# Persona: An Online Social Network with User-Defined Privacy

Randy Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee
University of Maryland
{randofu, bender, nspring, bobby}@cs.umd.edu

Daniel Starin
Starin Consulting*
dstarin@starinconsulting.com

## ABSTRACT

Online social networks (OSNs) are immensely popular, with some claiming over 200 million users [10]. Users share private content, such as personal information or photographs, using OSN applications. Users must trust the OSN service to protect personal information even as the OSN provider benefits from examining and sharing that information.

We present Persona, an OSN where users dictate who may access their information. Persona hides user data with attribute-based encryption (ABE), allowing users to apply fine-grained policies over who may view their data. Persona provides an effective means of creating applications in which users, not the OSN, define policy over access to private data.

We demonstrate new cryptographic mechanisms that enhance the general applicability of ABE. We show how Persona provides the functionality of existing online social networks with additional privacy benefits. We describe an implementation of Persona that replicates Facebook applications and show that Persona provides acceptable performance when browsing privacy-enhanced web pages, even on mobile devices.

## Categories and Subject Descriptors

C.2.0 [**Computer Systems Organization**]: Computer Communications Networks—*General*; C.2.4 [**Computer Systems Organization**]: Computer Communications Networks—*Distributed Systems*; E.3 [**Data**]: Data Encryption; H.3.4 [**Information Systems**]: Information Storage and Retrieval—*Systems and Software*

## General Terms

Design, Security, Performance

## Keywords

Persona, OSN, Social Networks, ABE, Privacy, Facebook

*Mr. Starin's work on this project was conducted as part of a graduate course at the University of Maryland.


*SIGCOMM'09,* August 17–21, 2009, Barcelona, Spain.


## 1. INTRODUCTION

Online social networks (OSNs) have become a de facto portal for Internet access for millions of users. These networks help users share information with their friends. Along the way, however, users entrust the social network provider with such personal information as sexual preferences, political and religious views, phone numbers, occupations, identities of friends, and photographs. Although sites offer privacy controls that let users restrict how their data is viewed by other users, sites provide insufficient controls to restrict data sharing with corporate affiliates or application developers.

Not only are there few controls to limit information disclosure, acceptable use policies require both that users provide accurate information and that users grant the provider the right to sell that information to others. Facebook is a representative example of a social network provider. The Facebook "Statement of Rights and Responsibilities" [9] *requires* that users "not provide any false personal information on Facebook" and "keep [their] contact information accurate and up to date." Further, it states that users "grant [Facebook] a non-exclusive, transferable, sub-licensable, royalty-free, worldwide license to use any IP [Intellectual Property] content that [they] post on or in connection with Facebook."

Cryptography is the natural tool for protecting privacy in a distributed setting, but obvious cryptographic schemes do not allow users to scalably define their privacy settings in OSNs. Users want to be able to share content with entire groups, such as their friends, their family, or their classmates. Public key cryptography alone is unsatisfactory when managing groups in an OSN: either users must store many copies of encrypted data, users are unable to give data based on membership in multiple groups, or users must know the identities of everyone to whom they give access.

To meet the privacy needs of an OSN, we propose Persona, an OSN that puts policy decisions in the hands of the users. Persona uses decentralized, persistent storage so that user data remains available in the system and so that users may choose with whom they store their information. We build Persona using cryptographic primitives that include attribute-based encryption (ABE), traditional public key cryptography (PKC), and automated key management mechanisms to translate between the two cryptosystems.

Persona achieves privacy by encrypting private content and prevents misuse of a user's applications through authentication. Persona allows users to store private data persistently with intermediaries, but does not require that users trust those intermediaries to keep private data secret. Modern web browsers can support the cryptographic operations

needed to automatically encrypt and decrypt private data in Persona with plugins that intercept web pages to replace encrypted contents. Lastly, Persona divides the OSN entities into two categories: users, who generate the content in the OSN, and applications, which provide services to users and manipulate the OSN content.

The rest of this paper is organized as follows. We describe the cryptographic primitives and how they comprise the correct cryptographic systems for Persona in Section 2. We present novel compositions of ABE and PKC functions that allow users to create flexible and dynamic access policies in Section 3. We describe the role of OSN applications in Persona and show that Persona supports existing OSN applications in Section 4. We present significant features of our implementation in Section 5. We evaluate the performance of Persona using data from a Facebook crawl and ABE microbenchmarks on a mobile device in Section 6. We describe related work in Section 7, discuss additional problems beyond the scope of this paper in Section 8, and conclude in Section 9.

# 2. CRYPTOGRAPHY IN A PRIVATE OSN

There are two tasks for encryption in building the private online social network. The first is to restrict the information available to applications as precisely as possible, so that individual organizations are not entrusted with large volumes of personal information. Although it is tempting to focus only on the exchange of information with friends, some applications may benefit from limited access to a user's profile, location, or messages, while carefully avoiding broad exposure.

The second task is to restrict the information shared with "friends" to what might be appropriate. We quote "friends" here because the type of social link might be more than, less than, or different from "friend." Family, neighbor, co-worker, boss, teammate, and other relations might define a connection in the social network. That connection is often simply termed "friend", regardless of the actual, off-line relationship. A user's decision to accept one of these pseudo-friends into their neighborhood (and avoid discussing certain topics) or exclude them (and avoid the benefits of social networking) represents a dilemma that can be avoided, if users may flexibly classify their "friends."

Alone, these two problems may be easily solved. A social network could help users define access policies that include or exclude defined groups of friends accessing different pieces of information. Such a feature would allow a user to tweet "called in sick to work" without telling co-workers. In practice, users segregate work colleagues from personal friends by subscribing to different social networks. To provide such functionality efficiently without the assistance of a trusted application provider requires some form of cryptographic support for group keying. In this section, we define two methods to share information with groups in an OSN.

What makes the OSN setting different from typical group keying scenarios is that the sender (to the group) may not be in charge of group membership. For example, Alice may post a message on Bob's wall, encrypted for Bob's friends, without (necessarily) knowing the list of Bob's friends. Further, Alice might wish to send a message to Bob's friends who live in the neighborhood: "Let's meet up tonight". Another aspect of the OSN setting is that the number of potential groups a user might encrypt to is very large (any possible combination of friends of their friends). Cryptographic support alone is not sufficient for building a distributed online social network; it is merely a necessary tool, difficult to apply, which shapes the eventual design.

## 2.1 Model

With the abstract goals of hiding personal information from aggregators and hiding personal information from colleagues, we next refine these goals down to concrete requirements for cryptographic methods.

Each Persona user generates an asymmetric key-pair and distributes the public key out-of-band to other users with whom they want to share data. We refer to these other users as friends, though the nature of each relationship is defined by the user.

Persona allows users to create "groups" and choose which users are part of a given group. Users control access to personal data by encrypting to "groups." Restricting data to specific groups allows users to have fine-grained control over access policy, which permits exchanging data with more restrictions.

Cryptographic primitives in Persona must allow users to flexibly specify and encrypt to groups. Users may specify groups using arbitrary criteria, but we expect users to choose groups based on transparent relationships such as "neighbor" or "co-worker" or on attributes such as "football fan" or "knitting buddy." Groups created by one user do not affect the groups that can be created by another. However, to support OSN communication patterns, the groups created by one user should be available for use, not just for decryption, but also for encryption, by friends.

## 2.2 Traditional public-key approach

Traditional public-key and symmetric cryptography can be combined to form an efficient group encryption primitive [30, 37]. To create a new group from a list of known friends, Alice encrypts a newly-generated group key with the public key of each member of the new group. She then distributes this key to the members of that group and uses the key to encrypt messages to the group. The group key may be symmetric, in which case only group members can encrypt to the group, or asymmetric, which allows non-members to encrypt as well.

