of bulk policy changes. However, both drawbacks are minor concerns, given the relatively small number of items that individual users are likely to generate. According to Facebook, the average user creates only 70 data items every month [50]. Even for bulk policy rebindings covering years’ worth of user data, iterating through all of a user’s items several times should be reasonably fast. As a result, the flexibility to bind arbitrary policy expressions to items at any point in the item’s lifetime outweigh the drawbacks.

Confidant’s approach to object protection is similar to recent work on managing access policies with Cimbiosys [87, 115]. Both systems manage data as a weakly-consistent replicated state store, and both treat objects and policies as first-class data items. Despite the similarities, there are several important differences between Confidant and Cimbiosys.

First, Confidant serializes all of a user’s updates by assigning new items a unique sequence number from the user’s name server. This eliminates the complexity and inconvenience of automatically or manually resolving concurrent updates. Avoiding the pain of handling conflicts is an important consideration for OSNs. Experience with the Coda file system found that most write conflicts were caused by users updating their data from multiple clients [81], which mirrors the common behavior of OSN users accessing services from both a PC and mobile device. Confidant’s name servers create single points of failure, but we believe that this is an appropriate tradeoff given inconvenience of handling
conflicts and the high availability of cloud services such as Google AppEngine.

Cimbiosys also applies access policies at a coarser granularity than Confidant. Cimbiosys access policies (called claims) are bound to labels rather than objects. Claims allow principals to read or write sets of objects that bear the same label (e.g., “photos” or “contacts”). However, because the labels assigned to Cimbiosys items are permanent and claims are expressed in terms of labels, it is impossible for users to change the permissions of a single item within a labeled set; permissions can only be granted or revoked at the granularity of complete sets of items with a particular label. While this is a reasonable design choice for the home-networking setting for which Cimbiosys was designed, it is inappropriate for OSNs.

As the authors point out, it is important for Cimbiosys users to label items correctly when they are created. This is too great a burden for OSN users, for whom fine-grained control is useful in many situations. For example, consider a user who initially labels an item “mobile photo” (perhaps accidentally) and shares it with her family and friends. Under Cimbiosys, if she later decided that it was a mistake to give her family access to the image, she would have to revoke her family’s access to all items labeled “mobile photo,” including any other images she might want to continue sharing with them. In Confidant, the user could simply create a new policy with a pointer to the image she would like to hide and a policy expression including only friends.

It should be noted that Cimbiosys could provide the same policy flexibility as Confidant by assigning each object a unique label, but the designers did not pursue this approach due to its perceived lack of efficiency. This design decision appears to be related to Cimbiosys’s focus on defining policy claims in terms of principals rather than groups of principals, and an implicit assumption about the number of objects a user owns. SecPAL (Cimbiosys’s policy logic) supports groups of principals, but without groups, creating a separate claim for each principal authorized to access each item in a massive data set might introduce scalability problems. Confidant avoids these problems because individual
user’s OSN data sets are relatively small, and by defining policies in terms of groups of principals.

3.3.3 Name servers

As described in Section 3.2.1, each user runs a name server within a low-cost cloud service such as Google AppEngine. Name servers manage two pieces of state: a list of IP addresses corresponding to online replicas and a logical clock. Entries in the list of IP addresses also include an expiration time, and become invalid if not updated in time. Name servers also maintain a list of storage servers authorized to act as replicas.

Retrieving a list of replicas is similar to a DNS lookup. Requests are unauthenticated, require no arguments from the caller, have no side-effects, and return the name server’s list of IP addresses, public-key pairs for each valid storage-server as well as the current value of the user’s logical clock. Name servers use SSL to preserve the integrity of queries.

Storage servers set the IP address where they can be reached by periodically contacting the name servers associated with the replicas they host. These calls refresh the expiration time of the server’s entry and allow the server’s IP address to be returned as part of a lookup. Expiration times are intended to be on the order of tens of minutes. Because only authorized storage servers should be allowed to serve as replicas for a user’s data, refreshing a server entry must be authenticated. Initial lease renewals require the name server and storage server to mutually authenticate and establish a session key using their signed certificates, but once the session key has been established it is used to authenticate future requests.

A name server’s logical clock is used to assign sequence numbers to items and to help replicas synchronize when they come back online. The value of the logical clock increases monotonically. When a client wants to assign a sequence number to a new item, it requests an increment; the name server responds by adding one to the existing value and returning the new value of the clock. Since only clients under the user’s control should be allowed
Table 3.1: Storage-server messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store request</td>
<td>{{g_0, g_1, \ldots, g_n}, cert_C, {replicas, obj, ap, rand, {hash(obj, ap)}}_K_C {{u_0, u_1, \ldots, u_n}} }</td>
</tr>
<tr>
<td>Policy update</td>
<td>{cert_C, replicas, ap, rand, {hash(ap)}}_K_C }</td>
</tr>
<tr>
<td>Retrieve request</td>
<td>{{owner, seq, {g_0, g_1, \ldots, g_n}}}</td>
</tr>
<tr>
<td>Retrieve response 1</td>
<td>{{g_0, g_1, \ldots, g_n}, cert_R, {obj, ap, rand, {hash(obj, ap)}}_K_R {{u_0, u_1, \ldots, u_n}}}</td>
</tr>
<tr>
<td>Retrieve response 2</td>
<td>{cert_R, ap, rand, {hash(ap)}}_K_R }</td>
</tr>
</tbody>
</table>

To advance the clock, increment requests are authenticated. As with entry-refresh requests, clients and name servers initially establish a session key with their signed certificates, and use the session keys to authenticate future requests.

3.3.4 Storage servers

Each Confidant user runs a storage server that contains plaintext copies of all of her objects and access policies. A storage server may also act as a replica for another user if the other user trusts the server’s owner to 1) read all of her objects, 2) enforce access policies, and 3) preserve the integrity of any data-processing scripts run on the server. As explained in Section 3.2.2, it is reasonable to assume that users can identify on the order of ten trustworthy replicas.

Once replicas have been selected, the user in control of each replica-set member must install its storage server’s public key at the data owner’s name server. Also, since replicas must authorize requests on behalf of the data owner, each member of a replica set must have access to all of the social attestations generated by the data owner. Sharing attestations with replicas does not affect confidentiality since, by definition, replicas already have full read access to the data owner’s objects. Storage servers store objects and their associated access policies in a relational database and local processes access Confidant data through SQL queries.

For the rest of this chapter, we assume that the entirety of a user’s data is managed by a replica set, but Confidant is general enough to accommodate multiple data partitions. For example, users wishing to separate their data into work data and personal data can do so
by creating separate sequence numbers and replica sets for each partition. The number of distinct data sets that a user wants to maintain with Confidant is limited by the amount of state she can afford to host in the cloud and the number of storage servers she trusts to host each partition.

Consistency

We apply an eventual consistency model to data stored within a replica set and rely on epidemic propagation to synchronize storage servers [53, 108]. Because objects and access policies are immutable, ensuring a consistent ordering of updates across replicas is not material. We are only concerned with whether the set of data items stored on behalf of a data owner is consistent with the set of all items the data owner has created.

To ensure eventual consistency, replica-set members periodically exchange missing items. As described in Section 3.3.3, each user has a logical clock, which is maintained by their name server. Since the clock maintained at the user’s name server only increments when a sequence number is assigned to a new item, replicas can reason about any items they might be missing by comparing the clock value at a name server and the sequence numbers of items they possess. A complete set of items would provide a gapless sequence ending in the value maintained by the name server. Missing numbers correspond to items that must be retrieved from other replicas.

Thus, when a storage server comes online, for each replica set of which it is a member, it must query the associated name server for the list of online replica-set members and the current state of the clock. If the storage server has all items with numbers less than or equal to the clock value, it makes itself available to clients by registering with the name servers. Otherwise, the storage server enumerates any missing items, and exchanges its list of missing items with each replica returned by the name server; based on the exchanged lists, items may be transferred in both directions. Once a storage site has synchronized with a replica set, it registers with the data owner’s name server, and repeats the process.
for its next replica set. To accelerate anti-entropy, servers also periodically synchronize with other replica set members.

**Updating and retrieving items**

The messages used to update and retrieve items are listed in Table 3.1.

To add a new object, a client first retrieves a list of online replicas and two new sequence numbers (one for the object and one for the object’s access policy). To store the new items, a client connects to the first server in the list returned by the name server and submits the store-request message described in Table 3.1. The header corresponds to a conjunction, \( \{g_0, g_1, \ldots, g_n\} \), from the the access policy \( ap \); this indicates which group keys, \( gk_0, gk_1, \ldots, gk_n \), are used to protect the message in transit. The client also sends the replica its client certificate, \( cert_C \).

The message payload consists of an object \( obj \), an access policy \( ap \), a random nonce \( rand \), and a signed hash of the object and access policy. The client certificate and signed hash prove to the storage server that the update was generated by a trusted client. If store requests were only protected with group keys, then anyone with access to the proper social attestations could create items on the user’s behalf; social attestations are meant to confer only read access, not write access. Note that the store-request message is vulnerable to a harmless man-in-the-middle attack in which another client swaps in its own certificate and re-encrypts the payload with its own private key.

Once a server has unpacked and verified a store request, it commits the new items to its local database and returns control to the client. The storage server is now responsible for propagating the update to the other replicas listed in the \( replicas \) field of the message payload. We rely on anti-entropy to spread new items to the rest of the replica set in the face of network partitions and server failures.

As mentioned previously, Confidant data is stored locally in a relational database. These databases include two tables for every data owner: one for objects and one for
access policies. Columns in the object table correspond to the fields in an object header as well as additional columns corresponding to the sequence number and each conjunction in the policy expression of the object’s access policy. The policy-expression columns of an object table keep storage servers from needing to access multiple tables when authorizing a retrieval request or sandboxing a script. If either the policy sequence number or expression for an object is null, the object cannot be served. Non-null policy cells for an object are set to the highest sequence number of all access policies referring to the object.

Columns in a policy database correspond to the fields in an access policy. When a server receives a new policy, it inserts the policy into the policy database before using the object owner and sequence-number fields to update the policy information for the object. Policy-update messages are similar to store-request messages, but do not require using group keys since access policies’ content does not need to be protected. However, a signed hash is used to ensure the integrity of a policy-update request.

Retrieve requests include a data owner, sequence number, and conjunction of groups. Because the policy expressions within a descriptor can become stale, clients use the conjunction as a proposed encryption scheme for the object. If the conjunction is listed in the object’s access policy, the storage server responds with the object encrypted with the proposed group keys. If the proposed conjunction is not part of the access policy, the storage server returns a signed copy of the policy so that the client can propose another set of group keys.

Data-processing framework

Our primary motivation for leveraging social relationships to select replicas is to enable scalable, general-purpose data-processing without sacrificing data confidentiality. Prior decentralized OSNs assumed that storage servers were not trusted to view plaintext data, which limited the class of operations that storage servers could perform on OSN data to feature-based searches (e.g., key-word [45, 97, 103] or location-based [86] search). Unfor-
fortunately, many popular and emerging OSN applications such as Twitter’s trending topics and face.com’s face-recognition service require iterating over a large corpus of plaintext data. Services such as these require a general-purpose distributed programming framework like MapReduce [42]. However, unless storage servers are trusted to view plaintext data, such applications can only be implemented by downloading an entire encrypted corpus to a client, where it must be decrypted and analyzed. This approach will not scale for clients executing on resource-limited desktop PCs and mobile devices.

