Building Confederated Web-based Services with Priv.io

Liang Zhang
College of Computer and Information Science
Northeastern University
Boston, MA
liang@ccs.neu.edu

Alan Mislove
College of Computer and Information Science
Northeastern University
Boston, MA
amislove@ccs.neu.edu

ABSTRACT

With the increasing popularity of Web-based services, users today have access to a broad range of free sites, including social networking, microblogging, and content sharing sites. In order to offer a service for free, service providers typically monetize user content, selling results to third parties such as advertisers. As a result, users have little control over their data or privacy. A number of alternative approaches to architecting today's Web-based services have been proposed, but they suffer from limitations such as relying the creation and installation of additional client-side software, providing insufficient reliability, or imposing an excessive monetary cost on users.

In this paper, we present Priv.io, a new approach to building Web-based services that offers users greater control and privacy over their data. We leverage the fact that today, users can purchase storage, bandwidth, and messaging from cloud providers at fine granularity: In Priv.io, each user provides the resources necessary to support their use of the service using cloud providers such as Amazon Web Services. Users still access the service using a Web browser, all computation is done within users' browsers, and Priv.io provides rich and secure support for third-party applications. An implementation demonstrates that Priv.io works today with unmodified versions of common Web browsers on both desktop and mobile devices, is both practical and feasible, and is cheap enough for the vast majority users.

Categories and Subject Descriptors

C.2.4 [**Performance of Systems**]: Distributed Systems—*Distributed applications*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

Keywords

Web; privacy; online social networks; confederated services; Web browsers

COSN'13, October 7–8, 2013, Boston, Massachusetts, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2084-9/13/10 ...$15.00.
http://dx.doi.org/10.1145/2512938.2512943.

1. INTRODUCTION

Users today have access to a broad range of free Web-based services (e.g., online social networks such as Facebook, microblogging services such as Twitter, content sharing sites such as Flickr). All of these services operate under a similar model: Users entrust the service provider with their personal information and content (e.g., their comments, photos, political and religious views, sexual orientation, occupations, identities of friends). In return, the service provider makes their service available for free by monetizing the user-provided information and selling the results to third parties (e.g., advertisers). Even though users are often provided with privacy controls on these sites, these controls generally only affect flow of information to other users or third-party applications; users today have no option of making their data private *from* the service provider. This model also makes it difficult for users to retrieve all of their data from the provider (e.g., if the provider closes the service [31, 32]) or remove their data entirely.

Researchers have investigated a number of approaches that provide users with greater control and privacy in such services, ranging from encrypting data uploaded to the provider [22,38,46] to dividing data between provider-hosted and user-hosted servers [10, 43] to implementing a fully decentralized system [11, 15, 18]. Unfortunately, none of these approaches have enjoyed widespread adoption, as they suffer from one or more of three general limitations:

- **Accessibility** Most proposals require users to install dedicated client software, such as desktop applications or browser plugins. As users typically access services from a variety of devices, these solutions require significant effort of the user (who has to install the software) and the developer (who has to build and maintain clients for various devices).
- **Reliability** Systems that rely on hosting content on end-user machines [11, 15, 18], home routers [30], or smartphones [45] maintain availability via replication. Unfortunately, such systems are known for suffering from fundamental reliability tradeoffs in dynamic environments [7].
- **Cost** Systems that require users to rent their own server from a cloud provider [43] or pay for subscription of the service [5] are likely to be too expensive for most users.

In this paper, we present Priv.io, an alternate approach to implementing Web-based services that provides users with control over their data, ensures privacy, and avoids the limitations of practicality, reliability, and cost. In Priv.io, each user provides resources necessary to support their use of the service by purchasing computing resources (storage, bandwidth, and messaging) from cloud providers such as Amazon Web Services or Windows Azure. Unfortunately, having users purchase *computation* from cloud providers is not practical in Priv.io: at the finest granularity, users still must purchase an entire virtual machine for an hour, and having an always-on server is too expensive for most users. Instead, Priv.io is built entirely in JavaScript, and all computation[1] is done within the users' Web browsers while they visit the Priv.io Web site. Priv.io works with unmodified versions of common Web browsers such as Safari, Chrome, Firefox, and Internet Explorer, as well as browsers on mobile OSes including Android and iOS.

The result is a *confederated*[2] service, where each user retains control over his or her own data. We demonstrate that services similar to Facebook, Twitter, and Flickr can be implemented in a confederated manner with very low monetary costs for most users. Thus, Priv.io provides users with an alternative to today's model of paying for Web-based services by giving up their privacy.

Priv.io provides strong guarantees of user privacy. Priv.io uses attribute-based encryption [9, 10] to encrypt all content stored on the cloud provider; this encryption is implemented in JavaScript within the user's browser. Thus, only the users' browsers ever see plaintext content. Priv.io also provides rich support for third-party applications (e.g., Farmville [53]) by providing an API that is implemented within the users' browsers. Priv.io uses browser-based sandboxing to ensure that third-party applications can only access the data that users allow and cannot leak any user information to the application provider or other third-parties.

We evaluate Priv.io using a number of techniques. *First*, we estimate the monetary cost of using Priv.io with traces from real-world content sharing sites; we demonstrate that 99% of users would pay no more than $0.95 per month if services similar to Facebook, Twitter, or Flickr were built using Priv.io. *Second*, we implement a prototype of Priv.io—available at `https://priv.io`—as well as two applications: a Facebook-like news feed application and an instant-messaging application. *Third*, we use microbenchmarks to show that downloading and uploading content is of similar speed to existing services, and that leveraging the users' browsers for computation is both efficient and practical. *Fourth*, we measure the user-perceived performance of Priv.io and demonstrate that Priv.io is practical on desktop browsers today and is likely to be sufficiently fast on mobile devices in the near future.