Distributing a new group key may coincide with sending a new message: to create a message for all of her friends, Alice might include both the keys and the data in the same object for efficiency. To efficiently reuse a group and key for many messages could require separating the keys from the data and caching the group key for use on later messages. We informally term the re-use of keys to avoid wasteful repetition of public key operations "recycling."

This protocol is computationally inexpensive, in that it does not require signatures; the worst an attacker could do is provide a faulty key that would soon be discovered. It is also flexible for the group creator, in that the original creator can enumerate any set of friends to include in the group. It is somewhat flexible for others, in that a friend who is a member of two groups ("neighbor" and "football fan") may encrypt a message for the union of these groups ("neighbor OR football fan") by encrypting the message with both group keys separately. However, a friend cannot further restrict access to an intersection ("neighbor AND football fan") without exposing the message to colluding friends that do

not match the expression (one a neighbor, the other a football fan). One could encrypt with one group key and then the other, but the colluding members of each set could decrypt the message intended for only the members with both attributes.

Allowing users to encrypt data for groups that they are not members of requires additional infrastructure. Alice can give her friends the ability to encrypt messages for any of her groups defined by an asymmetric keypair by publishing a list of her groups and their public keys. Other users consult this list to send messages to Alice's groups. However, only group members can encrypt to groups defined by a shared symmetric key.

## 2.3 ABE

Alternately, attribute-based encryption (ABE) [5] can be used to implement encryption to groups. To use ABE, each user generates an ABE public key (APK) and an ABE master secret key (AMSK). For each friend, the user can then generate an ABE secret key (ASK) corresponding to the set of attributes that defines the groups that friend should be part of. For instance, if Alice decides that Bob is a "neighbor", "co-worker", and "football fan", then she would generate and distribute to Bob an ABE attribute secret key that includes those three attributes. Bob becomes a member of the groups defined by combinations of those attributes.

In ABE, each encryption must specify an *access structure*: a logical expression over attributes. For instance, Alice can choose to encrypt a message with access structure ('neighbor' OR 'football fan'), where 'neighbor' and 'football fan' are attributes, rather than groups, and any of her friends who have an attribute secret key with either attribute will be able to decrypt the message. Alice can also encrypt to ('neighbor' AND 'football fan'). In this case, the ABE construction ensures that only friends with both attributes will be able to decrypt the message. Unlike in the traditional cryptography approach, a single encryption operation constructs the new group and provides the (symmetric) key that protects the rest of the message. Furthermore, any user who knows Alice's ABE public key can encrypt to any access structure (and thus create any group) by knowing the names and definitions of the attributes Alice defined.

ABE provides a natural mapping for the group encryption primitive that we envision for OSNs. This simplicity comes at a performance penalty: ABE operations are about 100-1000 times slower than those of RSA. These ABE operations can be avoided in practice by careful system design. Specifically, ABE defines new groups through attributes and permits sharing efficient, symmetric keys that can be "recycled" to avoid expensive operations. This approach means that ABE's performance penalty need only be paid when it provides its ease-of-use or third-party group-definition advantages, not for each operation.

Consider our example access structure ('neighbor' AND 'football fan'). In creating this group, Alice had to enumerate all her friends and distribute a new group key to matching friends. Now imagine that Bob wants to encrypt data to the same ('neighbor' AND 'football fan') group. With ABE, Bob would encrypt using ('neighbor' AND 'football fan') as the access structure. Under traditional cryptography, if Alice had pre-defined this group (and invited Bob), then Bob could encrypt using the group symmetric key. Otherwise, Bob can encrypt this message *only* if he can enumerate *all* of Alice's friends and know whether they belonged to both groups.

Using ABE allows friend-of-friend interactions without requiring enumerations of friend and attribute lists. A friend may limit who may read a response to a wall post to a more restricted group. For example, if Alice writes "I want to watch Serenity this weekend," as a post to her 'friends', Bob might reply "I have the DVD, let's watch it at my place," to Alice's 'friends' who also have the 'in-the-neighborhood' attribute. Without ABE, Bob would have to rely on Alice to have created this (intersection) group in advance. As long as users share attribute names (and their meanings) with friends, ABE provides an elegant mechanism for users to target information for friends-of-friends. The same functionality can be implemented without ABE, but requires more information exchange (lists of all friends-of-friends and their attributes) and a key distribution mechanism (that maps groups defined by friends to the group key).

# 3. GROUP KEY MANAGEMENT

We describe how Persona users define groups and how users generate and use keys corresponding to groups. Keys guard access to two types of objects in Persona: user data and abstract resources. In Persona, all users store their data encrypted for groups that they define. Any user that can name a piece of data may retrieve it, but they can only read it if they belong to the group for which the data was encrypted. Abstract resources represent non-data objects, for example, a user's storage space or a Facebook Wall. The set of possible operations on an abstract resource is tailored to the resource (for example, it is possible to *write* onto a storage space or *post* to a user's Wall). Each resource has a *home* which maintains and enforces the resource's Access Control List (ACL). The resource's *owner* may change the resource ACL and allow specific groups different levels of access to the resource. The Persona group management operations described in this section allow users to control access to data and resources. All Persona applications (Section 4) are built using these operations.

Each Persona user is identified using a single public key and stores their own (encrypted) data with a *storage service*. Users with existing relationships exchange their public keys and storage service locations out of band. Storage services support two operations for data storage and retrieval: put and get, which mimic the store and retrieve operations of a hash table. Storage is a resource in Persona, and users may grant other users (or groups) the ability to store (put) onto their storage service using the operations described in this section. Storage services are a specialized case of the broader class of Persona applications and are described in more detail in Section 4.1.

We use the notation shown in Table 1. In the algorithm listings, $u : \langle$protocol step$\rangle$ means user $u$ invokes the specified step.

## 3.1 Operations

Persona operations allow users to manage group membership and mandate access to resources. The operations combine ABE and traditional cryptography, allowing individuals to be securely added to groups defined using ABE and allowing group members authenticated access to abstract resources.

| Term | Definition |
|---|---|
| $u.SS$ | $u$'s storage service location |
| $u.K$ | Key $K$ created by $u$ |
| $(TPK, TSK)$ | PKC public/secret keypair |
| $(APK, AMSK)$ | ABE public/master secret keypair |
| $ASK$ | ABE user secret key |
| $\mathbb{AS}$ | Access structure |
| TKeyGen() | Generate RSA keypair |
| TEncrypt$(K, m)$ | RSA encrypt $m$ with key $K$ |
| TDecrypt$(K, c)$ | RSA decrypt ciphertext $c$ |
| TSign$(K, m)$ | RSA sign $m$ with key $K$ |
| ABESetup | Generate an attribute public key and master secret key |
| ABEKeyGen$(K, attrs)$ | Generate an attribute secret key with attributes $attrs$ |
| ABEEncrypt$(K, m, \mathbb{AS})$ | ABE encrypt $m$ with key $K$ and access structure $\mathbb{AS}$ |
| ABEDecrypt$(SK, PK, c)$ | ABE decrypt ciphertext $c$ with secret key $SK$ |

**Table 1: Notation used in this paper.**

### 3.1.1 DefineRelationship

Users invoke the DefineRelationship function to add individuals to a group. The user generates an appropriate attribute secret key using the ABEKeyGen function, encrypts this key using the target user's public key, and stores the encrypted key on her storage service. The target user can retrieve this encrypted key using a process described in Section 3.3, decrypt it, and use it as necessary.