Since a Confidant user’s replicas are trusted, the biggest challenge in designing a scalable data-processing framework is balancing the need to safely sandbox code executed on storage servers and and provide a rich programming API. We do not claim that our solution is perfect, only that it represents a reasonable point in the design space.

The unit of execution for Confidant’s data-processing framework is a script. Scripts must be constrained so that they cannot harm the storage servers on which they execute or access any unauthorized data. To protect the host storage server, scripts are written in Python and execute in a sandboxed Python environment, with pre-built libraries and modules; scripts run under a unique, temporary uid with limited privileges. The standard Unix chroot utility allows storage servers to start scripts in a temporary “jail” directory such that they will not be able to access any other part of the file system.

Storage servers impose CPU, core size and execution time limits on scripts, and run a per-script reference monitor that mediates scripts’ access to the object database. The DBus message system is used as an interprocess communication channel between a script and its reference monitor. Similar to Java RMI, the script obtains a proxy object of the reference monitor from the DBus registry service and uses its interface to query the object database. Scripts submit SQL queries to the reference monitor, which examines and rewrites their queries by adding predicates so that the query only returns data that is authorized by the group credentials submitted with the script. This creates a clean, flexible, and familiar programming environment for developers and relies on existing database mechanisms to
enforce confidentiality. After the script completes its work it creates a file in its temporary directory which is returned to the requesting client as a response. If a script exceeds its resource limits the reference monitor terminates it and the storage server sends back an error message.

3.4 Implementation

We have implemented a Confidant prototype based on the design described in Section 3.3. This section describes our client, name server, and storage server implementations. We also describe three data-processing scripts that run remotely on storage servers.

3.4.1 Client

Our client is implemented as a Firefox web-browser extension that rewrites Facebook web pages and communicates with Confidant name and storage servers. Our Firefox extension transparently integrates Confidant data with a user’s Facebook page.

We derive several benefits from interoperating with Facebook. One, users continue using their existing Facebook accounts, thus leveraging their considerable investment in creating social connections there and learning how to use the many popular features. Two, we take advantage of Facebook as a reliable medium for storing object descriptors and distributing them throughout the social graph.

Facebook remains an untrusted service that should not have access to users’ secret keys or sensitive data. Our browser extension modifies Facebook’s default behavior by listening for browser events such as document loads, form submits, and button clicks. When these events happen, our handling functions are triggered to run prior to Facebook’s original functionality. For example, when a user wants to share a status update or post a wall message, the browser extension intercepts the control flow. Once in control, it contacts the user’s Confidant name server to retrieve sequence numbers for the new object and policy as well as the IP address of the available replicas for storing these items. It
then creates an object and policy with the appropriate descriptor and sends the items to a replica. Finally, the extension substitutes the original content from the Facebook page with the object descriptor to be sent to Facebook.

To retrieve a status update the browser extension scans the loaded page for the object descriptors, parses them, obtains a correct replica IP address from the name server, and downloads the object. Then the extension performs integrity checks, decrypts the data, and replaces the descriptor with the actual content.

For uploading pictures we modified Facebook’s “Simple Uploader” that accepts individual pictures. The uploader proceeds in two steps. First, instead of uploading the actual picture to Facebook our extension sends a dummy image to be stored on Facebook. Next, when a user has to add a description of the picture, the extension creates a new object and descriptor. The object consists of the original picture and the picture’s description. Using the IP address of a replica from the name server, the extension sends the object to Confidant and substitutes the actual description with the object’s descriptor to be sent to Facebook.

Retrieving a picture in Confidant works similarly to a status update retrieval with the exception that the actual image is downloaded locally and linked to the web-page directly from the filesystem.

From our experiments with Firefox we found that asymmetric decryption using pure Javascript code was too slow. Therefore, our Confidant browser extension includes a Python script to perform RSA encryption and decryption. This script provides an interface to be used from the browser by the extension’s Javascript through PyXPCOM.

3.4.2 Name Server

As described in Section 3.3.3, each user runs a lightweight name server that maintains a list of available replicas. We host this server in the Google AppEngine. AppEngine is a highly available and low-cost cloud-computing infrastructure where users can run
applications written in several languages; we used Python. Google does not charge for running applications until they exceed a free-resource quota. We engineered our name server so that its resource utilization should remain comfortably within AppEngine’s free quota.

### 3.4.3 Storage Server

As described in Section 3.3.4, each user also runs a storage server with multiple roles: it maintains the primary copy of the user’s sensitive data; it maintains a replica of the data belonging to a subset of the user’s friends; it arbitrates and serves requests for this data; and it runs data-processing scripts submitted for execution by the user’s friends.

We implemented a storage server prototype in Python using the Django web framework. We also use MySQL to store data, Apache2 for serving requests, and JSON as a lightweight data-interchange protocol.

### 3.4.4 Data-Processing Scripts

As described in Section 3.3.4, Confidant allows data-processing scripts to execute on the distributed collection of storage servers. We have implemented three representative scripts and describe them below:

**Keyword-search Script:** The *keyword-search script* is a simple script that searches through a user’s friends’ accessible status updates and returns those that satisfy criteria such as falling within a date range or having specific keywords. This is the simplest script we implemented. It creates a SQL statement with the required conditions, receives the result from the reference monitor, encrypts the result, and sends back the encrypted data to the client.

**Trending-Topics Script:** The *trending-topics script* calculates the most frequently used words within a user’s friends’ status updates. This script prepares a dictionary object 

\[
[\text{"word1" : "count", "word2" : "count", ...}]
\]

using the available status updates and wall
messages on each storage server. We eliminate common words that should not be interpreted as trends such as “I”, “me”, “am”, and “you”. A client downloads these preprocessed objects from their friends’ replicas and merges them into a single list. The client then sorts the list by value to produce the most trending keywords.

**Face-Detection Script:** We implemented a *face-detection script* that returns friends’ pictures with a predefined number of faces in each of the returned pictures. For example, with $N = 1$ the script will return only portrait pictures, however with $N \geq 5$ it will return group photos such as pictures from a party or a conference. We implemented the face detection algorithm using the Python wrapper for OpenCV and we assume that the sandboxed environment has the wrapper along with the OpenCV library itself.

### 3.4.5 Untrusted Storage Server

In order to compare Confidant with the alternative approach to decentralized OSNs where storage servers are considered untrusted, we implemented a simple server that stores encrypted data and returns it to the requester with minimal security checks. We use this server to help evaluate the performance of Confidant in data-processing tasks such as finding trending topics and face detection.

### 3.5 Evaluation

In evaluating Confidant, we sought answers to the following questions:

- How does the performance of our data-processing framework compare to approaches that rely on untrusted storage servers?

- What is the latency of creating and retrieving photo and text objects?

- How many trusted friends will Confidant users need to ensure that at least one replica is always available?
3.5.1 Data-Processing Performance

In this section, we compare the performance of Confidant’s data-processing framework to the performance of approaches that rely on untrusted storage servers. We measured the end-to-end latencies of the face-detection, keyword-search, and trending-topics scripts described in Section 3.4.4 and compared them to the latencies of fetching data encrypted under AES from an untrusted server and processing it locally. We refer to the latter approach as Encrypted.

All experiments used EC2 virtual machines as trusted storage servers. Using Amazon’s EC2 allowed us to test the scalability of Confidant. We used a single untrusted server running on EC2 for all Encrypted experiments. Varying the number of untrusted servers had almost no effect on end-to-end latency of our Encrypted experiments because the decryption and processing time spent at the client dominated our measurements.

For a client, we used a consumer laptop (Intel Core 2 Duo processor, 1.86GHz, 6 MB L2, and 2 GB of memory) on Duke University’s network. The client ran a multithreaded python program that queried storage servers and processed replies as required. A separate thread was spawned for each request.

To populate the remote servers for our trending-topics and keyword-search experiments, we used status updates collected from Facebook by researchers at UCSB [89]. We used images from publicly available datasets from computer vision research groups at CMU, MIT and Cal Tech for evaluating the face-detection script’s performance. Our image corpus contained 70 images altogether. We tuned the parameters of the face-detection algorithm so that it was reasonably accurate and required about 500 ms to process a 30Kb picture. It should be noted that we did not evaluate the accuracy of our face detection algorithm and, thus, we do not report how many of the returned pictures were false-positives. The average size of a picture used for the evaluation is 30 KB, with sizes ranging from 8Kb to 200Kb.
Confidant achieves compute scalability by off-loading computational tasks to trusted remote servers, while the Encrypted approach must first download data to a client and decrypt it before processing it locally. To isolate the effect of computational scalability on end-to-end latency, during our experiments the Confidant clients and Encrypted clients only differed in how much processing they performed rather than the volume of data they received. For the keyword-search experiments, we partitioned 1,000 status updates across the remote storage servers. For the trending-topics experiments, remote Confidant scripts collectively returned dictionaries based on 1,000 status updates to the client. For the face-detection experiments, we assigned each remote server 10 photos. Finally, for our experiments, we conservatively assumed that each of the client’s friends had only one replica available.

Figure 3.2 shows the average over 20 trials of each script’s latency under Confidant divided by the latency to perform the same task under the Encrypted scheme. Please note the logarithmic scale of the y-axis. Standard deviations for all averages were less than 2%. For 100 friends, Confidant was between 3 and 30 times faster (trending topics and
face detection, respectively). While all three scripts performed better under Confidant, the face-detection script benefited the most from parallel remote execution due to the computational complexity of detecting faces in images. Trending topics benefits the least due to the computational complexity of merging dictionaries from many sources that must be performed at the client. Nonetheless, these results demonstrate the performance and scalability benefits of Confidant’s remote data-processing framework.

3.5.2 Access performance

To characterize the overhead of Confidant we measured the time for our client to complete four common OSN tasks: uploading a 120kb photo, downloading a 120kb photo, uploading a text status update, and downloading a text status update. To perform these experiments, we used a virtual machine hosted on Amazon’s EC2 as a storage server and our consumer laptop client with Firefox 3.0 and our Confidant browser extension. We
compared Confidant’s performance to Facebook’s using the same laptop and Firefox 3.0. Times were measured from the moment the DOM loaded to the time that the item was rendered in the browser (items were rendered for both uploads and downloads).

The results in Figure 3.3 and Figure 3.4 show the sources of delay for Confidant. Relative to Facebook, Confidant uploaded a status update 2.6 times more slowly, downloaded a status update 1.6 times more slowly, uploaded a photo 1.6 times more slowly, and downloaded a photo 4.3 times more slowly. The main source of extra delay for uploading and downloading status updates was the approximately 400ms spent contacting a name server, which is due to Google’s aggressive idling of AppEngine processes. For photos, the primary source of latency was processing the image, which included decrypting the image and verifying its hash in Javascript. For both figures, latency labeled “Other” included server-side processing and offloaded RSA checks.