The remainder of this paper is organized as follows. Section 2 presents a measurement study aimed at estimating the cost of Priv.io to users of a variety of today's Web-based services. Section 3 describes the design of Priv.io and Section 4 details how Priv.io supports third-party applications in a secure manner. Section 5 presents a discussion of some issues that arise when deploying Priv.io. Section 6 presents an evaluation of Priv.io, and Section 7 describes related work. Section 8 concludes.

[1]Of course, computation is also necessary on the cloud provider's side to implement the storage and messaging abstractions. For the purposes of this paper, computation refers to Priv.io and third-party application logic.

[2]We choose the adjective *confederated* rather than *federated*, as members in a confederation retain autonomy and are generally free to leave (e.g., the Articles of Confederation between the original 13 U.S. colonies).

2. OVERVIEW

Recall that our approach is to implement a Web-based service in a confederated manner, by having users provide the resources necessary to support their use of the service via cloud providers such as Amazon Web Services. While cloud providers typically offer bandwidth, storage, and messaging at relatively fine granularity, computation is still sold at a relatively coarse granularity (typically an entire virtual machine for an hour). As a result, even running the smallest of Amazon's EC2 servers (`t1.micro`) would cost a user $14.40 per month [3], not including EBS storage and I/O costs. Moreover, running an entire virtual server is overkill for most users; most of the time, this server would sit idle.

The result is that cloud services can be practically used today to provide storage, bandwidth, and messaging, but not computation. Our insight in Priv.io is to *use the user's Web browser to provide the computation needed* while they use Priv.io. Doing so provides a number of benefits: Using a user's Web browser for computation reduces costs (since users do not need to purchase computation), reduces security concerns (since content is encrypted in the browser, no third-party sees unencrypted content), and is practical (since most cloud providers allow storage and messaging services to be accessed via HTTP). However, only using browser-based computation also presents a few challenges: it results in a system where users are not always online (if a user does not have an browser window open to Priv.io, computation cannot be done on their behalf) and only provides a restricted model of computation (browser JavaScript is sandboxed, and cannot access the local disk or have unfettered access to the networking stack).

2.1 Cost study

Before we describe how we address these challenges, we briefly estimate the cost to users if services such as Facebook were implemented in a confederated manner. In other words, if each user contracted with a cloud provider to pay for their use of services such as Facebook, what would the per-user costs be? We examine this question in the context of social networking sites, microblogging sites, and content sharing sites.

Unfortunately, estimating the per-user cost is not entirely straightforward, as data availability is scarce and the costs of optimizations and overhead are hard to estimate. As a result, our goal is not to deduce the exact costs, but rather, to provide a reasonable estimate.

Social networking: Facebook To estimate the cost of storing and serving Facebook content, we use a collection of 651,539 Facebook profiles[3] from a large regional network [20]. The data includes all wall posts, status updates, photos, and videos uploaded by these users. We assume that photos are 64 KB [8], and that videos have a bitrate of 1.403 Mbps [35].

Unfortunately, our data set does not include how often content is viewed; the most detailed statistics on the viewing

[3]At the time of collection (2009), Facebook profiles were by default visible to all members of the same regional network.

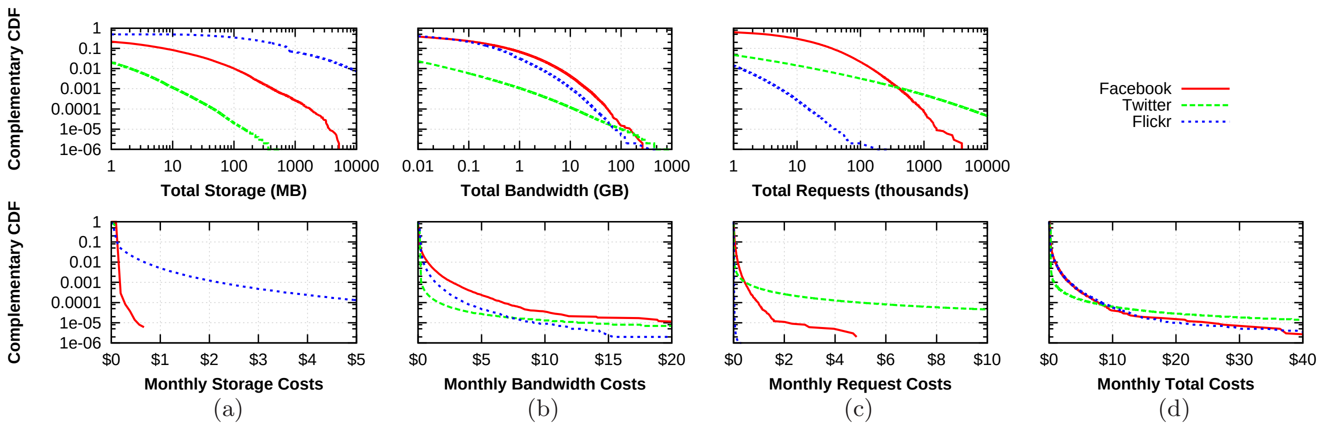


Figure 1: Complementary cumulative distributions of total monthly (a) storage, (b) bandwidth, (c) requests per user (top) and resulting costs (bottom) for Facebook, Twitter, and Flickr users (note the logarithmic scale on the y-axis). Also shown is the distribution of (d) total monthly costs. 99% of users would have to pay no more than \$0.95 per month in all three scenarios.

patterns of Facebook content come from the description of Haystack [8], Facebook's photo serving system. We use the photo view distribution (Figure 7 in [8]) to parametrize our calculations.[4] We assume that videos are viewed with the same popularity distribution as photos, but at 1/20th the rate.