**Algorithm 1** DefineRelationship$(u1, attrs, u2)$

$u1$: $A \leftarrow$ ABEKeyGen$(u1.AMSK, attrs)$
$u1$: $C \leftarrow$ TEncrypt$(u2.TPK, A)$
$u1$: $u1.SS$.put$(H'(u2.TPK), C)$
...
$u2$: $C \leftarrow u1.SS$.get$(H'(u2.TPK))$

**Example Usage:** Alice wants to confer the attribute 'friend' upon Bob. Alice creates $K = Alice.ASK_{\text{'friend'}}$, an ABE key associated with the 'friend' attribute. Alice computes $C =$ TEncrypt$(Bob.TPK, K)$ after obtaining Bob's public key from out-of-band communication with Bob. Alice stores $C$ on her storage service at the location $H'(Bob.TPK)$, where $H'(\cdot)$ is a hash function defined in Section 3.3. Bob retrieves $C$ from Alice's storage service and decrypts it, gaining the ability to decrypt content guarded by the attribute 'friend'. Although any user can retrieve $C$ from its well-known location, only Bob can decrypt it.

### 3.1.2 DefineTransitiveRelationship

The DefineTransitiveRelationship function allows a user Alice to define groups based on a group defined by another user, Bob.

Alice creates a new attribute to describe the new group 'bob-friend' and generates an $ASK_{\text{'bob-friend'}}$ with that attribute. Alice encrypts $ASK_{\text{'bob-friend'}}$ with the access structure ('friend') using Bob's attribute public key and stores the ciphertext on her storage service (Algorithm 2).

Users with the attribute 'friend' in Bob's ABE domain may retrieve and decrypt this key and use it to view content encrypted within Alice's ABE domain. Alice may include a traditional keypair, used for authentication to ACLs, in the ciphertext $C$. We describe how Bob's friends retrieve these keys in Section 3.3.

**Algorithm 2** DefineTransitiveRelationship$(u1, APK,$ access structure $\mathbb{AS}, attrs)$

$u1$: $A \leftarrow$ ABEKeyGen$(u1.APK, attrs)$
$u1$: $C \leftarrow$ ABEEncrypt$(APK, A, \mathbb{AS})$
$u1$: $u1.SS$.put$(H'(\mathbb{AS}, APK), C)$

**Example Usage:** Alice is advertising a party on an OSN and wants to invite Bob and any of Bob's friends. Alice discovers that Bob uses the attribute 'friend' to define who his friends are. Alice generates the group identity traditional PKC keypair $(TPK, TSK)$ for authentication, creates the new attribute 'bob-friend', and generates the attribute secret key $A = \text{Alice}.ASK_{\text{'bob-friend'}}$. Alice calculates

$$C = \text{ABEEncrypt}(Bob.APK, [A, (TPK, TSK)], \text{'friend'})$$

and stores it on her storage service at $H'(\text{'friend'}, Bob.APK)$. Alice also performs AssignRightsToGroup to generate group identity keys and instruct the application providing the event advertising service that $TPK$ can be used to authenticate RSVPs. Bob sends to each of his friends a link to the application that directs them to Alice's event. Bob's friends cannot initially view the data, so they get $C$, decrypt it, and view the event. They then get the group identity key, which allows them to authenticate and RSVP to the event.

### 3.1.3 AssignRightsToIdentity

Resource owners use AssignRightsToIdentity to provide other users specific rights to named resources. An example of such a right would be the ability to store data on another user's storage service; we describe other resources and uses in Section 4.

To assign rights, the user instructs the resource's home to add a (*public key, set of rights*) pair to the resource's ACL. If the public key was already in the ACL, then the rights are changed to those specified in the new rights set (Algorithm 3).

**Algorithm 3** AssignRightsToIdentity$(u1, rights,$ $TPK$, resource $r$, owner $o)$

$u1$: $o$.chACL$(r, TPK, rights)$

User $u2$ who possesses $TSK$ may exercise the named rights on the resource by authenticating to the resource's home node using $TSK$.

**Example Usage:** Alice wants to give Bob the ability to put data on her storage service. Alice instructs her storage service to create a new ACL rule based on $Bob.TPK$ that allows write access. Bob later calls the put function on the location $L$ with the world readable data $m$. Alice's storage service issues a nonce $n$, and Bob replies with TSign$(Bob.TSK, [n, \text{"}write(L, m)\text{"}])$. Alice's storage service verifies the signature against $Bob.TPK$, authenticating Bob's write according to Alice's access policy.

### 3.1.4 AssignRightsToGroup

The AssignRightsToGroup function allows a user Alice to provide resource access to a group $G$ rather than to an individual. The group is specified using attributes defined in Alice's ABE domain.

First, Alice creates a new $(TPK, TSK)$ pair specifically for $G$. Alice ABE-encrypts this keypair with an access struc-

ture that identifies members of $G$. Alice stores the resulting ciphertext on her storage service. This pair of PKC keys becomes the group identity and Alice can assign rights according to AssignRightsToIdentity. The pseudocode is presented in Algorithm 4.

**Algorithm 4** AssignRightsToGroup($u1, rights$, access structure $\mathbb{AS}$, resource $r$, owner $o$)

$u1$: $(TPK, TSK) \leftarrow$ TKeyGen()
$u1$: $C \leftarrow$ ABEEncrypt($u1.APK, (TPK, TSK), \mathbb{AS}$)
$u1$: $u1.SS$.put($H'(\mathbb{AS}, APK), C$)
$u1$: AssignRightsToIdentity($u1, rights, TPK, r, o$)

**Example Usage:** Alice wants to give her friends and her family the ability to put data on her storage service. Alice defines the group $G$ as the users who have 'friend' or 'family' in their ASK in Alice's ABE domain. Alice creates $K = (TPK_G, TSK_G)$, and stores

$$C = \text{ABEEncrypt}(Alice.APK, K, (\text{'friend' or 'family'}))$$

on her storage service. Anyone who possesses either of these attribute keys can retrieve $C$, decrypt it with their $ASK$, and use $TSK_G$ to authenticate to store data on the storage service as described in AssignRightsToIdentity.

## 3.2 Revocation of Group Membership

Removing a group member requires re-keying: all remaining group members must be given a new key. Data encrypted with the old key remains visible to the revoked member. The nominal overhead is linear in the number of group members but can be reduced [37].

An ABE message can be encrypted with an access structure that specifies an inequality ("keyYear $< 2009$"), and the message can be decrypted only if a user possesses a key that satisfies the access structure. This facility can be used to provide keys to new group members such that they cannot decrypt old messages sent to the group.

## 3.3 Publishing and Retrieving Data

Private user data in Persona is always encrypted with a symmetric key.[1] The symmetric key is encrypted with an ABE key corresponding to the group that is allowed to read this data. The group is specified by an access structure as described in Section 2.3. This two phase encryption allows data to be encrypted to groups; reuse of the symmetric key allows Persona to minimize expensive ABE operations.

Users put (encrypted) data onto their storage service and use applications to publish references to their data. Data references have the following format:

$$\langle \text{tag, storage service, key-tag, key-store} \rangle$$

The tag and storage service specify how to retrieve the encrypted data item, and the key-tag and key-store specify how to obtain a decryption key.

Users read data by retrieving both the item and the key. Suppose item $i$ is encrypted with symmetric key $s$. If user $u_1$ wants to read $i$ and $u_1$'s local cache or own storage service does not contain $s$, $u_1$ can retrieve the ABE-encrypted $s$ using the key-tag and key-store information in the reference. $s$ is encrypted under the access structure $\mathbb{AS}$ in the ABE domain defined by $APK$ ($u_1$ can infer both from the encrypted key). $u_1$ tries to decrypt $s$ using its ABE secret key, and if successful, decrypts $i$ using $s$. $u_1$ stores $s$, encrypted with their own public key, on their own storage service for future use. The encrypted key is stored at $H(\mathbb{AS}, APK)$, where $H(\cdot)$ is a hash function. If $s$ is instead encrypted with traditional public key $TPK$, $u_1$ stores the encrypted $s$ at $H(TPK)$.

Suppose user $u_2$ wants to encrypt a message for a set of users specified by access structure $\mathbb{AS}$ in the ABE domain with public key $APK$. The domain may belong to $u_2$ or to some other user; $u_2$ only needs to know the public parameters for this domain in order to encrypt.