We find these numbers encouraging. The latency of contacting a name server might be mitigated via caching (in the download cases), and more cryptographic functionality can be moved out of Javascript. It is important to note that while Facebook currently provides better performance, Confidant’s integration with Facebook means that it can only add overhead to these baseline numbers. Furthermore, unlike Confidant, Facebook does not encrypt any of its messages, leaving them exposed to the network.

3.5.3 Availability

Confidant relies on replicated storage servers to ensure that clients can successfully submit and retrieve items. The availability of a user’s replica set is a function of two properties: the number of trusted friends a user has (i.e., the size of the replica set), and the availability of individual replica-set members. To explore the impact of these properties on the rates at which clients can successfully read and write, we simulated Confidant using two types of traces. To characterize how many friends users have and how they interact with their friends, we used a trace of the Facebook graph and wall messages collected at MPI [111].
This trace captures links and interactions between users and consists of approximately 60,000 users, connected by over 0.8 million links, with an average node degree of 25.6.

To characterize storage-server availability, we used the well-known Gnutella trace [92] and Microsoft-workstation trace [27]. The Microsoft-workstation trace is much more forgiving and machines have a much higher average online time than hosts in the Gnutella trace. It should be noted that there were many IP addresses listed in the Gnutella trace that were never online, and we removed all such nodes. This left us with approximately 15,000 hosts over 60 hours. From the Microsoft trace, we used roughly 52,000 hosts’ records over 35 days.

Since the MPI trace contains more users than hosts in the Gnutella or Microsoft-workstation traces, we pruned the MPI trace to fit each availability trace. First, we sorted users in the MPI trace by level of interactivity (i.e., the number of wall posts and comments written and received), and assigned the users who were the most active a host from
the availability trace. Connections to users who were not among the most active were cut. The resulting subgraph was denser than the original, exhibiting an average connection degree of 44.8 and 30.2, for the top 15,000 (Gnutella) and top 52,000 users (Microsoft), respectively. 86% of the top 15,000 users have 10 friends or more, while 61% of the top 52,000 users have 10 friends or more.

We included the most-active Facebook users from the MPI traces because they generated the most write events for our simulation. Of course, there is a correlation between activity and connectedness, as our increased average node degree demonstrates. This correlation biases in our favor since the resulting graph includes a larger fraction of highly-connected users. However, even with the higher average node degree of 44.8, this is still substantially lower than the average of 130 reported by Facebook [50].

The relationship between a user’s activity on Facebook and the availability of her storage server is unknown. As a result, we explored three possible ways of assigning users in the MPI trace to hosts in our availability traces: randomly, by connection rank (a more connected user was assigned higher availability), and by interaction rank (a more active
user was assigned higher availability). Under each of these mappings, nodes had a maximum replication factor of 10. If a user had 10 or more friends, its replicas were chosen to be the 10 users it interacted with the most. Nodes with fewer than 10 connections used all of their connections as replicas.

We assumed that reads and writes were generated by an always-available client such as a mobile device. We simulated writes using interactions from the MPI trace. For a write event, if at least one member of a client’s replica set was available, the write was considered a success and was added to a read queue for each of its friends. Writes generated when no replica-set members were available counted as a failure and were not added to friends’ read queues.

Reads were not captured in the MPI trace so we generated them synthetically. Clients chose a random time between 30 minutes and 60 minutes, and after that time passed attempted to retrieve the objects in their read queue. This read rate is consistent with the reported behavior of Facebook users [50]. Read failures occurred if a client tried to read an object while no member of the replica set was available, or none of the online replica

![Graph showing simulated read-success rates (Gnutella).](image-url)

**Figure 3.6:** Simulated read-success rates (Gnutella).
servers had the object at that time. Otherwise, the read was considered a success. After clearing its read queue, clients chose another random period before they read again.

Figure 3.5 and Figure 3.7 show the read and write success rates for simulations using the Microsoft availability trace, while Figure 3.6 and Figure 3.8 show the read and write success rates using the Gnutella trace. As expected, nodes with more replicas had higher read and write success rates, regardless of how Facebook users were assigned host availability records. The overall higher read success rates across experiments are partially an artifact of failed write attempts being suppressed in the read results; if a node failed to perform a write, no other nodes attempted to read it.

One of the most striking features of our results is how much worse nodes with fewer replicas fared when low connectivity was correlated with low availability. This is because nodes with low connectivity were penalized twice. Not only did these users have access to fewer replicas, but their own server was assigned low availability as well. This combination of factors led to significantly lower success rates than for nodes with low connectivity under the other schemes or nodes with more replicas.
Unsurprisingly, read and write success rates under the Gnutella trace are much worse than under the Microsoft-workstation trace. The Gnutella trace likely provides a close to worst-case scenario for host availability since it captures client-process uptime rather than machine uptime. On the other hand, results from the Microsoft-workstation trace are likely close to a best-case scenario since the trace captures machine uptime in a corporate environment. Reality likely lies somewhere in between. It is important to note that even under the less-forgiving availability of the Gnutella trace, users with ten replicas fared well. The worst case for ten replicas was under a connection-based mapping, which generated a write-success rate of 99.6%. Under the Microsoft-workstation trace, three or four replicas provided a write-success rate of 99.5% using the connection-based mapping. Users with more than four replicas under the Microsoft trace had perfect read and write rates for all mappings. All of these results bode well for Confidant. Facebook users have an average of 130 friends, and we suspect that it would be straightforward for most users to identify 5-10 trusted friends who are willing to host their data.
3.6 Conclusion

We have presented Confidant, a decentralized OSN designed to support scalable data-processing. The key insight behind Confidant is that friends who already have access to a user’s data may be trusted to serve it as well. By replicating users’ data on their friends’ storage servers, Confidant allows data-processing code to be shipped to trusted servers. The key challenge in realizing this vision is managing access-control policies under weakly-consistent replication. Confidant addresses this challenge by eliminating write conflicts and through a simple mechanism for updating the bindings between access policies and replicated data. Finally, we evaluated Confidant using trace-driven simulation and experiments with a prototype implementation. Our results show that Confidant provides nearly perfect read and write success rates for users and good performance for data-processing tasks.
Enforcing third-party OSN applications to respect service provider’s policies

In this chapter we present Multi-user Taint Tracker (MUTT) [95] – “platform as a service” for third-party OSN cloud applications, which is designed to ensure that third-party applications adhere to access policies defined by the OSN service providers.

4.1 Introduction

Online social networks (OSNs) such as Facebook and Twitter archive and disseminate messages and media for hundreds of millions of users. These large-scale, multi-user services augment their core functionality through extension frameworks for third-party developers. *Applications* written within these frameworks consist of browser-side and server-side code that interact with a service through remote APIs. Applications are popular because they allow users to personalize the functionality and presentation of their OSN, while simplifying identity management and messaging for developers.

Ideally, application frameworks would offer these benefits without putting sensitive data at risk. Services such as Facebook actively sanitize applications’ browser-side code,
but because today’s frameworks have little control over how server-side code handles users’ data many apps have been caught leaking private data [14, 6, 12, 77, 54]. For example, Twitter recently suspended many popular applications for privacy violations, commenting that, “on an average day we turn off more than one hundred services that violate our API rules of the road. This keeps the ecosystem fair for everyone” [14].

Violations can be categorized in one of two ways: (1) unauthorized external sharing with entities such as third-party advertising networks [14, 6, 12], and (2) unauthorized internal sharing with other users of the service [77, 54]. To addresses the problem of unauthorized external sharing, systems such as xBook [102] provide a trusted runtime capable of limiting the communication of application components to a set of whitelisted administrative domains. However, controlling information flows at the granularity of an administrative domain is too coarse to prevent unauthorized internal sharing.

Unauthorized internal sharing occurs when an application obtains read and write permissions from many users and misuses them to circumvent a service’s internal access controls. Services provide users with settings for controlling how their data may be shared with others, what data an application may access, and which users may view messages posted by an application on their behalf, but these measures cannot always prevent leaks. For example, the “Top Friends” Facebook application allowed any user who installed the app to view the personal information of any other user who installed the app [77].

In this paper we present a set of server-side extensions for “Platform as a Service” (PaaS) infrastructures called MUTT that tracks and controls the flow of user data through hosted applications. MUTT integrates fine-grained information-flow tracking (i.e., taint tracking) into a PaaS’s runtime environment and storage to ensure that users’ data does not leak. The key question for MUTT is what responsibilities should be handled by the trusted PaaS and what should be left to the application developer’s discretion.

To answer this question, we studied the terms of service for several popular OSNs, including Facebook, Google+, and Twitter. Each of the services in our study require
applications to limit sharing to only “authorized users” and to keep cached access con-
trol policies “reasonably” fresh. As a result, MUTT assumes responsibility for the fresh-
ness of cached data and mediates interactions between applications and services through service-specific drivers. MUTT proxies all of a hosted application’s network communication through these drivers, which assign labels to an application’s data, processes, and communication channels. The MUTT runtime is responsible for propagating labels and enforcing access control policies based on these labels.

A study of existing applications and service APIs showed that most are compatible with MUTT. A review of 170 popular Facebook applications revealed that only dating applications require information flows to unauthorized users as part of their core functionality. Furthermore, only Facebook’s “customized” access control policies require additional API support.

To summarize, this paper makes the following contributions:

- We designed MUTT— a set of extension to PaaS infrastructure capable of preventing unauthorized internal sharing by OSN applications.
- We reviewed and classified 170 popular Facebook applications based on their inter-user data sharing.
- We studied the platform policies for Facebook, Flickr, Google, and Twitter, and analyzed how these policies can be violated by third-party applications.
- We implemented a MUTT prototype and showed that it imposes reasonable overhead on third-party applications.

The rest of this paper is organized as follows: Section 4.2 provides background information on the technologies underlying MUTT, Section 4.3 describes MUTT’s trust and threat model, Section 4.5 presents our design goals for MUTT, Section 4.6 presents MUTT’s
design, Section 4.7 describes our prototype implementation, Section 4.8 presents our evaluation results, and Section 4.9 provides our conclusions.

4.2 Background

In this section we provide background information on some of the enabling technologies referenced throughout the rest of the chapter.

OSN applications

Many online social networking (OSN) services allow third-party developers to build service extensions called applications that benefit from accessing a user’s data (e.g., her social graph and personal information).

Since these applications are the focus of our work, it is important to provide a concrete definition. An OSN application is third-party code that utilizes a user’s social graph as a key feature of its functionality. Without access to a user’s social graph, the app would be impossible to use. For example, many OSN games are popular because their OSN friends use them, send them reminders, and “ask for help.” Other apps use pictures, videos, and status updates to create collages, ask questions, or collect opinions.