Microblogging: Twitter To estimate the cost of storing and serving Twitter content, we use a data set containing an almost complete set of all tweets issued up to September 2009 [12]. This data set contains 1,755,925,520 tweets issued by 54,981,152 users. We observe that the average size of a tweet (including all metadata fields) is 2551 bytes. Unfortunately, we do not have tweets view counts, but we estimate this by using the number of Twitter followers (subscribers) each issuing user has (i.e., every follower views every tweet).

Content sharing: Flickr To estimate the cost of storing and serving Flickr content, we use a data set from early 2008 consisting of 2,570,535 Flickr users sharing 260,317,120 photos [29]. This data set contains all of the users in the large, weakly connected component on Flickr. We know how many photos were uploaded by each user, but we do not know the number of times each photo was viewed. Instead, we derive the view distribution from studies by Yahoo! researchers [50]. We assume that, on average, photos require 4 MB of storage, and 2 MB of bandwidth per view (photos are typically encoded on disk in multiple sizes [8]).

Analysis We estimate the monthly per-user storage, bandwidth, and request costs for each of these three sets of users on Amazon's S3 service.[5] Figure 1 presents the complementary cumulative distributions of the total storage, bandwidth, and requests per user per month (top) and resulting costs (bottom). Figure 1(d) presents the overall cost per user per month.

We make a number of observations. *First*, the different systems show different cost characteristics: the cost of hosting Facebook content is dominated by bandwidth (due to the high average user degree, requiring distributing the same content to many friends), the cost of hosting Flickr content is dominated by storage (due to the high resolution of Flickr photos), and the cost of hosting Twitter content is dominated by requests (due to the small but frequent content). *Second*, we observe that for the vast majority of users, the total costs are quite tiny: for 99% of users, the monthly total costs are no more than \$0.95 (Facebook), \$0.88 (Flickr), and \$0.23 (Twitter). *Third*, our calculations assume a naïve design; optimizations such as content aggregation and caching are likely to provide lower costs in practice.

3. DESIGN

We now detail the design of Priv.io, comprised of two components: *Priv.io core* and *applications*. Priv.io core provides libraries for accessing user information, manipulating the user's data, and communicating with other users; most user-facing functionality is built as applications on top of Priv.io core. When user visits `https://priv.io`, the Web server returns Priv.io core's JavaScript. This page allows a user to register, log in, control Priv.io settings, and install applications. It also serves as a container for hosting sandboxed applications, and provides libraries for these applications to use. Below, we describe the design of Priv.io core, followed by how applications are implemented (Section 4).

We begin by discussing the assumptions we make (Section 3.1), followed by the Priv.io building blocks (Sections 3.2 and 3.3). We then describe how these are used to implement basic Priv.io functionality (Section 3.4).

3.1 Assumptions

The Priv.io core design includes three components: the Priv.io Web server, users' Web browsers, and users' cloud providers. We briefly overview the assumptions we make

[4] We note that Facebook receives 120 million new photos and 100 billion photo views per day [8]. Given that photos receive 29% of their lifetime views on their first day [8] and, at the time, Facebook users had an average of 130 friends [19], we estimate that newly uploaded photos receive 1.85 views per friend of the uploader on their first day.

[5] At the time of publication, Amazon charges \$0.095/GB/month for storage, \$0.12/GB for outgoing bandwidth (with the first GB free each month), and \$0.004 per 10,000 GET requests [4].

Provider	Storage	Messaging	REST API	Object Versioning	DNS Support	Authentication
Amazon	☑	☑	☑	☑	☑	☑
Azure	☑	☑	☑	☑	☑	☑
Google	☑	☑	☑	☑	☑	☑
HP Cloud	☑	☐	☑	☐	☐	☑
Rackspace	☑	☐	☑	☑	☑	☑
Dropbox	☑	☐	☑	☑	☐	☐

Table 1: Summary of required features in Priv.io, and their current support by major providers. Amazon, Azure, and Google support all required services today.

about each of these. We assume that some entity runs the Priv.io Web server (for now, our research group runs the server, but it could easily be run by a non-profit organization). As we will see later, the Priv.io Web server receives relatively few requests, and it is feasible to run such a server with few resources (for higher reliability, the site could be served using techniques like geo-replication or content distribution networks). We assume that users are running the latest version of a common Web browser with JavaScript and HTML5 support. We assume the security of DNS (i.e., that an attacker cannot modify Priv.io DNS entries).

We assume that the cloud provider provides certain services, listed below:

- **Storage/Messaging** We assume the provider offers both data storage and messaging (distributed queue) services.
- **REST API** We assume that operations can be performed via a REST API [42], enabling access to the API via JavaScript from the user's browser.
- **Versioning** We assume that the provider supports storing multiple versions of objects.
- **DNS support** We assume that users can access their storage containers via DNS names (e.g., `bob.s3.amazonaws.com` maps to Bob's storage).
- **Authentication** We assume that the provider allows permissions to be specified on stored objects.

Table 1 details which of today's providers support these features; we observe that three providers exist that can support Priv.io today. We assume that the users' cloud providers are honest-but-curious, meaning the providers faithfully implement the service that the users have contracted for (e.g., storing objects, retrieving the latest version of objects, delivering messages) but may attempt to decrypt data or messages. Finally, we assume that users' cloud providers are available, meaning the providers do not close their service without warning (users are of course free to migrate their data to new cloud providers at any time).

3.2 Attribute-based encryption

Similar to other content-sharing systems such as Persona [10], Priv.io uses attribute-based encryption (ABE) [9]. In general, ABE dramatically simplifies key management when sharing content with multiple parties. To use ABE, users first generate an ABE public key and an ABE master key (the former is made publicly available and the latter is kept private). Users can then generate ABE private keys for each of their friends, where each ABE private key is generated with one or more *attributes* such as `friend`, `family`, or `yearBorn=1963`.