$u_2$ looks for a symmetric key for this group by invoking $u_2.SS$.get($H(\mathbb{AS}, APK)$). Such a key would exist if $u_2$ had previously encrypted or decrypted messages for this group. If the retrieval succeeds and the encrypted symmetric key is found, $u_2$ decrypts it using his own public key and obtains the symmetric key $s$.

If the retrieve fails, $u_2$ constructs a new symmetric key $s$, encrypts it with his own PKC public key and stores it in $u_2.SS$ under the tag $H(\mathbb{AS}, APK)$. $u_2$ further encrypts $s$ using ABEEncrypt with access structure $\mathbb{AS}$ and $APK$ and stores this ABE-encrypted symmetric key on $u_2.SS$ with the tag $H'(\mathbb{AS}, APK)$. $H'$ is a hash function different from $H$. By construction, the ABE-encrypted key can be decrypted exactly by those users who belong to the group to which the message is encrypted. This group may not include $u_2$. If $u_2$ wishes to encrypt $s$ with traditional PKC instead of ABE, $u_2$ encrypts with public key $TPK$ and stores the encrypted key at $H'(TPK)$.

Finally, $u_2$ encrypts the message using $s$ and stores it using tag $M$. $u_2$ can then publish a reference to this item of the form:

$$\langle M, u_2.SS, H'(\mathbb{AS}, APK), u_2.SS \rangle$$

Other users resolve the reference by invoking $u_2.SS$.get($M$) which will retrieve the original message encrypted with $s$.

In this example, $u_2$ obtained the decryption key from his own storage service (or created a new key and put it on his own storage service). In general, however, $u_2$ may already know a different key for this group (for example, one that was used by a different user to encrypt to the same group) that is stored on some other storage service. Instead of creating his own key, $u_2$ may choose to refer to this pre-existing key instead.

# 4. APPLICATIONS

Persona users interact using applications. Even core functions of current OSNs, including the Facebook Wall or Profile, exist in Persona as applications. In this section, we describe how applications use the group key and resource management operations of Section 3.

Persona applications export a set of functions (an API) and a set of resources over which those functions operate. When there are resources, such as file stores or documents, two functions are expected in the API. First, register allocates a resource for a principal (to create a Wall, for example). Registration with an application returns a reference to the newly-allocated resource to the client. Second, chACL allows the owning principal to define access restrictions via ACLs: for a given resource and a given principal, permit an

[1] Users may store public data in plain-text to reduce overhead.

operation. Applications will support further operations, as we describe below, starting with the basic storage service.

## 4.1 Storage Service

Storage is a basic Persona application that enables users to store personal data, make it available to others who request it, and sublet access to storage for applications to use for per-user metadata. A user trusts a storage service to reliably store data, provide it upon request, and protect it from overwrite or deletion by unauthorized users. A user does not trust a storage service to keep data confidential, relying instead on encryption to guard private information.

The storage service exports both get and put functions. The storage application returns data whenever the get is invoked with a valid tag. The invoking principal is not authenticated or validated, since the expectation is that data is protected via encryption.

The put function requires the invoking principal $n$ to authenticate to the storage application. When $n$ wants to put data, she presents her public key $K$ and the store identifier $s$ to the storage application. The storage application ensures that $(K, \mathsf{put})$ exists in the resource ACL corresponding to $s$, and authenticates $n$ using a challenge-response protocol. $n$ may write into $s$ if the authentication succeeds.

Applications must store the metadata they have constructed. They can provide their own storage or use a storage service. If the application provides its own storage resource, the application returns a handle to the resource when a user registers with the application. The user can then call AssignRightsToIdentity to give other users access to the application's storage resource.

The user can instead provide the storage resource to the application and invoke:

$$\mathsf{AssignRightsToIdentity}(user, \texttt{write}, App.TPK, c, user.SS)$$

where $c$ is a storage resource on $user.SS$, to allow the application to write onto the user's storage server. The user now registers with the application, passing it the storage resource $c$ in which to store the metadata:

$$R \leftarrow App.register(user.TPK, c)$$

In turn, the application returns a reference ($R$) to the resource corresponding to the application instance.

To prevent an attack in which another user $u_2$ pretends to own $c$, the registering user must prove that he owns $c$. He does this by writing a nonce provided by the application into $c$. The application ensures the nonce is present before writing.

## 4.2 Collaborative Data

The predominant method of sharing data in OSNs is via collaborative multi-reader/writer applications. For instance, the quintessential Facebook application, the Wall, is a per-user forum that features posts and comments from the user and his friends, the Facebook Photos application stores comments and tags for each picture and displays them to friends, the MySpace comments section allows friends to write to a user's page and read others' comments, and each photograph posted to Flickr has a page where members of the Flickr community can comment on photographs. Instead of re-implementing each OSN application in Persona, we present a generic multi-reader multi-writer application named Doc. Doc can be used as a template for implementing a variety of OSN applications, as we describe in Sections 4.2.1–4.2.4.

Doc is organized around a document shared between collaborating users. Users register with the Doc application and create a new *Page*. The application associates a resource with this Page, and allows the user to provide read or write access to other users (or groups). The Page metadata contains references to encrypted data; the application is responsible for formatting this data for display. Users who are allowed to write to the Page contact the application with data references, and Doc updates the Page appropriately. The Page can be stored by the application or on a storage server specified by the original user (in which case the user has to provide the Doc with write access to the Page stored on the storage server). We describe these steps next.

**Reading the Page.** To allow Bob to read content in her Page, Alice must give Bob appropriate keys and a reference to her Doc. In particular, Alice must provide an attribute secret key *ASK* that will allow him to decrypt (some subset of) the data in the Page. Alice decides which attributes Bob should get and calls

$$\mathsf{DefineRelationship}(\text{Alice}, attrs, \text{Bob})$$

to issue an *ASK* to Bob. Obviously, Alice may already have given Bob these attributes, in which case this step can be skipped. In either case, she provides him with a reference to her Page.

Bob can now retrieve the Page metadata, resolve data references, and decrypt (potentially only a subset of) the Page data.

**Writing to the Page.** Alice may want to provide Bob with the ability to write to her Page, where writing is a function exported by the Doc application. She does so by adding Bob's public key to the Page's resource ACL by invoking:

$$\mathsf{AssignRightsToIdentity}(\text{Alice}, \texttt{write}, Bob.TPK, D, Doc)$$

Bob may now write onto the Page. Bob stores (appropriately encrypted) data onto a storage-server and notifies the Doc of a write onto Alice's Page. The Doc application must authenticate Bob and ensure that his public key is in Alice's Page's ACL with the proper right. If the authentication succeeds and Alice has provided Bob the write right, then the Doc application updates the Page metadata (either stored at the application or on a storage server specified by Alice) with the data reference provided by Bob. The interpretation of the Page metadata is application-specific.

Alice may authorize multiple users to write to the same Page. Conflicting updates or concurrent writes are handled by the Doc application, possibly by storing the Page as an append-only log. Users need not encrypt using a single access structure, and may choose any access structure they desire. They may even write onto a Page using an access structure that cannot be decrypted by some of the Page's readers.

In summary, Doc is a general multi-reader/writer template for storing and formatting metadata with references to encrypted content. Doc can easily be tailored to implement many useful OSN applications, as we demonstrate next.

### 4.2.1 Wall using Doc

The Facebook Wall is a multi-user collaborative application that allows a user's friends to read messages, post messages, and comment on posts onto a shared document, called

the user's Wall. Doc can be used to (almost trivially) implement the Wall application. Unlike the Facebook Wall, the Persona Wall is distributed: it allows users to choose where the Wall metadata is stored. All posts and comments are stored on storage servers owned by the poster/commenter. The Wall document itself contains rendering information and references to writes onto the wall. These references must be resolved (i.e., the data fetched from appropriate storage servers) and decrypted before rendering the Wall. End-user applications may intelligently cache data and keys to reduce rendering latency.