Since Facebook is the world’s most popular OSN service and has the most popular application framework, we will describe its framework in greater detail. Frameworks for other services are similar. The browser-side code of each Facebook application is sanitized by Facebook to ensure that it is properly sandboxed. This sanitized code may communicate with Facebook and with a pre-specified home server running on remote infrastructure. Server-side code not developed by Facebook runs outside of Facebook’s datacenters, usually inside their own [9] or on public cloud infrastructure such as EC2 or AppEngine. In fact, Facebook recently announced a partnership with cloud service Heroku [3] which makes it easier to run the third-party applications with features like automatic deployment and instant scaling. This is encouraging since MUTT could also be integrated into the
Heroku PaaS.

**OAuth**

Each application must obtain a user’s consent during the installation process in order to access her data. An application’s server-side code can also request access to a user’s data in “offline mode” through OAuth.

OAuth is an open standard for federated authorization used by most web services. OAuth allows a user to share the data stored by a service with another entity (e.g., an application) without needing to share coarse credentials such as a username and password. Under OAuth a service creates a capability called an OAuth *token* for the application that can later be presented to access a user’s data. This token typically contains a list of resources that can be accessed, an expiration date, and other relevant information.

The procedure for obtaining an OAuth token is as follows: when an application requests access to a user’s data it redirects the user running the application to the service in control of the data, passing along a list of data types that it would like permission to access (e.g., photos, messages, videos) as a parameter. The user directly authenticates herself to the service and approves the requested permissions. Then the service redirects the client back to the application, passing along a secret string that the application uses to obtain its token directly from the service.

After presenting the string to the service and retrieving a token, the application can store the token locally and encode it within a cookie that is stored on the client for use in subsequent requests. The token-encoding included in the cookie acts as a *user ID*. An application can use the user ID included in client requests to retrieve the appropriate OAuth token from storage. The token can then be used by the application when making API calls to the OSN.
**Taint Tracking**

Dynamic information-flow analysis (i.e., taint tracking) is a well-known technique for monitoring data dependencies as a system executes. Imagine an application that fetches a user’s pictures, processes them, and displays the result (e.g., a collage). Taint tracking can be used to detect when the application discloses the user’s pictures and collage to unauthorized users.

In general, taint tracking works as follows. First, all sensitive data must be labeled using a special *taint label* that carries information about its source. Second, taint labels must be propagated to other objects whose values are derived from sensitive data. Third, if two or more tainted objects are used to create a new object (e.g., string concatenation or arithmetic operations) the resulting object must be assigned with a newly created taint label according to a special *merging policy*. Finally, the system must check the eligibility of exiting tainted objects outside of the application according to *taint export policies*.

**Platform-as-a-Service**

Platform-as-a-Service (PaaS) is a computing model that provides a homogeneous environment or platform for running multiple applications within possibly heterogeneous cloud infrastructure. Usually PaaS environments offer automated scaling in response to the dynamic resource requirements of an application.

The most popular PaaS is Google App Engine (GAE). For our MUTT prototype, we modified AppScale – an open source implementation of GAE developed by researchers from UCSB [36].

4.3 Trust and threat model

Like many other taint-tracking systems, MUTT cannot track data unless it is properly labeled. Hence, the confidentiality guarantees of the platform depend on (1) preventing
labeled data from flowing into unauthorized sinks, (2) utilizing information from existing service APIs to appropriately label data, and (3) propagating labels throughout the PaaS language runtime and built-in libraries. As a result, we assume that the entire platform (language runtime, libraries, and modules) is trusted.

Since we explicitly chose to design MUTT as an extension to an existing platform (i.e., AppScale [36]), we inherit some security properties from it as well as the underlying software. If the Python runtime, application server, or a standard library is compromised, users’ data can leak. Another attack is to compromise the platform’s built-in C module to remove taint labels from objects containing sensitive data.

It is difficult to prevent leaks through covert channels like timing attacks and a program’s control flow without over tainting. However, we support a special taint flag that forbids using an object with tainted values within a conditional statement. This restriction eliminates any possible leaks through implicit flows, but may not be appropriate for all applications. We discuss this tradeoff further in Section 4.6.4.4.

Considering these challenges, we primarily target MUTT at preventing overzealous developers from overshar ing sensitive data or handling it inappropriately (e.g., forgetting to delete it after the user uninstalls the application). In fact, we believe that most of the cases in which third-party apps violate service platform policies are inadvertent.

Finally, a user and service have to trust the PaaS hosting third-party applications. If a developer claims that her app is hosted by MUTT, users and services can use techniques such as logical attestation [93] to verify those claims. Services such as Facebook, Google, or Twitter should take a leading role in providing their users with information about third-party applications, and it would be easy to include meta-data about an application’s hosting platform during the OAuth authorization process. Hence, users could decide to grant permissions based not only on functionality and design of an application but also on the credibility of the cloud platform on which the application is hosted.
Table 4.1: Facebook applications

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
<th>DAU</th>
<th>MAU</th>
<th>Developer</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>FarmVille</td>
<td>Game</td>
<td>10M</td>
<td>30M</td>
<td>Zynga</td>
<td>USA</td>
</tr>
<tr>
<td>Texas HoldEm Poker</td>
<td>Game</td>
<td>7M</td>
<td>35M</td>
<td>Zynga</td>
<td>USA</td>
</tr>
<tr>
<td>Daily Horoscope</td>
<td>Personal</td>
<td>6M</td>
<td>14M</td>
<td>Soft Reklam</td>
<td>Turkey</td>
</tr>
<tr>
<td>Badoo</td>
<td>Dating/Matching</td>
<td>2M</td>
<td>35M</td>
<td>Badoo</td>
<td>Cyprus</td>
</tr>
<tr>
<td>Pet Society</td>
<td>Game</td>
<td>1.5M</td>
<td>9M</td>
<td>Electronic Arts</td>
<td>USA</td>
</tr>
<tr>
<td>Mafia Wars</td>
<td>Game</td>
<td>1.4M</td>
<td>7.6M</td>
<td>Zynga</td>
<td>USA</td>
</tr>
<tr>
<td>Phrases</td>
<td>Personal</td>
<td>1.3M</td>
<td>16M</td>
<td>Takeoff Monkey</td>
<td>N/A</td>
</tr>
<tr>
<td>Zoosk</td>
<td>Dating/Matching</td>
<td>1.3M</td>
<td>22.8M</td>
<td>Zoosk</td>
<td>USA</td>
</tr>
<tr>
<td>AreYouInterested?</td>
<td>Dating/Matching</td>
<td>0.8M</td>
<td>12M</td>
<td>SNAP Interactive</td>
<td>USA</td>
</tr>
<tr>
<td>Monster Galaxy</td>
<td>Game</td>
<td>0.6M</td>
<td>17M</td>
<td>Gaia Online</td>
<td>USA</td>
</tr>
<tr>
<td>Cupid</td>
<td>Dating/Matching</td>
<td>0.2M</td>
<td>8M</td>
<td>Cupid</td>
<td>USA</td>
</tr>
<tr>
<td>Friend.ly</td>
<td>Dating/Matching</td>
<td>0.2M</td>
<td>7M</td>
<td>Friend.ly</td>
<td>USA</td>
</tr>
<tr>
<td>FriendMatrix</td>
<td>Personal</td>
<td>0.2M</td>
<td>4M</td>
<td>Matrix Studios</td>
<td>N/A</td>
</tr>
<tr>
<td>My Top Friends</td>
<td>Personal</td>
<td>5K</td>
<td>140K</td>
<td>MiniMax</td>
<td>China</td>
</tr>
<tr>
<td>FriendCollage</td>
<td>Personal</td>
<td>N/A</td>
<td>303K</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Museum of Me</td>
<td>Personal</td>
<td>N/A</td>
<td>200K</td>
<td>Intel</td>
<td>USA</td>
</tr>
</tbody>
</table>

Table 4.2: Categories of Facebook applications

<table>
<thead>
<tr>
<th>Category</th>
<th>Required access to</th>
<th>Required data sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Public data</td>
<td>Private data</td>
</tr>
<tr>
<td>Games</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dating/Matching</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Personal</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

4.4 Status Quo

In this section we survey the status quo of (1) OSN apps’ access to user’s personal information by reviewing and analyzing existing popular Facebook applications, and (2) services’ expectations of how applications should handle users’ data. We will use our findings to motivate MUTT’s design.

4.4.1 Classification of applications

4.4.1.1 Applications

To understand the nature of third-party apps we examined more than 200 popular applications from an open application directory [4]. Despite the fact that we picked only Face-
book applications, we believe these findings are widely applicable to other frameworks since the Facebook Platform is the most popular platform for OSN apps and provides the most flexible and complete API for developers. Thus, we picked these applications based both on their popularity (i.e., *daily active users (DAU)* and *monthly active users (MAU)*) and diversity, aiming to have a broad sample of existing OSN applications.

We excluded three categories of applications from our study. First, we did not consider applications developed by Facebook (e.g., FBML, Discussions) since they remain under the service’s control. Second, we did not study Facebook applications that do not run in the cloud such as mobile applications (e.g., HTC Sense, Android, BlackBerry fb apps). Finally, we did not study apps that only connect Facebook with another well-established service like Bing, Windows Live Messenger, Yahoo!, or YouTube. The reasons for these exclusions are that (1) these apps are not OSN applications according to our definition from Section 4.2, and (2) these applications run in home datacenters and infrastructures rather than in an open PaaS environment.

After excluding the above-mentioned categories of Facebook applications, we examined 170 popular third-party apps and found that all can be placed into one of three categories: *gaming*, *dating*, and *personal* apps. Table 4.1 shows a sample from the study, with five applications per category. The table is sorted by DAU and contains additional information about each application (e.g., their popularity and information about the developers). As can be seen from the table, games and dating are the most popular categories. However, we intentionally included another class of applications – personal apps. These apps vary in terms of functionality and usually generate aggregate views about a user's social graph, photos, or friends. We did not identify a dominant app in this category, but we found several moderately popular ones.
4.4.1.2 Data sharing analysis

To infer the risks that each category of applications poses, we analyzed each app to understand what type of data it can access and whether this data could be leaked to other users. We wanted to learn the following:

- Do the apps require access to public data such as name, user ID, profile picture, gender and/or private data such as pictures, location, birthday or religious/political views?

- Do the apps require the ability to share data with other application users within and/or outside of the user’s social graph?

Of course, some apps may request more data than they need for purposes unrelated to the primary app functionality. However, we believe that most of these incidents are due to overzealousness rather than malice, and in fact Facebook actively tries to prevent such behavior [52].

It is important to acknowledge that we never had access to the source code of the applications in our study. Thus, our classifications are based solely on our analysis of the Facebook Platform API, experience with building simple applications, and observations of app behavior.

The result of our analysis is summarized in Table 4.2. As can be seen from this table, all applications have access to user’s public data (e.g., name, user ID, profile picture, gender, hometown, networks, and friends list), and both dating and personal apps require access to some of a user’s private data (e.g., pictures, bio, posts, or favorite places).

Each reviewed category also has different information-flow requirements. To make an appropriate match, dating apps need to share a user’s data with people outside of her social graph. Gaming applications heavily use an owner’s social graph to engage his or her friends in the game. Finally, personal applications do not necessarily need to use the
social graph to \textit{propagate} information. Data sharing in this scenario is generally controlled by the user of the app. For example, the FriendMatrix application just produces a collage using accessible pictures of a user’s friends and does not push anything outside of the user’s profile. In this case the owner can restrict access to this collage by adjusting the appropriate privacy settings on the album containing the picture.