Users can encrypt content items using expressions over attributes, and only friends whose ABE private key satisfies the given expression are able to decrypt. For example, one such expression might be

$$\texttt{family} \lor (\texttt{yearBorn} < 1980)$$

ABE is collusion-resistant [9], meaning users cannot collude to decrypt content that they could not decrypt separately. For a more detailed description of ABE, we refer the reader to the paper by Bethencourt et al. [9].

3.3 Priv.io building blocks

We now describe the building blocks used in Priv.io; a reference for the notation used is provided in Table 2. As is typical in Web-based services sites, users in Priv.io choose a username and password. Each user u has an ABE master key m_u and an ABE public key P_u. Each user also has a special ABE private key p_u^{self} with the attribute `self`; this allows other users to encrypt messages for u using the `self` attribute, similar to more traditional public key encryption.

The Priv.io Web server serves two functions. First, it distributes the Priv.io JavaScript, CSS, and images to the users when they visit `https://priv.io`. Second, it maintains the `priv.io` DNS domain, which serves as a directory for users' cloud providers.

The Priv.io JavaScript provides libraries for using the REST APIs of the cloud providers' storage and messaging services via XML HTTP Requests (XHRs). In order to use these APIs, though, the JavaScript must respect the default same-origin policy enforced by browsers (i.e., by default, the Priv.io JavaScript cannot make an XHR to `alice.priv.io` unless the HTML document was originally loaded from `alice.priv.io`). Priv.io addresses this problem in one of three ways: (a) providers such as Amazon's S3 and Windows Azure allow users to specify a Cross-Origin Resource Sharing (CORS) [17] policy, allowing such access, (b) systems like Amazon's Simple Queuing Service provide a permissive `crossdomain.xml` file, allowing a small embedded Flash object to make cross-domain requests, or (c) other providers like DropBox allow a stub HTML file to be placed on the target domain, which is used to load the JavaScript in a separate `iframe`.

Notation	Meaning
P_u	u's ABE public key
m_u	u's ABE master key
p_u^v	u's ABE private key given to friend v
p_u^{self}	u's special ABE private key with policy `self`
C_u^{user}	u's credentials for accessing his cloud services
C_u^{friend}	u's credentials, given to his friends, allowing limited access to his cloud services

Table 2: Notation used in the description of Priv.io.

All Priv.io encryption and decryption is implemented in JavaScript; more details are provided in Section 6.

3.4 Priv.io operations

Registration When signing up with Priv.io, a user u visits `https://priv.io` and provides their desired username, password, email address, cloud provider, and two sets of provider access credentials (e.g., AWS access/secret keys). The first set of credentials (C_u^{user}) are to be used by the user himself, while the second set (C_u^{friend}) are to be used by the user's friends. Of the user-provided data, *only the user's username, email address, and cloud provider* are uploaded to the Priv.io server (the email address allows the user to later change their cloud provider).

Meanwhile, the Priv.io JavaScript generates an ABE master key m_u and ABE public key P_u, as well as an ABE private key p_u^{self} with the attribute `self`. Then, using credentials C_u^{user}, the JavaScript creates two storage containers on the cloud provider: a publicly-readable container, and a private container that can only be read with one of the user's two credentials. Finally, the JavaScript creates the user's message queue, configured so that C_u^{friend} is only able to write to the queue.

Upon receiving the user's registration request, the Priv.io server marks the username as assigned and sets up the user's DNS entries. Each user has three DNS entries: `[username].priv.io` maps to the user's public container, `private.[username].priv.io` maps to the user's private container, and `queue.[username].priv.io` maps to the user's message queue.

The Priv.io JavaScript then creates two files in the publicly-readable container: `public_key` containing P_u, and `credentials` containing

$$[m_u, p_u^{\text{self}}, C_u^{\text{user}}, C_u^{\text{friend}}]$$

encrypted using the user's selected password.

Login After a user is registered, login is straightforward. The user visits `https://priv.io` and enters their username and password. The Priv.io JavaScript fetches `[username].priv.io/credentials`, and decrypts the file with the user's password. If the password was correct, the login can proceed, as the JavaScript now has all of the credentials and keys needed to operate on the user's behalf. It is worth noting that the only interaction with the Priv.io server is fetching the Priv.io root page; all others are with the user's cloud provider. A diagram is provided in Figure 2.

Friending Priv.io is built to allow users to interact with friends, and friends need not share the same storage provider. Users can discover friends either through existing friends (e.g., users can browse the list of their friends' friends), or via out-of-band means (e.g., users can exchange Priv.io usernames).

To become friends, users need to securely exchange ABE keys (p_u^v) and credentials for their cloud providers (C_u^{friend}). To do so, let us assume that Alice and Bob wish to become friends in Priv.io. Alice first fetches Bob's ABE public key from `bob.priv.io/public_key`. Then, Alice generates an ABE private key p_{Alice}^{Bob} for Bob, with the attributes Alice assigns to Bob (e.g., `colleague`). Alice then stores

$$[p_{Alice}^{Bob}, C_{Alice}^{\text{friend}}]$$

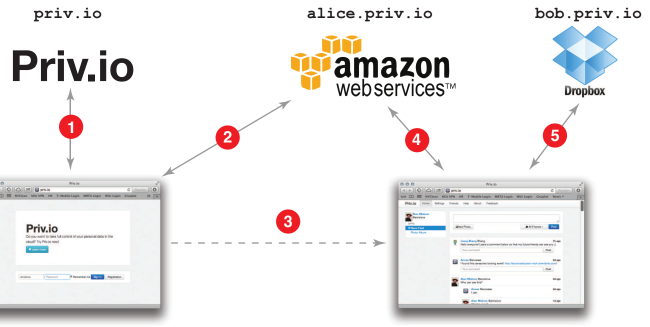


Figure 2: Diagram of login process for user Alice in Priv.io. Alice ❶ visits `https://priv.io`, obtaining the Priv.io JavaScript. Upon entering her username and password, her browser ❷ contacts her cloud provider S3, ❸ verifies her password , and communicates with ❹ her cloud provider as well as ❺ the cloud providers of her friends. Note that the only communication with the main Priv.io server is fetching the original JavaScript.

encrypted under Bob's ABE public key with attribute `self` at the location `alice.priv.io/friends/bob`. Bob performs similar actions for Alice.