#### 4.2.2 Chat and Status Updates over Doc

A chat application can use Doc as the template. A chat session is a shared document to which the chat host invites other users (and provides them write access to the chat Doc). The chat application has to implement auxiliary UI functions (such as an invite notification, and polling for new messages), but the basic structure follows that of a simple Doc onto which users may append messages.

Doc can also be used to implement user-specific status updates. The user creates a status Doc and provides read-only access to other users (or groups) who can periodically read the Doc to receive updates. The reference to the status update Doc may be obfuscated such that unauthorized users are not able to detect changes in status (even if they are not able to decrypt the status message).

#### 4.2.3 News Feed using Doc

The news feed in Facebook collects "stories" from other applications to provide a temporal view of Facebook activity. In Persona, the user provides the news feed with a list of applications that he wants to appear in his feed, and an *APK* and $\mathbb{AS}$ (or perhaps several access structures along with a policy dictating when to use each access structure) with which to encrypt the feed. Only the user may change the list of monitored applications. The news feed application retrieves the metadata from the selected applications and parses it to create a history of changes to the user's applications' metadata. The application writes this history as a user would write a Page; only the news feed may write to this metadata. Viewing the feed consists of viewing the Page. The contents of the Page are visible to anyone that can satisfy $\mathbb{AS}$.

#### 4.2.4 Other Applications

Other popular Facebook applications such as Profiles, Photos, Groups, and Events can be implemented using Doc as well. These applications can be implemented by altering the interpretation and presentation of metadata and tailoring the API to the relevant task. Though Doc is sufficient for many Facebook applications, we consider examples of existing applications that require additional features in the following sections.

### 4.3 Selective Revelation

The user may want to share some personal data with an application. One such example is an application that allows users to search for others. Alice can choose exactly the information by which other users can find her by only sharing that data with a Search application. Another example is the *Where I've Been* Facebook application [36]. Users enter a list of countries or cities that they have lived in, visited, or want to visit, and the application shows a map with these locations highlighted. Users can also compare maps with another user to see which locations they have in common.

In order to permit applications that post-process personal data, we allow them to decrypt certain data by giving them an *ASK*. Alice encrypts a list of cities she has visited with the access structure ('classmates' or 'where-ive-been'). She generates an *ASK* and encrypts it with the Where I've Been application's *TPK*:

DefineRelationship(Alice, 'where-ive-been', Where I've Been)

When she registers to use the application, she gives it a reference to the encrypted key. The application retrieves the key and can now decrypt and parse Alice's list of cities to produce the highlighted map. This general approach of selectively revealing user data to applications has been discussed earlier in [20].

Application functionality that can be implemented without revealing personal information is surprisingly broad; however, in some cases, the application must compute transforms over the user's data. This is the case for the Where I've Been application, especially when it has to compare the locations of multiple users. We return to the general problem of structuring private applications and the tussle between application functionality and user privacy in Section 8.

### 4.4 Applications that use the social graph

The graph of social connections between Persona users is not public. It is realized only in the collections of public keys of friends a user stores, and given meaning only through the assignment of attributes using DefineRelationship. This obscurity of friend links frustrates applications such as those that analyze the graph of connections to help connect with more friends (People You May Know) or to visualize interconnections between friends (the Friend Wheel).

To enable these applications, users have two options. A user may publish social links to each application using selective revelation or by directly uploading a set of relationships. Alternatively, a single, somewhat trusted social link application might provide access to other applications.

Published edges in the social graph are protected just as other data in Persona: encrypted to be hidden from arbitrary users and applications, but exported to chosen users and useful applications that may access only what they require.

### 4.5 Inherently private applications

Persona allows for potential applications which are not realistic on OSNs without privacy. For instance, a user might want to have a Medical Record application where she stores her medical data. She might not want her employer or her friends to see her data, but she would want to share it with her doctor. She may even have many doctors, and it may be helpful for them to collaborate in a central location. There is no technical difference between this application and Doc. However, these applications are uniquely available on Persona because they operate on sensitive private data.

## 5. IMPLEMENTATION

Our Persona implementation consists of two Persona applications (a storage service and a customizable Doc appli-

cation) and a browser extension for viewing encrypted pages and managing keys.

## 5.1 Storage Service Application

Our Persona storage service application is an XML RPC server using PHP and Apache with a MySQL database backend. The service implements the storage API described in Section 4.1.

## 5.2 Doc Application

We have implemented a Doc application (Section 4.2) in PHP with a MySQL backend for storing metadata. Using the Doc as the base, we implemented Profile and Wall applications.

Our Profile application presents an interface for the user to put data onto her profile and read others' profiles. The profile metadata (stored by the Profile application in a MySQL database) consists of references to encrypted profile data items. The Profile application allows only the registered user to write onto the Doc Page.

Our Wall application is identical in structure to the Profile, but allows other users to write onto the Doc as well. The Wall application allows users to post new items and reply to existing items. The Wall application constructs the Wall Doc metadata file threading posts and replies. As with all applications, the posts and references themselves are stored on other storage services, and the Wall application operates using item references only.

## 5.3 Browser Extension

Users interact with Persona using a Firefox extension. The extension uses the XPCOM framework in the Mozilla Build Environment to access the OpenSSL and cpabe [2] libraries for cryptographic operations. The extension allows users to register with applications, encrypt data to groups, resolve data references, decrypt data using appropriate keys, and facilitate out-of-band public-key exchange.

The browser extension is a trusted component in Persona; it is, in fact, the only one. The extension implements a secure keystore, to which users upload their private and public keys. The extension is given a list of public keys corresponding to the user's contacts. These keys are also stored (encrypted with the user's public key) on a storage service. When a user uses a new browser, the extension is initialized with the user's private key and a reference to the user's permanent keystore. The extension then downloads all of the other keys from the storage service.

When an encrypted Persona page is loaded, the extension processes the elements on the page and replaces them inline if necessary. There are two main types of replacement: resolution of data references and replacement of special tags.

**Data reference resolution.** The extension parses item references, fetches the items from storage services, decrypts the items, and verifies any signatures on those items. In our implementation, all data is signed by the creator and verified if the signer's key is known. Data resolution is recursive: encrypted data may contain references to more encrypted data.

Our extension uses an XML-RPC javascript library capable of sending asynchronous RPCs. During page processing, all data items are fetched asynchronously using `XMLHttpRequest`. If the items are encrypted with an unknown key, the keys are also fetched asynchronously. Once all keys and data items have been fetched, the extension sequentially decrypts (and verifies) each item, and replaces the references with the decrypted text. We are currently extending our implementation to decrypt items as they arrive rather than waiting for all fetches to complete.

**Replacement of special tags.** Persona users may not want to share their list of contacts (to be precise, their public keys) with applications. Instead, this list is kept encrypted with the user's public key on a storage service, which the extension downloads upon initialization. The extension recognizes a "friend-form" tag sent by an application, and replaces this with a drop-down box containing a list of the user's contacts. This facility is used in our Profile application to allow a user to view their contacts' profiles.

The extension allows users to encrypt data to groups. It replaces embedded forms with a text box into which the user can enter private data. When the submit button is pressed, the extension prompts the user for a policy under which to encrypt the data, performs the encryption (constructing and publishing symmetric keys as necessary), puts the encrypted data on the user's storage service, and replaces the form data with a reference to the encrypted data item.

**Caching.** To reduce latency, the extension caches various keys and contact information. This includes keys the user has created: an RSA public key (137 bytes for 1024-bit moduli), RSA private key (680 bytes), *APK* (888 bytes), and *AMSK* (156 bytes). For each friend, the extension caches their storage service information, RSA public key, and *APK*. The extension also stores the *ASK* (the size varies: 407 bytes for one attribute and 266 bytes for each additional attribute) created for that friend along with the attributes associated with the *ASK*. For each policy that the user is a part of, whether it is created by the user or a friend, the extension caches the RSA keypair and the symmetric key.