We found that some third-party applications cannot be strictly associated with a particular category. For example, calendars are personal applications, but require access to the user’s social graph as a part of their basic functionality.

\textbf{4.4.1.3 Application vs. service data}

While studying applications and the Platform API we found that third-party apps also treat a user’s social graph differently.

For example, personal applications in general do not cache a user’s social graph since the apps from this category are usually stateless. Instead, they gather information at the time they run to generate a result. When it runs again it re-generates the result based on the most current state of the social graph. Thus, these applications always use a fresh social graph and other data.

On the other hand, many reviewed applications maintain their own social graphs as a part of their functionality. In Farmville [7] friends can be virtual neighbors, send requests and gifts to each other, including leaving notifications on each others’ profiles. Badoo [1] also maintains its own social graph using the initial friends list. Users of this application can answer trivial questions about their friends to receive bonus points and post these answers on their friends’ profile pages. Moreover, users can rate their friends’ pictures as a part of app’s functionality. We found that in both cases, even if user A “unfriends” user B, user B would still be able to post notifications on user A’s profile page by relying on the application’s permissions. User A can only block the notifications by blocking the application entirely.
This varying behavior raises the question of how MUTT should treat third-party applications. If an app is a service extension then it must keep an up-to-date view of the user’s data, particularly her social graph. Otherwise, we could treat each third-party application as a separate OSN, and allow it to maintain its own social graph. However, we believe that this approach would be counter-intuitive for a user since it requires managing multiple social graphs within a single framework. We understand that some applications may need to maintain their own application-specific groups and friends lists. Thus, the problem of user-friendly and privacy-preserving data sharing within such groups in the presence of a dynamically changing social graph requires more attention.

4.4.2 Service policies

We also wanted to understand service’s expectations of how third-party apps’ should handle users’ data. We reviewed the Terms of Services for Facebook, Flickr, Google, and Twitter and identified three common requirements for third-party apps. From now on we will call these requirements a service’s default policies.

First, service providers require that any data fetched by a third-party application be shared only with “authorized users.” For example, if a picture on Flickr has a tag “public” it can be shared with everyone. In contrast, if the owner of the picture specified an access control list (ACL), the app should respect this restriction. As another example, Facebook states that no data (including aggregated, anonymous and derivative) can be shared with any third-party company, even with a user’s consent.

Second, services have special requirements for caching data. Platforms require that caches be updated within a “reasonable” amount of time. In addition, Facebook requires apps to delete all of a user’s information upon the user’s request or if Facebook disables the app.

Third, apps are required to keep items’ ACLs fresh. For example, when an app fetches a user’s picture it might have a specific ACL \( acl_1 \). However, after some time the user might
change the picture’s ACL to a more restrictive ACL $acl_2$, such that $acl_2 \subseteq acl_1$. Thus, if the app still uses $acl_1$ it would violate the user’s privacy. In such cases, the third-party app is bound by the caching policies and must refresh a data item’s ACL within a reasonable amount of time.

For further references we will summarize and define these default policies as follows.

**Definition 1. Default policies:**

#1: No unauthorized disclosure of data.

#2: Reasonable caching policies.

#3: Fresh cached and remote ACLs.

4.5 Design Challenges and Goals

Ideally, MUTT could support all possible applications and enable arbitrary policies supplied by users and OSN providers. However, as a first step toward this aim and primary goal of this work, we seek to design a system that can enforce the default policies outlined in Section 4.4.2 for the applications we reviewed in Section 4.4.1.

4.5.1 Design Challenges and Limitations

Despite our modest design requirements, the impreciseness of the service requirements and a limited Platform API create challenges. Consider, for example, default policy #1, which states that data can be shared only with the authorized users.

We have examined the APIs of Google+, Twitter, Facebook, and Russian site Vkontakte [16] to understand if they can be used to ensure the authorized exposure of a data item. We found that within all of these platforms, with the exception of Facebook, (1) every user-supplied data item such as a wall post or personal note has a unique global ID, and (2) an application can retrieve an ACL for this data item through its ID. Hence, using these APIs MUTT can verify whether sharing an item is authorized.
However, we found that despite of this functionality it is still hard to generalize and enforce default policy #1 for at least two reasons.

First, Facebook’s API does not give full access to a data item’s ACL, and thus it is not always possible to ensure authorized sharing. For example, imagine a user giving a specific application a permission to access her pictures. Then, if this user has an album with the access control list that includes only a subset of her friends, the app can still access it and potentially share pictures from the album with the user’s entire social graph, including friends who were not supposed to see the album.

This leak cannot be prevented as long as the Facebook Platform API does not expose the complex ACLs of individual data items to third-party apps. A complex ACLs is generally created manually, as opposed to standard ACLs such as “friends”, “friends of friends” or user’s networks. Moreover, it is evident that this information is sensitive itself, and it is unclear if Facebook will give access to such ACLs in the future.

Second, and more important, depending on the application, the definition of “authorized disclosure” can be ambiguous. For example, consider any dating or matching app. Typically, a dating service has access to a user’s personal data such as her name, location, and pictures. Using this data, the app tries to identify matches outside of the user’s social graph. Thus, this information can potentially flow anywhere within the entire social network, including malicious, data-gathering profiles since most parameters (e.g., gender, age, location and picture) can be manipulated to ensure a high rate of matching. For such cases, it is not clear how can one should define “authorized disclosure,” even assuming a non-malicious application.

Considering these issues, we will refine the default policy #1 from Section 4.4.2 to be more precise: “sensitive data can only be shared only within the owner’s social graph.”

Note that this requirement would disallow MUTT from hosting dating applications since these apps often share sensitive data with the users outside of owner’s social graph.

Similar to default policy #1, default policy #3, which requires cached ACLs to be fresh,
depends on the Platform API for accessing items’ permission lists. Without an OSN’s co-
operation enforcing this policy could significantly degrade app performance, since ensur-
ing the consistency of local and remote ACLs we would require expensive extra API calls
for nearly every operation. As a result, we leave data freshness as a tunable parameter
whose value can exposed to users and services through the attestation process.

With this in mind, our refined default policies are as follows:

**Definition 2. Default policies (refined):**

#1: User’s tagged data cannot flow outside of her social graph.

#2: Respect items’ caching policies.

#3: Opportunistically verify the consistency of item’s ACL.

4.5.2 Design Goals

After discussing the challenges and limitations of building a PaaS for third-party appli-
cations capable of enforcing service access control policies, we now state our goals for
MUTT’s design.

- Our system should be as transparent as possible for users, developers, and OSN
  providers. It should support offline requests and any existing functionality without
  assuming changes to platform APIs.

- At a minimum, our system should enforce the default policies described by the
  refined default policies.

- Our system should be easily extensible to support arbitrary (e.g., user-supplied or
  service provider’s) policies.

- Our system should have reasonable performance overhead.
4.6 Design

4.6.1 Strawman

We begin by describing a simple design for building a PaaS capable of protecting the privacy and security of users’ data. To access a user’s OSN data a third-party application must first obtain an OAuth token [10] from the OSN. Each token is a unique randomized string that functions as a capability and allows the application to access a user’s data.

After presenting the string to the OSN and retrieving a token, the application stores the token locally and encodes it within a cookie that is stored on the client for use in subsequent requests. The token encoding included in the cookie is a user ID. An application can use the user ID included in a client request to retrieve a corresponding token. Such a token can then be used by the application when making API calls to the OSN for accessing a user’s data.

Our Strawman augments the PaaS hosting a third-party application by interposing on
incoming client requests and interactions with a local database. Interposing on incoming
requests allows the PaaS to assign the user ID associated with a request to the application
instance that handles it, just as UNIX and other operating systems assign UIDs to processes
forked on behalf of a particular user.

The PaaS can then annotate any data items added to the application’s database with the
user ID of the instance performing the insertion and enforce user-ID-based access control
as other instances access the database. This approach is analogous to the way that a UNIX
and other operating systems regulate accesses to a file system, although it is stricter since
the PaaS ensures that a data item can be accessed by only processes with a matching user
ID. To further ensure that the instance dispatched to handle a user’s request will not access
another user’s tokens or data, the PaaS can disallow IPC between instances.

The simplicity of the Strawman is appealing, but it is also limited in a number of ways.
First, because it does not regulate network accesses, the PaaS cannot prevent an application
from leaking sensitive data to other servers. For example, an application instance could
use urllib to send data to arbitrary external sources via HTTP. Second, some “mashup”
applications may want to process data from multiple OSNs, but this is difficult when user
IDs are bound to a single OAuth token (and, hence, a single OSN). Finally, since this
design only allows data to be exported back to its owner, the Strawman can only support a
handful of personal apps.

4.6.2 MUTT Architecture Overview

To address the shortcomings of the Strawman we designed MUTT– a Multi-User Taint
Tracking system for third-party OSN applications. We extended the Strawman by adding
dynamic taint analysis to the language runtime of the PaaS. Taint tracking assigns a label to
each data item that encodes its origin. The runtime assigns labels to items at well-known
taint sources and propagates them as an application executes. Application code cannot
directly modify the labels of any data it manipulates. Whenever application code updates
an item’s value, the runtime also updates the item’s label to reflect the sources of its new values. For example, when an application creates a new string by concatenating two other strings, the new string’s label is set to the union of the others.

For a PaaS, taint sources are the OSNs from which an application fetches a user’s data. Thus, the runtime must interpose on and interpret any OSN API calls made by an application so that the OSN-of-origin and type information can be appropriately encoded in any retrieved item’s taint label before the item is returned to application code. MUTT ensures that data can be properly labeled by introducing a suite of trusted Service drivers capable of (1) retrieving data on behalf of an application instance, (2) labeling data before it is returned to the instance, and (3) mediating requests to modify remote OSN data.

Implementing separate drivers for multiple OSNs supported is reasonable since most OSNs expose well-defined, RESTful APIs to applications. Note that a Service driver needs to handle the initial token negotiation as well as subsequent API calls for accessing and updating a user’s data. Drivers label any tokens retrieved from an OSN, and an application is responsible for passing the appropriate token back to a driver during later interactions with the OSN.

Figure 4.1 shows a high-level overview of MUTT. Application code runs in an isolated App instance in a modified taint-tracking language runtime. A Client interacts with the app instance via the AppServer, which redirects requests to the appropriate instance and labels the communication endpoint associated with the client using the “online” OAuth tokens embedded in those requests.

The app can fetch a client’s OSN data by through the appropriate Service driver, which is responsible for labeling retrieved data items. In addition, the app can make arbitrary HTTP requests through urllib as long as no tainted data leaves the system through this library, i.e., outgoing requests must not contain tainted data. Data objects can be saved persistently in a Taint-preserving storage that regulates access to data and serializes/deserializes taint labels. In addition, the system can schedule an offline request such as cron to
do a batch processing using “offline” (previously saved into a local storage) OAuth tokens.