Bob then fetches `alice.priv.io/friends/bob`, and decrypts it using p_{Bob}^{self}. Bob is then able to write to Alice's queue and read from Alice's private container (using $C_{Alice}^{\text{friend}}$), as well as decrypt Alice's shared objects (using p_{Alice}^{Bob}). Alice fetches `bob.priv.io/friends/alice` and has similar privileges. Each of the two stores a copy of the newly-acquired credentials and keys in their own private storage encrypted under the policy `self`, allowing each to obtain them on subsequent logins. Finally, both remove the encrypted files from their public container.

If Alice and Bob are two hops away (i.e., one of Alice's friends is also a friend of Bob), Priv.io automatically uses one of the intermediate friends to relay the request. Priv.io sends a message to the intermediate friend, who forwards it on to Bob; Bob is then automatically notified of Alice's incoming friend request. Otherwise, Alice must tell Bob using out-of-band means that she has issued the request. Since the vast majority of friendships in online social networks are established between users who are friends-of-friends [28], we expect most friend requests to be able to be relayed.

Default attributes To simplify sharing, Priv.io generates private keys for friends with two default attributes, in addition to any user-provided attributes. The first attribute is `@username`, which allows users to share content with only a single user (e.g., if Alice wished to share content only with Bob, she could specify the policy `@bob`). The second attribute is `@@`, which is given to all friends. This attribute allows users to share content with all of their friends.

Modifying friend permissions Users in Priv.io may want to change the permissions given to friends, either to add attributes, remove attributes, or remove the friend entirely. Adding attributes simply requires generating a new ABE private key for the friend, and giving the friend the new key. Removing a friend is the same as removing all attributes from the friend.

Removing attributes from a friend requires re-keying. To simplify this process, Priv.io assigns an integral value to each

ABE attribute, where the value is initialized to 0 and is incremented each time a user has that attribute removed. For example, consider user Alice with friends Bob and Charlie assigned the following attributes

Bob	:	`@@=9`, `@bob`, `work=2`, `soccer=1`
Charlie	:	`@@=9`, `@charlie`, `work=2`, `it_dep=3`

Now, if Alice wishes to remove the `work` attribute from Charlie, Priv.io increments the `work` value to 3, and reissues an ABE private key to Bob with the attributes

Bob	:	`@@=9`, `@bob`, `work=3`, `soccer=1`

(note that Priv.io does not need to re-issue a key to Charlie). Any new content Alice shares with the `work` attribute is encoded with the policy `work`$\geq$`3` ensuring that only friends with re-issued keys have access.[6]

Communication Priv.io uses the messaging service of users' cloud providers to enable communication with friends. After logging in, the Priv.io JavaScript connects to the user's queue and processes any messages. While online, the Priv.io JavaScript remains connected the queue and continues to process any additional messages. The only Priv.io control messages that are sent are updating ABE private keys and friendship requests; all other messages are application-level messages and are delivered to the corresponding application (discussed in the following section). Friends do not need to be online for the user to send messages to them; cloud providers typically buffer messages for multiple weeks.

Caching encryption policies ABE operations are significantly more expensive than symmetric encryption operations. To mitigate the impact of expensive ABE operations, Priv.io is configured to use ABE to only encrypt and decrypt AES keys. Actual content objects are then encrypted under AES keys. Furthermore, Priv.io caches the AES keys used for each unique encryption policy; doing so allows Priv.io to only invoke expensive ABE operations when establishing friends, modifying friends, or using a new encryption policy.

4. THIRD-PARTY APPLICATIONS

Almost all user-facing functionality in Priv.io is implemented as applications on top of the Priv.io core libraries. Similar to existing sites like Facebook, applications may be implemented by third parties, and need not be trusted. Applications are implemented using HTML and JavaScript, and are displayed to the user as part of the Priv.io Web page. Thus, the challenge in Priv.io is to provide rich support for third-party applications, while simultaneously providing strict guarantees of security and privacy for users. In particular, we wish to ensure that applications cannot leak user information back to the application provider or any other entity.

4.1 Application API

Priv.io presents an API for applications to be written against. Since Priv.io is implemented entirely within a user's browser, the API is implemented within the browser as well. Priv.io is designed to support social networking-like applications (Facebook, Twitter, and Flickr), but could also be used to build other applications (e.g., Web-based document editing, shared calendars, etc.). Applications in Priv.io are logically separate and cannot exchange data or messages.

Similar to the approach taken by services such as Facebook, applications must request and receive *permission from the user* to make various API calls. When requested, Priv.io presents a dialog to the user, identifying the application and the access that it desires. Priv.io records the user's response, and then uses the specified policy to allow or deny API calls by the application.

A subset of the Priv.io application API is presented in Table 3, and is discussed below:

Method	Description
`requestPermissions(`c`)`	Requests access for the application to methods c
`getUsername()`	Returns the user's username
`getFriends()`	Returns usernames of the user's friends
`getFriends(`u`)`	Returns usernames of friend u's friends
`getAttributes()`	Returns the set of attributes assigned to the user's friends
`store(`k, v, p`)`	Stores data v under key k, encrypted with policy p
`retrieve(`u, k`)`	Returns the value previously stored under key k in u's storage; may return multiple versions
`send(`u, m`)`	Sends message m to friend u's instance of this application
`receive()`	Receives any pending messages
`delete(`m`)`	Marks a previously received message as successfully processed

Table 3: Subset of the Priv.io application API, covering API permissions, user information, storage, and communication. All methods are protected by permissions (users must permit applications to make API calls).