This caching and recycling of symmetric keys allows the extension to pay the cost of an ABE decryption only when it encounters an item encrypted using a new key reference. This will occur when the encryption uses a new policy (corresponding to a new group) or an existing policy to which a user has encrypted with a new symmetric key. The latter might occur if the encrypting user is not part of the group and is unable to decipher existing symmetric keys for that policy. The common operation of the extension does not require expensive ABE operations.

## 5.4 Integrating Persona with Facebook

Current deployments of OSNs underline their undeniable popularity. It is not realistic to assume that Persona (or some other privacy-enabled network) will replace existing OSNs. Instead, we expect users to migrate personal information onto private networks, while continuing to use existing OSNs for public data.

We have designed Persona to inter-operate with existing OSNs, and our prototype integrates with Facebook. Persona applications are accessible as Facebook applications and can interact with Facebook's API, providing privacy-enabled applications through the familiar Facebook interface. Conversely, existing Facebook applications can be made Persona-aware on a per-application basis. Users protect their private data by storing it on Persona storage services rather than on Facebook; only fellow Persona users will be able to access the data, and only if they are given the necessary keys and access rights.

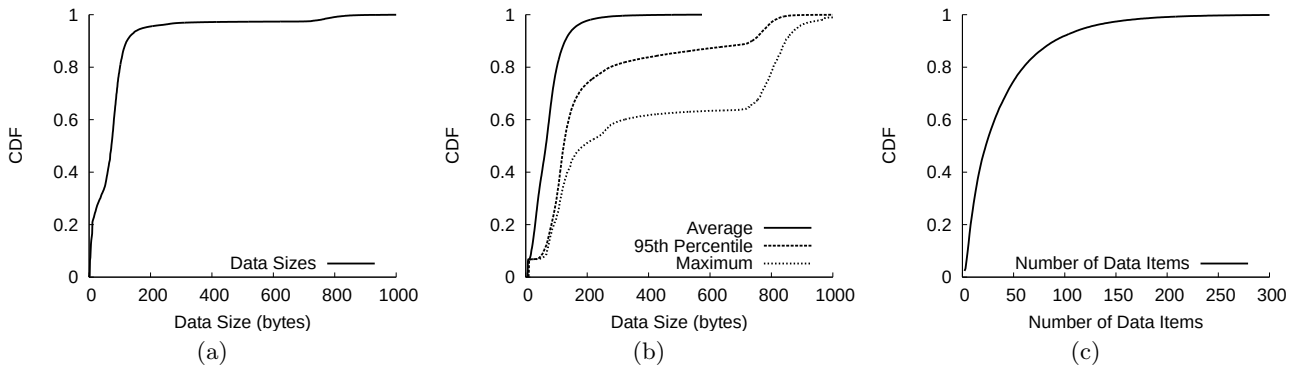


**Figure 1: (a) CDF of the size of Facebook profile data items. (b) CDF of the maximum, 95th percentile, and average of the size of data items per Facebook profile. (c) CDF of the number of data items on Facebook profiles.**

**Using Persona applications within Facebook.** Users log-in to Persona by authenticating to the browser extension (which then decrypts and encrypts data transparently), and then log-in to Facebook as normal. A Facebook-aware Persona application is akin to any third-party Facebook application, and can be selected for use as any other Facebook application. Unlike other applications, Persona applications use markup that is interpreted by the Persona browser extension, and are aware of data references.

Traditional Facebook applications may use the Facebook API to communicate to users by sending notifications, displaying items on the Facebook wall, and sending application invitations. The same facilities are available to Persona applications. We have implemented an abstract OSN interface that Persona applications use to access OSN APIs. While our design is general, our current implementation has only been tested with Facebook. Our Doc-based applications are accessible via Facebook as Facebook applications.

**Using Facebook applications on Persona.** Once users begin to use Persona, existing Facebook applications may want to provide Persona users with the ability to store private data. Minimally, each application has to be ported to operate using Persona data references, though some applications that transform user data may require a complete rewrite. We discuss application porting in Section 8.

# 6. EVALUATION

In this section, we quantify the processing and storage requirements of Persona and measure the time to render Persona-encrypted web pages.

The key parameters of our evaluation are the sizes and number of distinct data elements that might be stored on a single Persona page. Each distinct element represents a request to a storage server and may, if the policy and associated key are unknown, also imply a request for a group key and its decryption with ABE. This process represents the performance cost of Persona. We estimate these parameters using Facebook as a model, combining real user profiles from Facebook with observations of application-provided limits on the number of items per page.

User profiles can contain hundreds of data items. We use profile data in our evaluation because it exposes the worst case performance of Persona, where users must fetch and decrypt many individually encrypted data items. Our data is from a crawl of Facebook profiles gathered in January, 2009. The crawl contains the HTML of the profile pages of 90,269 users in the New Orleans network; of those pages, 65,324 pages contain visible profiles, and 39 pages had miscellaneous errors that left them unusable.

We parse these Facebook profiles into data items that could be individually encrypted. First, we parse the document based on fields such as Name, Birthday, Activities, Interests, etc. We then decompose fields which contain multiple items separated by commas, bullet points, or line breaks. Under this decomposition, users would be able to, for example, individually encrypt every TV show, book, and movie that they enjoy, if they chose to do so.

Figure 1 (a) shows a CDF of the sizes of all data items and Figure 1 (b) shows a CDF of the maximum, 95th percentile, and average data item sizes on a per-profile basis. These plots show that most of the data items are small, but many pages also have a few large items. We also present a CDF of the number of data items per profile in Figure 1 (c). These figures provide a backdrop for the performance of Persona: our results show that the number of data items on a page determines the page load time.

## 6.1 Desktop Performance

We evaluate our Persona implementation on a desktop computer using a 2.00 GHz processor and 2 GB of RAM. The desktop, storage service, and application server are connected through a router which introduces an artificial delay, chosen uniformly between 65ms and 85ms, on each packet. These values reflect high latencies observed by King [15] and represent a case where the storage service is far away from the user.

We use two experiment scenarios. The first, termed *cool*, represents Persona in its initial state, when group symmetric keys must be retrieved from a storage service and decrypted. The second, termed *warm*, represents Persona usage in the steady state, when all symmetric keys associated with groups have been cached. We repeat the *cool* experiment scenario three times, varying the number of user-defined groups between 1, 10, and 100. We run only one *warm* experiment scenario since no key fetches and no ABE decryptions are needed. In each data set, we randomly assign each data item to one of the user-defined groups.

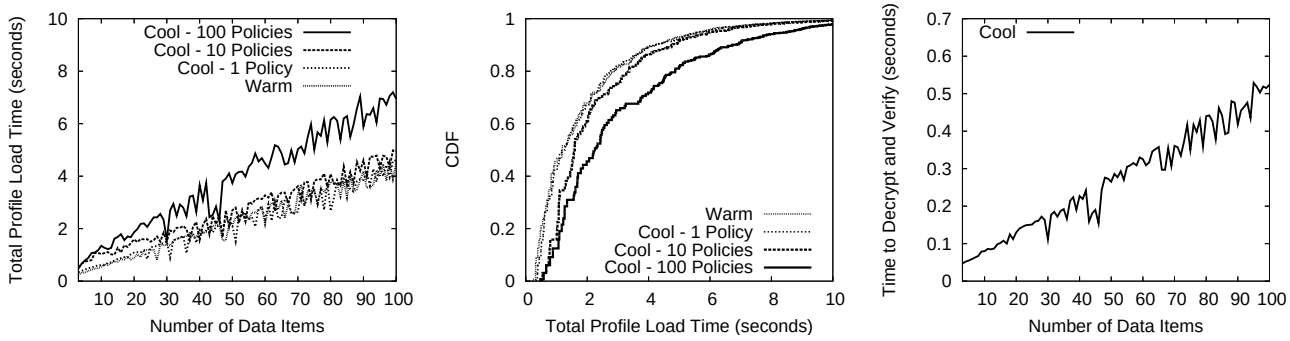


**Figure 2: (a) Total time needed, in seconds, to present Facebook profiles composed of encrypted data items. (b) CDF of total time to load Facebook profiles. (c) Total time needed, in seconds, to decrypt encrypted data items in Facebook profiles in the *cool* data set with 100 groups. Note the difference in scale from (a).**

For each Facebook profile, we first encrypt and store each of the data items in Persona. We then retrieve a page that contains references to all of these data items. In the *cool* data set, we asynchronously fetch the keys needed to decrypt all of the items in the page. In both *cool* and *warm*, we also asynchronously fetch the encrypted data items themselves. Once all keys and data items have been fetched, we decrypt the data items on the page, verify their signatures, and re-render the page. For efficiency, rather than evaluating every profile, we evaluate a profile page drawn randomly from the set of all pages that have $x$ items, for all values $x$ for which there is a profile with $x$ items.