The application can export processed data outside of the system either directly to the Client or send the update to the OSN via the appropriate Service driver, e.g., to modify user’s profile, upload a picture or send a notification. For the former, the reference Monitor checks the taint labels of the outgoing objects and ensures that the Client can access these objects. In the latter case, a Service driver is responsible for verifying that the taint sink is authorized to receive the object.

In the remainder of the section we will discuss the most important system components in greater detail.

4.6.3 OSN drivers

As mentioned earlier, we require every OSN supported by MUTT to have a corresponding Service driver. Each driver is a small program module that proxies incoming or outgoing data from the specific OSN provider to the app instance and from the app instance to the OSN provider accordingly. All data retrieved from the OSN on behalf of the application is labeled before being handled to the app. When the application instance needs to update remote OSN data, it has to use the driver’s interface for submitting such a request. In this case the driver verifies that tainted data satisfies default policy #1, i.e., the user receiving the outgoing object is in a social graph of the owner (or all owners in case of an aggregated data item) of this object.

4.6.4 Language runtime

In this subsection we will discuss our taint label design and its propagation in Python.

4.6.4.1 Taint structure

Performance is a critical concern for dynamic taint analysis because the runtime must interpose on all operations in order to update items’ labels. The key challenge for a PaaS
taint tracker is to provide an efficient way to represent and update labels that is also flexible enough to support the policies and functionality need by users and applications.

For our design, we introduce a simple taint label that contains information about a data source. We do not currently support arbitrary, cross-service policies (see Section 4.5), and so it is sufficient to propagate only information about an item’s owner.

Listing 4.1: Taint structure

Listing 4.1 shows the structure of the taint label we used in our design. The taint label contains a token-set – a structure that stores one or more OAuth tokens used to retrieve or construct the tainted object. It also has a flags field that carries additional information about a data object. Currently, we use this field to indicate special data items such as OAuth tokens or passwords so that they can be given special treatment (e.g., to forbid using such objects in conditional statements). However, this field can be used more generally, for example, to carry information about object’s data type. It can also be used to encode other information about how data should be handled, including retention or inter-user sharing policies. We leave a discussion of what to do with these extra bits for future work.

4.6.4.2 Taint labeling and propagation

Any object can be tainted by the application through the extended class interface. In MUTT, all objects retrieved through a Service driver get an appropriate taint value and their token-set contains at least the OAuth token used for retrieval. After creation the taint structure cannot be changed from the Python-level code. Attempts to create an object from an already tainted object raise an exception.

A taint structure is propagated from one object to another in two cases: (1) during the assignment of a tainted object, and (2) when two or more objects are used to produce a new one. The former is just a trivial copy of the taint structure, whereas for the latter requires
merging the inputs’ taint labels. To merge taint labels we use a union of each input’s token-set and bitwise OR operation for flags fields. Performing a union operation of two or more token-sets can be slow when the sets are large. However, we found that the operations that involve merging many objects with different token-sets are usually scheduled as the offline jobs, making the performance not critical.

4.6.4.3 Taint Sinks

Every tainted object can be exported outside of the system through two interfaces: it can be sent to a user as a response or used to update OSN data via the appropriate driver. By default, the reference monitor ensures that all exported data can only be accessed by an authorized user, i.e., the owner of this data or her friends. More formally, object \( O \) with token-set containing tokens \( t_1, t_2, \ldots, t_n \) can leave the system through a sink authorized by token \( t_k \) only if \( \forall t_i, i = 1, 2, \ldots, n : \) an owner of \( t_k \) is a friend of the owner of \( t_i \).