User information Similar to the Facebook API, applications can request profile information about the current user or any of the user's friends.

Storage Applications are allowed to store and retrieve data from the user's private storage container. Each application is given a storage folder, and is only able to access its own content (applications cannot read other applications' data). When storing data, applications specify an ABE policy for encrypting the data (e.g., `self` for only the user, or `family` for all friends with the attribute `family`). Applications can also request to read data in friends' containers written by another instance of the same application, but can only do so if the user is able to decrypt the data.

Communication Applications are allowed to send and receive messages to and from the same application run by friends. This is implemented by sending messages to the specified friend's message queue, and reading from the user's own queue. The Priv.io code multiplexes and demultiplexes messages, and buffers any incoming messages for an application until it is run.

4.2 Managing applications

Developers register Priv.io applications with the Priv.io Web server similar to user registration; each application is given a unique name (e.g., `newsfeed`). The Priv.io Web server makes the application available to users from a subdomain that is hosted by the Priv.io Web server (e.g., the

[6]Any previously-shared content will still be accessible to Charlie, as it was encoded with `work`$\geq$`2`. If this is not desired, content can be re-encrypted with an updated policy.

Newsfeed app is available at http://newsfeed.app.priv.io/). The Priv.io Web server is responsible for serving all application HTML, JavaScript, CSS, and images.

Users install an application by providing Priv.io with the app name (e.g., newsfeed); Priv.io records all apps that a user has installed in the user's private storage, along with their permissions, and reloads the list upon each login. Users can later remove an application by asking Priv.io to delete it. Priv.io then removes the application from the user's list, and deletes any application-stored data and queued messages.

4.3 Security and privacy

Running third-party applications in Priv.io brings up two security concerns: *First*, can we restrict applications to only using the Priv.io API? In other words, can we prevent applications from accessing Priv.io JavaScript objects, or conducting attacks like cross-site scripting [33,34], frame hijacking [6], or frame busting [40]? *Second*, can we prevent applications from leaking user data obtained from the Priv.io API, either via XHRs or by loading DOM (document object model) objects?

In order to address the these concerns, Priv.io *sandboxes* all third-party application code using iframes, loading each application in a separate iframe. Applications access the Priv.io API using the postMessage [23] feature in HTML5 to send API requests to the main Priv.io frame (the application's parent frame). If the API request is allowed (based on user preferences), the response is delivered back to the application via postMessage on the application's iframe.[7] This mechanism prevents applications from directly accessing any Priv.io JavaScript objects.

However, iframes by default are allowed to load arbitrary content, meaning an application could leak user information obtained from the Priv.io API by loading DOM objects. For example, an application wishing to leak the information that user Alice is friends with Bob could request to load http://malicious-domain.com/alice-bob.png. To constrain applications from leaking data, each application's iframe is loaded with a Content Security Policy[8] (CSP). In brief, CSP allows a server to specify what client-side actions the pages it serves can take. Priv.io instructs the browser to disallow the application's iframe from making any network requests other than to [appname].app.priv.io (which is hosted by the main Priv.io server). As a result, the application is constrained to only using the Priv.io API.

4.4 Limitations

Due to the architecture of Priv.io, there are a few applications that exist on sites today that cannot be replicated. For example, any operation that requires a global view of the user data (e.g., global search) is not possible, as there is no entity in Priv.io that can view all data. Other examples include applications that allow users to interact with random users who they are not friends with (e.g., ChatRoulette).

However, many services that might appear to require global information can usually be at least partially replicated. For example, a "friend suggestion" feature could potentially be implemented as an application that collects the structure of the user's local network (friends and friends-of-friends) and suggests others the user likely knows [41]. We leave a more in-depth exploration of such techniques to future work.

[7]Applications cannot impersonate other applications (making messages sent via postMessage appear as if they are from another origin), since the postMessage mechanism is secured by the browser [6].

[8]CSP is a new security mechanism provided in HTML5, and is supported by the latest versions of many browsers.

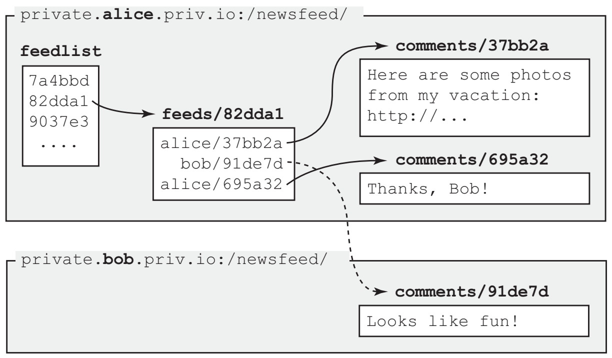


Figure 3: Diagram of how Newsfeed uses storage in Priv.io. Each user stores their own content, and the user who creates each thread stores a feed file linking together all comments.

4.5 Demonstration applications

To demonstrate that existing Web-based services' functionality can be reproduced in Priv.io, we outline two applications that we have implemented.

Newsfeed Priv.io provides functionality similar to Facebook's News Feed via the Newsfeed application. In the application, users start a thread by posting a comment, uploading a photo, or sharing a link. Each thread is shared with a specific ABE policy, controlling which of the user's friends are able to see the thread. Friends who are able to see the thread are able to comment on the thread, and the comments are made visible to all friends for whom the thread is visible. Similar to the News Feed, the threads are sorted by creation date.