**Page load time.** Page load times increase linearly with the number of elements. Figure 2 (a) shows how long it takes to download, decrypt, and display the profile page for each of our experiments, as a function of the number of data items on the page. We extrapolate the distribution of page load times per Facebook profile in Figure 2 (b). The median page load time is 2.3 seconds and the maximum is 13.7 seconds. Most pages consist of a few, small entries, so most are loaded quickly. The *cool* data sets are comparable to the *warm* data set, indicating that retrieving keys is not too expensive. These times may also represent a worst case; if users aggregate their data more coarsely there will be fewer data items, requiring fewer fetches and thus fewer round-trip times. Another possible improvement would be to cache commonly retrieved data items, but we have not performed this optimization.

**Encrypted data size.** We show how much larger the encrypted data is for individual data items in Figure 3 and for entire profile pages in Figure 4. There is a substantial increase in the size of the stored data, and this will affect both the storage capacity of the storage services and the network resources required to transfer data. The storage services are inherently distributed, so they should be able to scale to support the needs of the system.

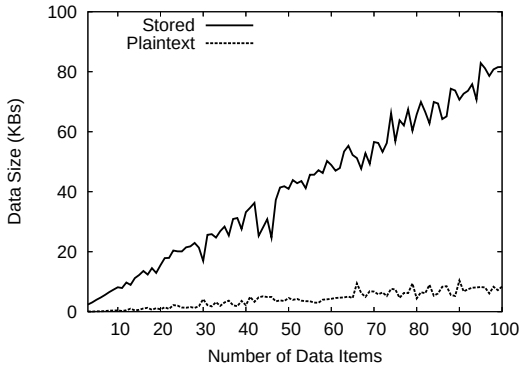


**Figure 3: Total size of plaintext and stored (ciphertext and signature) data for Facebook profile pages by number of data items on the page.**

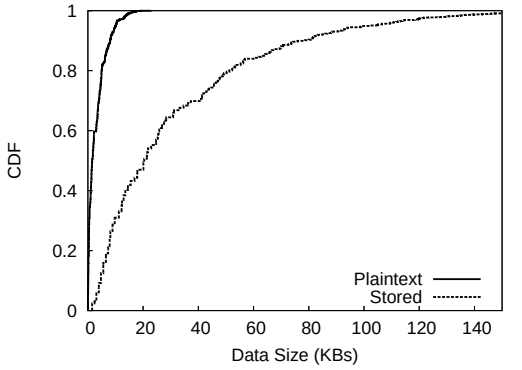


**Figure 4: CDF of total size of plaintext and stored (ciphertext and signature) data for Facebook profile pages.**

## 6.2 Mobile Device ABE Performance

Mobile devices are increasingly used for limited access to OSNs. MySpace, Facebook, and LinkedIn [24] all have iPhone applications, and there are many twitter and instant messaging clients. Persona, to provide a substitute, must also be realizable on mobile devices. Enabling mobile devices with Persona-based security would enable users to exchange their current locations with friends but not third parties, enabling functionality similar to that of Loopt [25] without trusting the service provider.

The requirements for encryption performance of mobile OSN clients are a bit different from their desktop counterparts. Because of their smaller screens and often slower net-

work connections, the requirements for decryption are less demanding: when only a few messages may be retrieved or displayed at a time, decrypting only a few items is necessary. Conversely, mobile devices tend to have limited computation power and limited battery life, so the operations themselves should be reasonably interactive.

We cross-compiled the cpabe [2] libraries and their dependencies (pbc [27], gmp, glib, openssl, gettext, libiconv, and libintl) for the iPhone SDK 2.2.1 [4].[2] Some of these libraries (e.g., libcrypto from OpenSSL) are present on the device but not included in the official SDK. Cryptographic operations supported on the device may be implemented using hardware acceleration when applications are written using Apple's defined APIs; writing directly to the OpenSSL library forgoes these potential advantages. In other words, our benchmark is sufficient to show that ABE is practical on a widespread mobile device, but not intended to compare ABE or AES performance from one device to another.

On a first-generation iPhone (620MHz ARM), decryption of ABE encrypted text fragments smaller than 1KB takes approximately 0.254 seconds. This value is the average time to decrypt 40 randomly-generated messages of 40 different sizes drawn uniformly at random from 0 to 4095 bytes of 5 different access structures having one to five attributes. Message size and access structure have little effect: the message itself is encrypted using AES-128, and the access structure appears to have a greater effect on the time to encrypt than to decrypt. Encryption times average 0.926 seconds with one attribute (an average of 25 messages of 25 sizes; some of this time is likely consumed by AES-128 key generation) and 0.43 seconds for each additional attribute.

We believe that the 0.254 second object decryption time compares favorably to the typical RTT of cellular data systems (Lee [22] reported a 417ms average RTT for 1x EV-DO) and does not preclude a mobile Persona.

# 7. RELATED WORK

We present related work dealing with studies of OSN privacy, systems implementing privacy on OSNs, access control, ABE, and systems built on ABE.

**OSN studies.** Several works examine the characteristics and recent growth of OSNs [12, 17, 19, 28, 29]. Krishnamurthy and Willis [20] study how OSNs share users' personal data with third parties such as applications and advertisers. They note that Facebook places no restrictions on the data that is shared with external applications. Advertisers use personal data, as well as information acquired through cookies, to serve targeted ads.

Prior research has characterized privacy problems with OSNs. Acquisti and Gross [1, 13] show that Facebook users at CMU often share more data than they are aware of. Lam et al. [21] study a Taiwanese OSN to show that users' annotations compromise the privacy of others. Ahern et al. [3] study Flickr to see how location information is leaked through users' photographs. Several studies [16, 18, 38] exploit the friend graph to infer characteristics about users. Persona resolves these issues by allowing users to precisely express the policies under which their data, including friend information, is encrypted and stored.

[2]Patches to enable cross-compilation of these libraries using Apple's gcc compiler are available at http://www.cs.umd.edu/projects/persona

**OSN privacy systems.** The research community has recognized the problem of privacy in OSNs and proposed several solutions which build on top of existing OSNs. NOYB [14] hides an OSN user's personal data by swapping it with data "atoms" of other OSN users. NOYB provides a way to map these atoms to their original contents. flyByNight [26] is a Facebook application that facilitates secure one-to-one and one-to-many messages between users. Finally, Lockr [34] uses ACLs based on social attestations of the relationship between two users, similar to how Persona distributes *ASK*s to users that satisfy certain attributes. Persona and Lockr both use XML-based formats to transfer privacy-protecting structures.