4.6.4.4 Implicit flows

We also introduced a special taint flag that prevents non-conditional objects from being accessed in conditional statements. We believe that for some sensitive data such as OAuth tokens this approach would drastically reduce the chances of leakage through covert channels. If a non-conditional object is used in a conditional statement, the language runtime throws an exception error (Listing 4.2, lines 1-4). However, it is possible to clear this flag for certain aggregate functions, such as strlen or count (Listing 4.2, lines 6-9). We leave this decision to specific implementations since some data may be too sensitive even for aggregation.

```plaintext
# token_obj is tainted as non-conditional
1 if token_obj is guess_value:
2   # raises an exception
3       return guess_value
4
5 """ however, this is fine if function len clears
6 the non-conditional flag off """
7 if len(token_obj) == 0:
8       print "wrong token size"
9```

85
4.6.5 Platform

4.6.5.1 Persistent storage

Besides language support, the application should be able to save a tainted object and later retrieve it without losing its taint value. AppScale uses the Google Datastore API as a database-agnostic interface to store objects persistently. We modified it to support serialization and deserialization of taint labels so that tainted objects could be stored and retrieved using this API. These modifications were performed in the Google Datastore code, which allowed us to support all databases shipped with the AppScale.

Memcache is a distributed in-memory key-value data store useful for caching. We patched memcache’s implementation of its binary protocol so that taint values could be serialized and deserialized when storing and retrieving tainted objects.

4.6.5.2 Application server

AppScale’s built-in application server (AppServer) is responsible for serving incoming requests and sending response objects back to clients. In MUTT the AppServer also serves as a reference monitor and must check if a response is eligible to leave the system based on the object’s token-set and sink’s label.

4.6.6 MUTT Policy Enforcement

Enforcement of default policy #1 is embedded in the design of MUTT. Assuming that all sensitive data is properly tagged, the propagation logic is correctly implemented, and the Platform is not compromised, we claim that user’s sensitive data cannot be exposed outside of her immediate social graph.

MUTT supports caching policies (default policy #2) by providing a secure API that an OSN Service Provider can use to mark application data that belongs to a specific user
for retention. For example, a Service Provider can use this API to delete a user’s app data when she uninstalls the application from her OSN profile. Our system can also expire certain data items by including into information about the time it was fetched in the taint label. MUTT’s offline jobs can periodically crawl the database to see if a certain data item is expired and delete it if needed. However, since this requires more specific instructions and policies from the OSN Platform, this feature is currently unsupported.

Finally, before exporting any tainted object, MUTT’s reference monitor verifies that the data owner and the recipient of this data are OSN friends. This ensures that even in case of a locally cached social graph the ACL of the object is the most recent. Similarly MUTT can verify the consistency of complex ACLs. A more difficult requirement is to ensure that an item’s ACL has not changed, e.g., from “public” to “private.”

There are several simple ways for MUTT to handle cached ACLs. First, the reference monitor could retrieve the current ACL associated with an object before it is released. This would provide strong privacy guarantees but introduce expensive checks into the critical path. Alternatively, MUTT could periodically refresh the ACL associated with an object. This would remove checks from the critical path, but at the cost of ACL freshness. As stated earlier, the Paas’s handling of cached data can be exported to services and users so that they can decide whether cached data is handled appropriately. Ideally, OSNs would cooperate with MUTT by actively invalidate cached ACLs as they change, but no OSN currently supports such callbacks.

4.7 Implementation

We have built a MUTT prototype based on the design described in Section 4.6. We used an open source implementation of Google AppEngine, AppScale [36], as a primary platform for our system. In this section we discuss our modifications to the Python runtime and its libraries to support taint propagation as well as other changes we made to AppScale.
Overall, we modified fewer than 2,000 lines of C-code (including adding new drivers and policies) in the Python source and about 5,000 lines of Python-code in AppScale.

4.7.1 Python
4.7.1.1 Python runtime

To make our prototype’s code more extensible, we strived to modify the existing Python runtime such that most changes were at the highest level of abstraction. However, since Python is written in C, we added the taint structure to PyObject.HEAD – a preamble of every object in Python. This allowed us to implement generic drivers for tainting and merging Python objects. Moreover, this helped us minimize the changes required to the Python code. Instead of making changes to every primitive type, we could place most of the taint propagation and merging logic in abstract and object class implementations. The downside of this approach is that even non-taintable objects such as containers or function objects contain a taint structure. We discuss the overhead of this design choice in Section 4.8.1.

Adding a taint support to all of Python’s primitive types allowed us to automatically support any complex objects such as classes or containers with tainted members. For example, an array or dict may include tainted and untainted items, and a class object could have tainted fields. However, when such a complex object is serialized (for example, in order to be exported) with the use of the tainted items, the resulting taint label will be constructed from their taint labels.

Additional modifications were required to cope with the fact that all objects in original Python are immutable, allowing its developers to optimize the creation of primitive types. For example, small integers are constructed beforehand, stored in a table and can be shared among different objects. Thus, when an application wants to create a new object with a small int value or a short string, instead of expensive construction of a brand new object, the runtime just returns a reference to a previously created object and increments its ref-
count by one (for automatic garbage collection). However, this behavior is not desirable for tainted objects, since otherwise we cannot have two objects with the same value but different taint labels. We modified this mechanism to prevent sharing of objects when they are tainted, hence disabling the original optimization for tainted objects.

4.7.1.2 Libraries

As discussed in Section 4.6.2, for our security guarantees to hold our system needs to (1) disallow modifying the taint tags and (2) control the sinks and sources of sensitive data. It is easy to achieve the former by disabling the import of dynamic C-modules, since only C-level code can modify objects’ taint structure. As a consequence, we had to statically link all common dynamic C-modules.

To control taint sinks, we modified the Python socketmodule to provide a more narrow interface and disallow arbitrary network connections. Besides Appscale’s built-in application server and Django, the only uncontrolled source of data in the AppScale is urllib – a Python library for working with HTTP. We rewrote this library in C so that only untainted data could leave the system through urllib’s interface.

Since all data flowing from OSNs should be tainted according to the specified policies, each OSN source needs to have a trusted driver that can retrieve data from the service on behalf of the user and taint it before handling this data to the application. We implemented two drivers to handle Facebook and Twitter data.

4.7.2 AppScale

We used AppScale as a basis for building a prototype of our system. AppScale can be deployed in Amazon EC2 as well as on a cluster of local machines. To support taint-tracking and security policies we changed the application server to handle the tokens and modified the database drivers so that tainted objects could be saved and retrieved from persistent storage without compromising data security.
We also modified \textit{memcache} library shipped with Appscale. Memcache is a distributed in-memory key-value data store useful for caching. We patched memcache’s implementation of its binary protocol to enable taints’ values serialization and deserialization for storing and retrieving tainted objects.

4.8 Evaluation

To be practical and deployable, MUTT must provide reasonable performance. To understand the sources of overhead in our prototype, we wanted to know (1) what is the storage overhead of adding a taint structure to every object in Python runtime, and (2) how do creation and propagation of taint labels and condition checks to prevent implicit flows affect the performance of basic Python operations.

For evaluation we used Appscale’s image with the modified Python runtime on a \textit{large instance} (64-bit architecture, 7.5 GB memory, 2 virtual cores) of Amazon EC2 cloud computing infrastructure.

4.8.1 Storage overhead

Listing 4.1 shows the actual taint structure used in our prototype. The size of this structure is 12 bytes on a 64-bit machine. As described in Section 4.7.1, each object in our Python runtime stores a pointer to a taint structure. Thus, the size of PyObject\_HEAD increases from the original 24 bytes to 32 since the size of the pointer is 8 bytes. In addition, each object in Python has a pointer to a PyTypeObject that serves as a reference to the common methods and properties shared among all objects of the particular type. We added a flag to PyTypeObject indicating whether this data type is taint-aware for fast checks. The unmodified size of each PyTypeObject is 384 bytes, so after adding the flag of type \textit{int} (along with additional 8 bytes from PyObject\_HEAD) it increased to 396 bytes.
4.8.2 Runtime overhead

4.8.2.1 Microbenchmarks

First, we wanted to measure the overhead introduced to basic primitive operations by adding the taint-tracking mechanism to Python. Table 4.3 shows the results of the microbenchmarks. We recorded the latencies of three basic operations in (a) the unmodified Python and the modified runtime with (b) non-tainted, and (c) tainted objects.

To evaluate the performance of tainted objects we varied the number of tokens in the arguments’ token-sets from 1 to 50. In case of 1 token we tainted only one argument with exactly 1 token. For 10, 20 and 50 tokens we used objects with each token-set containing 5, 10 and 25 tokens accordingly. The size of token we chose for the evaluation is 20 characters (Facebook’s fbuid is 21 and Google’s 15).

The latency of assigning a tainted integer is 0.22 $\mu s$ and it is comparable to assigning a non-tainted object (0.19 $\mu s$). Since our taint-merging policies are identical for all binary operations we do not report a breakdown of our results for all of them and show only the latencies for string concatenation and integer addition. Merging a non-tainted object and an object with 1 token is about 2 times slower than the same operation in unmodified runtime. We believe, that the scenario with 1 token represents the common case when a tainted object is being merged with non-tainted data. On the other hand, the latencies of merging objects with multiple tokens grows linearly with the number of tokens in their token-sets. However, we envision that the operations that require manipulations with data objects containing multiple tokens (e.g., complex object) can be done offline (e.g., as cron jobs) and thus they are not as time critical as online requests.

Finally, we measured the overhead of our implicit flow prevention mechanism. The additional check adds about 0.01 $\mu s$ to the original comparison.
Table 4.3: Microbenchmarks results

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unmodified (µs)</th>
<th>Untainted (µs)</th>
<th>Tainted (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 token</td>
</tr>
<tr>
<td>Assign (int)</td>
<td>0.19</td>
<td>0.19</td>
<td>0.22</td>
</tr>
<tr>
<td>Int addition</td>
<td>0.10</td>
<td>0.11</td>
<td>0.21</td>
</tr>
<tr>
<td>String concat</td>
<td>0.13</td>
<td>0.14</td>
<td>0.26</td>
</tr>
<tr>
<td>Compare (int)</td>
<td>0.07</td>
<td>0.07</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Table 4.4: Latencies of selected tests from *pybench* suite

<table>
<thead>
<tr>
<th>Test</th>
<th>Untainted (ms)</th>
<th>Tainted (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateStringsWithConcat</td>
<td>115</td>
<td>250</td>
</tr>
<tr>
<td>StringSlicing</td>
<td>93</td>
<td>175</td>
</tr>
<tr>
<td>CreateUnicodeWithConcat</td>
<td>104</td>
<td>246</td>
</tr>
<tr>
<td>SimpleIntegerArithmetic</td>
<td>123</td>
<td>278</td>
</tr>
<tr>
<td>SimpleFloatArithmetic</td>
<td>120</td>
<td>266</td>
</tr>
<tr>
<td>SimpleLongArithmetic</td>
<td>104</td>
<td>272</td>
</tr>
<tr>
<td>SimpleIntFloatArithmetic</td>
<td>105</td>
<td>269</td>
</tr>
</tbody>
</table>

4.8.2.2 *Benchmarks with pybench*

We also evaluated our prototype on *pybench* – standard test suite shipped with the Python source code. In contrast to our own microbenchmarks where we measure the latencies of isolated operations, *pybench* is a more comprehensive set of tests that performs a combination of various operations over specific types of data.

The latencies of running noteworthy tests from *pybench* using both untainted and tainted objects are shown in Table 4.4. For these tests we used tainted objects with 1 token per object. Similar to our previous results, the latency of the tainted objects is also about 2.5 times higher than the original.

4.9 Conclusion

This chapter has presented the design and implementation of MUTT. MUTT is a set of PaaS extensions that prevents unauthorized internal sharing through OSN applications. MUTT’s design is informed by a survey of 170 OSN applications as well as the APIs and terms of service for several OSNs. By integrating taint tracking into a PaaS runtime
and storage and by giving the trusted PaaS control over cached data, MUTT provides much stronger confidentiality guarantees for third-party code with access to sensitive OSN data.
The importance of protecting privacy and confidentiality in OSNs and systems in general has attracted significant attention from the research community. In this chapter we discuss prior work that is related to the systems proposed in this dissertation.

Chapter 5.1 compares Vis-à-Vis and Confidant to the most relevant related work. In Chapter 5.2 we discuss in detail prior work that is related to the proposed Multi-User Taint Tracker system.

5.1 Privacy-preserving OSNs

5.1.1 Related work: Vis-à-Vis

PeerSon [30] and [41] are two recent proposals for peer-to-peer OSNs with similar goals and structures as Vis-à-Vis. PeerSon relies on encryption to protect users' replicated data, and uses a two-tier architecture in which the first tier is organized as a DHT for lookups. The second tier allows users to exchange data and is implemented as a strongly-connected overlay. Cutillo et al [41] utilize a DHT for lookups and storing social groups' information. In this scheme, logical “matryoshkas” store data and are organized as concentric “rings of friends”. Profiles are replicated in the innermost layer of a user’s matryoshka to increase
availability of the data. In contrast, Vis-à-Vis is designed to support flexible degrees of location sharing among large groups of users, possibly in the absence of strong social ties. In addition, Vis-à-Vis relies on a compute utility to improve service availability.

Puttaswamy et al [85] protect users’ location information assuming that storage providers are untrusted. Vis-à-Vis represents a philosophical departure by entrusting compute utilities such as EC2 with access to unencrypted versions of this data. However, we feel that trusting compute utilities is warranted given their business models and terms of use. As TPM-enabled services are more widely embraced by utilities, the trustworthiness of users’ VISs will increase. Furthermore, Vis-à-Vis leverages the trust it places in compute utilities and the VISs they host to provide services such as range queries over location data that are not possible when servers hold encrypted data.

The hierarchical organization of Vis-à-Vis shares many traits with both P2P-RTree [78] and Census [40], although neither is focused on OSN privacy. P2P-RTree is a spatial index designed for peer-to-peer networks. The subset of location-tree information maintained by VISs in Vis-à-Vis is very similar to the information stored by peers in P2P-RTree. However, the latter does not provide any fault-tolerance mechanisms or consistency guarantees.

Census [40] is a platform for building large-scale distributed applications that provides a consistent group membership abstraction for geographically dispersed nodes. Census allows the entire membership to be replicated at all nodes, and tightly integrates a redundant multicast layer for delivering membership updates efficiently in the presence of failures. Vis-à-Vis also uses leases and multicast to maintain consistent views of the membership among participating VISs, but does not require the entire tree to be replicated at each node because users are likely to be involved with many groups.

5.1.2 Related work: Confidant

FriendStore [110] is a backup system that identifies trustworthy storage sites through interpersonal relationships. FriendStore leverages trustworthy remote storage servers to mit-
igate long-standing fairness problems in peer-to-peer systems. A primary difference between Confidant and FriendStore is the nature of the data they must manage. Backup data is not meant to be shared and thus FriendStore does not require the same level of complexity to manage access-control policies that Confidant does.

While Confidant leverages knowledge of the social graph to provide data privacy without compromising data processing in a decentralized OSN, SPAR [84] uses social information to improve the scalability of centralized OSNs such as Facebook or Twitter. By co-locating the data of socially proximate users on the same physical machine, SPAR can reduce the time to compose a user’s update feed and eliminate network traffic. SPAR may also be useful for decentralized systems such as Confidant for reducing the number of storage servers clients have to communicate with.