Newsfeed stores three types of objects using the storage API. Each individual comment is stored by the user who created it. The user who created the thread also stores a feed object, which simply contains references to all comments in the thread (including the user's original comment). Finally, each user has a single feedlist object that contains references to all feeds created by the user. Newsfeed uses the communication API when a user comments on a friend's feed. A message is sent to the user who owns the feed containing a reference to the comment; when the user owning the feed receives the message, Newsfeed adds the reference to the feed object, allowing other friends to then see the comment.

When users launch the Newsfeed application, it scans all of the friends' feedlists, integrating all of the visible feeds into a single news feed. A diagram showing Newsfeed's use of the storage API is presented in Figure 3.

Chat Priv.io allows users to "chat" by providing the instant messaging application Chat. The application is written entirely using the communications API. Users invite others to chat via a invitation message, and each chat message is broadcast to all other participants of the chat. As a result, Chat provides similar functionality to applications on existing sites, and could easily be extended to (optionally) archive conversations, allow file transfers, and so forth.

5. DISCUSSION

We now discuss a few deployment issues with Priv.io.

Consistency and reliability In Priv.io, users only write to their own storage location, preventing a number of consistency problems. However, users may be logged in to Priv.io from multiple locations at once, exposing Priv.io to potential consistency issues due to multiple writers. To address this problem, Priv.io leverages the object versioning (described in Section 3) supported by the cloud provider. Specifically, when Priv.io writes an updated version of an object to the user's storage location, it first checks to see if there is a newer version of the object present than the one that its pending write is based on. If such an object exists, Priv.io first downloads the updated object, and *merges* the two. Finally, Priv.io writes the new version of the object back, and deletes both of the previous versions. As a result, Priv.io itself and all applications must be able to perform merges on storage objects that may have diverged.

Priv.io allows each user to select the desired level of availability and durability for their content through the choice of their cloud provider. For example, on Amazon's S3 service, users can choose between eleven 9s of durability or four 9s of durability for content, at different price points.

Reliable message delivery Because Priv.io is implemented within a browser, the user could decide to close the window at any time, thereby killing all Priv.io JavaScript. This property makes implementing reliable message delivery for applications particularly challenging, as a message may be delivered to an application, but the Priv.io window could be closed before the application finishes processing the message. To avoid such a scenario, Priv.io requires applications to explicitly call `delete(`m`)` on each message m after they have finished processing it. Only at that point is the message deleted from the cloud provider's message queue.[9] Thus, Priv.io provides *at-least-once* delivery semantics for messages, and applications must be written to tolerate receiving the same message multiple times.

Security To prevent man-in-the-middle attacks on Priv.io users, all interaction with the Priv.io Web server is over HTTPS. In the future, we aim to provide support for DNSSEC to ensure the integrity of Priv.io DNS entries as well (e.g., to prevent cache poisoning attacks).

Each user's encrypted `credentials` file is stored in a publicly visible location (e.g., `alice.priv.io/credentials`). As a result, we are particularly concerned about brute-force password cracking attacks. To reduce the ability for an attacker to decrypt a user's `credentials` file, we first choose a random initialization vector when encrypting the credentials and salt the password, preventing attackers from using rainbow tables [36]. Second, we use the PBKDF2 password strengthener [25], which greatly raises the cost of a brute-force attack. Third, we require strong passwords for users in the form of pass phrases [37], which often possess more entropy than basic passwords.

An additional concern is whether an attacker can perform a man-in-the-middle attack during friend establishment, or intercept the exchanged friend credentials. We first note that all friend exchange information is encrypted with the destination's ABE public key, making it unreadable by the attacker. Second, since each user is the only entity that is able to write to their public storage area, malicious users are unable to forge friend requests with credentials of their choosing.

Privacy As privacy is of paramount concern in Priv.io, we now briefly analyze what an attacker can determine about other users. We first note that the attacker can only read the user's public storage location; he cannot access the user's queue or private storage location. There are three types of objects stored in the public storage area: the `credentials` file, the `public_key` file, and the temporary friend request files discussed in Section 3.4 (recall that these files are available while a friend request is outstanding). Thus, malicious attackers are able to determine if user u is currently trying to become friends with user v. However, we note that attackers must guess the identity of v correctly (attackers cannot "list" all request files), and the window of opportunity is likely to be short.

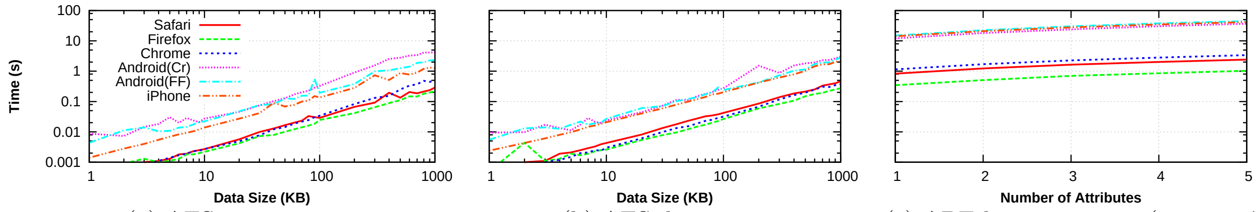
Incremental deployment Signing up for Priv.io is more complicated than signing up for existing services like Facebook: with Priv.io, users must first sign up with a cloud provider, and then register for Priv.io using those credentials. Luckily, signing up with cloud providers is often relatively simple; for example, signing up for Amazon Web Services requires only filling out two Web forms (personal information and billing information), and much is carried over if the user already has an Amazon account. Regardless, we will continue to look for opportunities to lower the burden for signing up with Priv.io.