**Access control and ABE.** In Persona, the attributes a user has determines what data they can access. This resembles role-based access control [11] and attribute-based access control, which bases authorization decisions on the attributes assigned to users [6, 40]. Attribute based encryption (ABE) was introduced as an application of a type of identity based encryption (IBE) called fuzzy IBE [32]. Unlike early ABE schemes, CP-ABE [5], which Persona uses, binds ciphertexts to access structures while secret keys contain attributes. Ciphertexts can be decrypted with a key that contains a set of attributes that satisfies the access structure. Multi-authority ABE [7, 23] removes the need for transitive key translations but requires each user to have a globally-unique identifier and the attribute set to be partitioned amongst the users.

Pirretti et al. [31] show how to build a dating social network that only reveals information about a user if their attributes match another user's desired description. Unlike Persona, their system relies on a single authority to generate all secret keys. Traynor et al. [35] introduce a tiered architecture to improve the performance of ABE so that it scales to millions of users.

# 8. DISCUSSION

Our Persona prototype and evaluation demonstrates new functionality and reasonable performance. In this section, we discuss unexplored questions a large-scale deployment will have to confront.

**Factoring applications.** Persona was motivated by the observation that current OSN applications have complete access to user data. Current Persona applications, on the other hand, have no access to user data and must operate entirely using data references. Applications that act on user data must be given selective access as described in Section 4.3. This approach is similar to how others [20] have discussed statically classifying user data in OSNs for application access.

An alternate design is to refactor applications into one piece administered by the application provider (as now), and another piece capable of transforming user data that would be executed on a trusted host (likely, within the user's browser). Existing taint-tracking techniques [33, 39] can be used to guarantee that user-data remains safe. This option relieves the user from thinking about what data should be released to which applications; however, application design and implementation must undergo a substantial change.

**Factored data.** Persona decouples application metadata from encrypted content. This may lead to cases when one is available but not the other. Ideally, data and metadata would share availability, but combining both might lead to

unacceptable performance or violate storage policy (about where data might be stored). A scalable policy-compliant design for a fate-sharing [8] dissemination infrastructure is an open problem.

**Deployment incentives.** OSNs are popular, in part, because they are free. Persona's design requires users to contract with applications, and some applications, such as the storage service, may have little incentive to provide free service. Users may have to pay for this storage or agree to use some other service or applications in exchange for free storage. Other applications—for instance, versions of Doc—may augment the metadata with advertisements, which may provide a sustaining deployment model. As privacy-enhanced OSNs become popular, current OSN providers may choose to incorporate privacy features, in effect supporting the Persona + Facebook model we have implemented.

# 9. CONCLUSION

Privacy controls provided by existing OSNs are not sufficient since they rely on trusting the OSNs with data from which they can profit. We have shown how ABE and traditional public key cryptography can be combined to provide the flexible, user-defined access control needed in OSNs. We have described group-based access policies and the mechanisms needed to provide decryption and authentication by both groups and individuals. We have demonstrated the versatility of these operations in an OSN design called Persona, which provides privacy to users and the facility for creating applications like those that exist in current OSNs.

To prove the feasibility of Persona, we implemented and evaluated Persona on Facebook profile data. Median load times in Persona are 2.3 seconds and the median size of the encrypted profile data is 20.4 KB. We have shown that we can achieve privacy in OSNs with acceptable performance even on mobile devices.

## Acknowledgments

We would like to thank Alan Mislove for providing us with the Facebook profile data used in our evaluation. We would also like to thank Balachander Krishnamurthy, Dov Gordon, Katrina LaCurts, and our anonymous reviewers for their assistance and comments. This work was supported in part by NSF grant CNS-0626629.

# 10. REFERENCES

[1] A. Acquisti and R. Gross. Imagined communities: Awareness, information sharing, and privacy on the facebook. In *PET*, 2006.

[2] Advanced crypto software collection. http://acsc.csl.sri.com/cpabe/.

[3] S. Ahern, *et al.* Over-exposed?: privacy patterns and considerations in online and mobile photo sharing. In *Human Factors in Computing Systems*, 2007.

[4] Apple iPhone SDK. http://developer.apple.com/iphone/.

[5] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *Security and Privacy*, 2007.

[6] P. A. Bonatti and P. Samarati. A uniform framework for regulating service access and information release on the web. *Journal of Computer Security*, 2002.

[7] M. Chase. Multi-authority attribute based encryption. In *TCC*, 2007.

[8] D. Clark. The design philosophy of the darpa internet protocols. In *SIGCOMM*, 1988.

[9] Facebook statement of rights and responsibilities. http://www.facebook.com/press/info.php?statistics# /terms.php?ref=pf.

[10] Facebook statistics. http://www.facebook.com/press/info.php?statistics.

[11] D. F. Ferraiolo and D. R. Kuhn. Role-based access controls. In *National Computer Security Conference*, 1992.

[12] M. Gjoka, M. Sirivianos, A. Markopoulou, and X. Yang. Poking facebook: Characterization of OSN applications. In *WOSN*, 2008.

[13] R. Gross and A. Acquisti. Information revelation and privacy in online social networks (the facebook case). In *WPES*, 2005.

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

[15] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *IMC*, 2002.

[16] J. He, W. W. Chu, and Z. V. Liu. Inferring privacy information from social networks. In *ISI*, 2006.

[17] J. Kleinberg. Challenges in social network data: Processes, privacy and paradoxes. In *KDD*, 2007. Invited talk.

[18] A. Korolova, R. Motwani, S. U. Nabar, and Y. Xu. Link privacy in social networks. In *Information and Knowledge Mining (CIKM)*, 2008.

[19] B. Krishnamurthy. A measure of online social networks. In *COMSNETS*, 2009.

[20] B. Krishnamurthy and C. E. Wills. Characterizing privacy in online social networks. In *WOSN*, 2008.

[21] I.-F. Lam, K.-T. Chen, and L.-J. Chen. Involuntary information leakage in social network services. In *IWSEC*, 2008.

[22] Y. Lee. Measured TCP performance in CDMA 1x EV-DO network. In *PAM*, 2006.

[23] H. Lin, Z. Cao, X. Liang, and J. Shao. Secure threshold multi authority attribute based encryption without a central authority. In *INDOCRYPT*, 2008.

[24] Linkedin. http://www.linkedin.com/.

[25] Loopt. http://www.loopt.com.

[26] M. M. Lucas and N. Borisov. flybynight: Mitigating the privacy risks of social networking. In *WPES*, 2008.

[27] B. Lynn. *On the implementation of pairing-based cryptosystems*. Ph.D. thesis, Stanford, 2008.

[28] A. Mislove, *et al.* Measurement and analysis of online social networks. In *IMC*, 2007.

[29] A. Mislove, *et al.* Growth of the flickr social network. In *WOSN*, 2008.

[30] D. Naor, M. Naor, and J. B. Lotspiech. Revocation and tracing schemes for stateless receivers. In *CRYPTO*, 2001.

[31] M. Pirretti, P. Traynor, P. McDaniel, and B. Waters. Secure attribute-based systems. In *ACM CCS*, 2006.

[32] A. Sahai and B. Waters. Fuzzy identity-based encryption. In *Eurocrypt*, 2005.

[33] U. Shankar, *et al.* Detecting format-string vulnerabilities with type qualifiers. In *USENIX Security*, 2001.

[34] A. Tootoonchian, *et al.* Lockr: Social access control for web 2.0. In *WOSN*, 2008.

[35] P. Traynor, K. Butler, W. Enck, and P. McDaniel. Realizing massive-scale conditional access systems through attribute-based cryptosystems. In *NDSS*, 2008.

[36] Where I've been. http://apps.facebook.com/whereivebeen/.

[37] C. K. Wong, M. Gouda, and S. S. Lam. Secure group communications using key graphs. *SIGCOMM CCR*, 28(4):68–79, 1998.

[38] W. Xu, X. Zhou, and L. Li. Inferring privacy information via social relations. In *ICDEW*, 2008.

[39] H. Yin, *et al.* Capturing system-wide information flow for malware detection and analysis. In *CCS*, 2007.

[40] T. Yu, M. Winslett, and K. E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *Transactions on Information and System Security*, 2003.