Cimbiosys [87, 115] comes closest to Confidant’s scheme for managing replicated access policies. The two systems have a great deal in common, but their are two main differences. First, Confidant eliminates write conflicts by serializing a user’s updates through her name server. Second, in Cimbiosys, policies are bound to immutable object attributes (labels), while in Confidant, policies are bound to individual data items. By binding policies at a finer granularity is more appropriate for OSNs, where users often want to change the permissions of a single item. However, Confidant can also enable efficient bulk policy changes due to the limited scale of user’s personal OSN data sets.

Most of the proposed decentralized OSNs, such as Persona [26], NOYB [56], fly-ByNight [74], PeerSoN [31], and others [21, 45, 97, 103, 86], assume that the underlying storage service responsible for holding users’ personal data is not trustworthy.

Persona [26] assumes that remote storage servers are untrusted. We have articulated the advantages of locating trusted storage servers throughout this dissertation. However, it is worth noting that Confidant’s collusion protections are weaker than those provided by Persona. Persona uses an attribute-based encryption scheme to defend against attacks in which colluding users combine keys to falsely claim membership in an intersection of
groups. In Confidant we trade robustness to collusion attacks for a faster, simpler protection scheme based on common symmetric-key algorithms.

NOYB [56] encrypts data that users hand to the service which is appealing because it may allow users of existing OSNs to retain their existing profiles. It is unclear however, if NOYB can generalize to non-textual information, while both Vis-à-Vis and Confidant can secure arbitrary data types.

Like NOYB, flyByNight [74] uses encryption to ensure that sensitive data is never posted to OSNs in unencrypted form. flyByNight is also appealing because it allows users to continue using existing OSNs and the social state that users have accumulated there. However, flyByNight remains vulnerable to several types of attack from within the OSN, which decentralized approaches avoid by doing away with centralized OSNs.

Lockr [109] is an identity-management tool for OSNs that allows users to codify their relationships through social attestations. Confidant borrows this idea from Lockr and uses it to manage groups of users.

5.2 Privacy-preserving web frameworks

5.2.1 Language-based information flow control

Researchers have been extending programming languages by adding taint checking capabilities to secure applications and systems for over a decade [79, 80, 59, 104, 88, 57].

Some languages [79, 80, 59] provide static information flow analysis to ensure proper usage of data within an untrusted program. Whereas, other approaches [104, 88, 57] provide an ability to check data flow dynamically at runtime.

MUTT builds on and inspired by many of the techniques and approaches pioneered in the aforementioned work. However, one of the fundamental design goals for MUTT was to run the existing third-party applications within the existing OSN ecosystem. In contrast, most language-based approaches are incompatible with legacy software and systems.
Fabric [72] is a distributed platform that enables secure computations across multiple mutually distrustful machines. It uses a special Java-like language for consistent object dissemination and remote procedure calls. As MUTT, Fabric can track sensitive information on a granularity of an object. In contrast to Fabric, the main focus of our work is to enforce the adherence of third party applications to the supplied access and data flow policies.

Resin [116] is a language runtime that enables data flow assertions when tainted data is about to leave the system. Resin assumes that programmers manually annotate objects containing sensitive data, and supply merging and export policies for such objects.

Resin is the closest to our system in terms of design choices and implementation; hence it might be trivial to use Resin for building a policy-enforcing cloud service like ours by tagging sensitive data at the sources and automatically providing the appropriate policies. However, to make such a system deployable one has to solve a number of challenging problems. First and foremost, one has to come up with the default privacy policies that could be embedded into the system to add reasonable security and privacy assurance to the OSN users without breaking the ecosystem of the third-party applications. We did that by reviewing both existing third-party apps and Platform’s privacy policies. Second, one must ensure the isolation of the platform’s components and language runtime. Third, one must provide a way to handle sensitive data from multiple service providers and multiple users to support mashups.

5.2.2 System-based information flow control

Researchers have also been using system-based information flow control mechanisms to defend against application bugs, system misconfigurations and data leakage [44, 63, 117, 118, 102, 64].

AsbestOS [44], Flume [63] and HiStar [117] are UNIX-like operating systems that enforce an information flow control between processes. They ensure that a process has an
appropriate set of labels when communicating with another process, thus mitigating the exposure in the case of possible application bugs and configuration errors. MUTT takes a similar approach by relying on a taint propagation as an information flow control mechanism. However, HiStar’s and Flume’s taint granularity are at a process level which would require decomposing even a simple OSN application into multiple functional components, e.g., a mashup getting data from multiple service providers.

DStar [118] is a framework that extends DIFC support to distributed systems. In DStar mutually distrustful components can comprise a secure system by relying on an exporter that translates local protection of data into a network protection mechanism and vice versa. DStar is particularly useful to glue distrustful components of a large distributed system with different access controls, e.g., multi-tiered applications, so that a compromise of a one machine would not lead to the compromise of the whole system.

Similarly to Fabric [72], DStar’s main focus is on providing end-to-end security guarantees in distributed systems, whereas our focus is to enforce third-party apps to respect supplied data flow policies. However, we think it is possible to use a system like DStar or Fabric as a step further to provide a fully secure distributed platform.

xBook [102] is a framework for building privacy-preserving third-party applications for OSNs. It is similar to MUTT as it helps to prevent leaking users’ data through the malicious or buggy applications running outside of the service providers’ datacenters. Unlike MUTT, xBook can also defeat client-side attacks by isolating DOM components with the different privacy expectations, proper event handling and arbitrating information flow between various components of the application through the client-side xBook API. We did not address the client-side threats in our work and concentrated on the server-side attacks and violations. We argue, however, that compared to the server-side attacks, the client-side attacks are much easier to detect and even prevent by, for example, manual or automatic inspection of HTML/javascript code and isolation using existing browser extensions. In addition to that, xBook is also different from our system in other ways. xBook requires
the applications to be split into the functional components written in ADsafe – a safe subset of javascript language. In contrast, MUTT gives the PaaS environment where users can deploy standard Django applications with the minimum changes. As MUTT, xBook also uses a taint-propagation mechanism to prevent a potential leakage of the sensitive data, however, its taint granularity is a component level (similar to Flume), whereas we propagate taint labels on an object level.

World Wide Web Without Walls (W5) [64] proposes a new web architecture where (i) users’ data is stored in a trusted infrastructure maintained by a trustworthy entity and (ii) this data can be safely accessed by authorized applications/services. W5 guarantees confidentiality of users’ data and prevents its unintended disclosure by decoupling applications from data and leveraging DIFC technology.

One can argue that W5 would solve most of the problems discussed in this dissertation. In fact, Vis-à-Vis and MUTT resemble building blocks of W5 but in a smaller scale. Vis-à-Vis leverages cloud as a trusted data storage, whereas MUTT uses taint-tracking to enforce third-party apps to adhere to the privacy policies supplied by the service providers. However, to make W5 a reality one has to re-engineer the entire Internet and its ecosystem. On the other hand, all systems proposed in this dissertation can be deployed without requiring fundamental changes to the underlying technologies and infrastructure.
Online social networks is the essential part of the modern Internet but they present a range of unique privacy threats to their users. The trust and privacy guarantees of the existing OSN services are based only on a single entity – good will of service providers. However, the goals and motivation of the service providers are often in odds with users’ privacy expectations.

This dissertation presents three systems that help to mitigate the fundamental issues associated with the status quo of the existing OSNs and provide better privacy guarantees to the OSN users. We achieve this goal by shifting trust from service providers to more trustworthy entities.

Vis-à-Vis relies on a cloud computing providers’ business model to build a privacy-preserving OSN platform for location-based services. Confidant achieves the same goal by leveraging trust embedded in the social network. And, finally, MUTT utilizes information flow control mechanisms to enforce third-party OSN extensions to adhere to the policies provided by the service providers.

In this chapter we summarize the research contributions made by this dissertation and discuss possible future research directions.
6.1 Contributions

This dissertation makes contributions in three major areas. The first area is conceptual – it consists of the novel ideas generated by this work. The second area is a set of artifact system that we developed to validate this dissertation. The final area of contribution is the experimental evaluation of the systems that validate the feasibility of this dissertation’s thesis.

6.1.1 Conceptual contributions

This dissertation makes three following conceptual contributions:

- We showed that building a privacy-preserving OSN platform can be achieved by shifting trust from OSN providers to other entities, e.g., cloud providers or close friends.
- We introduced the notion of a “socially-informed” replication. This replication leverages close social ties for building privacy-preserving OSN services with extensible and scalable functionality support.
- We studied existing third-party OSN applications’ structures and service providers’ privacy requirements to elaborate a set of privacy policies that could be used to build a privacy-preserving platform for third-party cloud-based OSN extensions.

6.1.2 Artifacts

In the course of this dissertation, we have developed three major artifacts to validate the thesis: Vis-à-Vis, Confidant, and MUTT. We built prototypes of these systems to confirm the feasibility of systems’ implementations and their usability.

- We built Vis-à-Vis, a decentralized OSN framework that supports the range queries, and low latency location updates with consistency guarantees.
• We built a prototype of Confidant, a privacy-preserving OSN using commodity computers and integrated it into Facebook using a Firefox browser extension.

• We also implemented several applications to validate that “socially-informed” replication can support rich data-processing functionality not scalable in case of OSN architectures using untrusted storage.

• We built MUTT by adding taint-tracking support to Python’s language runtime, AppScale and third-party libraries necessary for running OSN extensions and preventing them to mishandle users’ data.

6.1.3 Evaluation results

The evaluation results in this dissertation present the most important insight: it is possible to build decentralized privacy-preserving OSN services with rich functionality and acceptable performance.

• The evaluation of Vis-à-Vis shows that in a real-life scenario (i.e., when location updates dominate initial check-in operations) a distributed OSN service has performance comparable to its centralized counterpart.

• We also showed that a hierarchical structure is very effective for building the location-based services since it requires minimal state exchange between local nodes whenever a user updates her location.

• The evaluation of Confidant demonstrates that “socially-informed replication” allows to scalably support data-processing scripts. Their performance is 3-30 times better compared to competing approaches that use untrusted storage. This holds even for simple applications like trending topics due to the size of a user’s social graph. However, this trend will be even more evident with complex applications that process large data such as video or music.
• The evaluation of our unoptimized prototype of MUTT confirms feasibility of using taint-tracking mechanism to enforce third-party OSN apps to adhere to the services’ policies and prevent leaking privacy-sensitive data.

6.2 Future work

6.2.1 Cloud-desktop hybrid decentralization

In Chapter 3 we discussed that the main disadvantage of hosting VISs on desktop machines is the low availability of these machines. Although “socially-informed” replication mitigates this problem, nodes holding replicas may still experience correlated failures. For example, if a user replicates her data on desktop machines that are located in the same household, all replicas may be down if the house loses electricity.

One way of improving the availability in a presence of correlated failures is a hybrid approach based on Vis-à-Vis and Confidant. Thus, we would like to design a decentralized OSN as a combination of cloud-based and desktop-based decentralization. During normal operation, a user’s data is served from a VIS running in the user’s own desktop machine. When this machine becomes unavailable, a standby VIS is resumed inside the utility computing infrastructure. This solution could combine high availability with low cost because the cloud-hosted VIS would provide a stable backup while remaining quiescent most of the time.

6.2.2 Privacy of the requester

The importance of protecting privacy in OSNs has attracted significant attention from the research community. And as we showed in this dissertation, decentralized OSNs arguably give the most privacy guarantees while providing required functionality and scalability.

However, privacy-preserving authentication and authorization in the systems without trusted centralized service is not yet explored. One can imagine, that users may utilize an ad-hoc or federated authentication systems such as OpenID [82] to log-in to their friends’
machines to request data they are authorized to see. The problem is that in decentralized OSNs each user has a complete control over her machine. Hence, data owner would have the entire log of all requests coming to the server. This property might be highly undesirable and seen as a violation of requester’s confidentiality. We think that the ability to track friends’ activity would discourage user interaction in the social network. In fact, Facebook explicitly prohibits exposure of such information either through its API or any affiliated applications [5].

One way of mitigating the problem of confidentiality of the requester in decentralized OSNs is by using social attestations [109] instead of the explicit authorization. A social attestation is a group certificate issued and signed by a particular user, e.g., Bob’s family or Alice’s classmates. Every friend in this groups has an unforgeable group-wide certificate which he or she can use to retrieve an item with the access control list that requires a membership in the group. Thus, friends do not have to reveal their identity to get data, instead, they prove their right to access items using different social attestations.

Despite of its obvious advantages, a social attestation is prone to the same types of attacks as K-anonymity [90]. First, constructing a group with only one friend and assigning a data item with the ACL requiring to prove membership in this group is essentially the same as an ad-hoc authentication. Second, if a profile has multiple data items with different access control lists requiring distinct social attestations, a user requesting these items might leak her identity through intersections of the items’ ACLs and temporal locality of the distinct requests. We believe, that it is generally hard to defend against a former attack since a user should be able to set her ACLs as she desires (even if the ACL includes only one person). However, we want to explore a defense against leaking the requester’s identity through ACLs intersections.
6.2.3 Declarative language for defining ACLs for OSNs

Driven by design and implementation of Vis-à-Vis, Confidant and MUTT, we have started to look for ways to improve expressiveness and enforcement of privacy settings for the OSN users regardless of the underlying system. The main question we want to answer is whether it is possible to have a simple but yet rich way to define, understand and enforce privacy preferences in the OSN context?

We argue, that there is a need for a unified declarative language for this purpose. If this language is designed and accepted by major online social services, it would help users and application developers to reason better about privacy in different OSNs. In addition, this unified language would simplify a cooperation between various social services, e.g. LastFM and Facebook, since the translation of the privacy preferences from one service to another would require little if any effort. Finally, we envision that the run-time environment of this language could be incorporated into a system similar to MUTT.
**Bibliography**


Biography

Amre Shakimov was born in Karaganda, Kazakhstan on January 2, 1982.

His research interests include privacy and security in Online Social Networking services and cloud computing. He received a B.S and M.S in Applied Mathematics and Computer Science from Karaganda State University in 2003 and 2005 accordingly.