For social networking-like services, the network effect[10] has resulted in entrenched service providers (e.g., Facebook); being the only one of your friends to be on a new social network is unlikely to provide significant benefits. Thus, it may be difficult to initially attract significant numbers of users to Priv.io without an established user base. However, we note that Priv.io is not limited to social networking applications, as current popular Web-based applications like Google Docs could easily be implemented in Priv.io and afforded the same privacy benefits. This approach may serve as a mechanism for attracting users, who later may also use the social networking features.

Finer-granularity computation The design of Priv.io is partially driven by the high cost of purchasing computation from cloud providers. However, as new cloud providers enter the market (e.g., resellers of existing cloud providers) or the price of computation drops, purchasing computation may become feasible. If so, this would open new opportunities for Priv.io, but may also come with a different set of privacy properties. For example, if users could purchase computation at a low-enough cost, Priv.io could easily interact with existing legacy systems that cannot be supported with in-browser computation (e.g., applications such as Google Mail—requiring SMTP—could be replicated in a confederated manner). However, doing so would allow the cloud provider to potentially view raw content (as incoming emails

[9]Message queues like Amazon's SQS and Microsoft's Azure Queue Service support similar semantics; if a recipient dies before marking a message as processed, the message is eventually delivered again.

[10]The *network effect* describes the value of a network as the number of participants grows. In brief, it captures the notion that with each new user, the number of potential links increases, thereby increasing the value for all participants.



(a) AES encryption (b) AES decryption (c) ABE key generation (`gen_key`)

Figure 4: Priv.io encryption and decryption performance when run in different browsers. Shown is (a) AES encryption and (b) AES decryption for objects of different sizes. Also shown is (c) ABE key generation for keys with an increasing number of attributes (ABE encryptions under policies of increasing lengths shows very similar behavior).

would be observed in plaintext by the user's server). Regardless, we plan to explore ways of integrating purchased computation into Priv.io as future work.

6. EVALUATION

We now present an evaluation Priv.io, covering both microbenchmarks and measurements of Priv.io performance under different workloads.

We have implemented a prototype of Priv.io that supports almost all of the features described thus far. Priv.io currently supports using Amazon Web Services as a cloud provider, with support for SQS and S3. Support for Windows Azure and Google Cloud Platform is in progress. Priv.io supports third-party applications; both of the applications described in Section 4.5 are implemented and installed by default for each user.

Since all of Priv.io is implemented in JavaScript, it is open-source and available to the research community at `https://priv.io`. The implementation is compatible with the latest versions of common desktop Web browsers, as well as browers on Android and iOS.

The Priv.io core code represents 5,931 lines of JavaScript, excluding encryption and user interface libraries. We use the Stanford JavaScript Crypto Library for all AES operations. We used Emscripten [52] to compile the Ciphertext Policy ABE library [16] (as well as other dependencies) into JavaScript. The resulting encryption library totals 621 KB. All of these libraries are static and can easily be cached by Web browsers.

6.1 Microbenchmarks

Storage size Priv.io objects require storage on the user's cloud provider and encounter overhead, due both to encryption metadata (initialization vectors, etc.) and the base64 encoding used. The fixed overhead of using AES encryption is 145 bytes, and the fixed overhead of using ABE encryption is 345 bytes plus approximately 370 bytes per policy attribute. The base64 encoding introduces an additional 33% overhead. The ABE public keys are 1184 bytes, the encrypted `credentials` file averages 1457 bytes, and each friend request file averages 2400 bytes.

Browser	`setup` **time (s)**	`decrypt` **time (s)**
Safari	0.91	0.99
Firefox	0.63	0.36
Chrome	1.22	1.38
Android(Cr)	12.92	14.54
Android(FF)	14.63	13.08
iPhone	14.40	15.92

Table 4: Average time taken to generate an ABE master and public key (`setup`) and decrypt an ABE message (`decrypt`) in various browsers.

Content loading latency Loading objects in Priv.io enjoys the benefits of the user's cloud provider; we found the latency to be comparable to loading content from traditional Web sites. Using the `us-east-1` Amazon Web Services S3 storage service and loading to a client located in Boston, we found the latency of loading 64 KB objects via Priv.io to be 154 ms.

Encryption and decryption We now examine the encryption and decryption performance in Priv.io. We first focus on AES encryption. Using the latest version[11] of common browsers, we encrypt and decrypt objects of varying sizes using the AES library. We repeat each test 10 times, and report the average in Figures 4(a) and 4(b). We observe that AES encryption and decryption time correlate linearly with object size, and are fast: for 100 KB objects, both are under 43 ms for all desktops and under 327 ms for all mobile devices.

We now examine the performance of ABE. There are four ABE operations that we need to consider: `setup` (generate public and master keys), `gen_key` (generate a private key), `encrypt`, and `decrypt`. Of these, the compute time of `gen_key` and `encrypt` depend strongly on the number of attributes; the compute time of the other two is relatively constant.

We first report the performance of `setup` and `decrypt` in Table 4. We observe that the performance ranges from under 1.4 seconds on desktop browsers to about 15 seconds on mobile devices. We next examine the performance of `gen_key` and `encrypt`, shown in Figure 4(c) (the two operations show almost identical performance, so we only present the results for `gen_key` for brevity). We observe a strong linear relationship with the number of attributes used, ranging from about one second for a single attribute on desktop browsers to 45 seconds for five attributes on mobile devices. We again note that the expensive nature of ABE operations is unlikely to impact users on a regular basis, as they are only necessary when adding/modifying friends, or encrypting content

[11]Safari 6.0.4, FireFox 21.0.1 and Chrome 27.0.1453, all on OS X 10.8.3; Android Chrome 27.0.1453 and Firefox 21 on a HTC One X (AT&T); Mobile Safari 6.0 on an iPhone 5 running iOS 6.1.4. Mobile devices are connected via WiFi.

Browser	Priv.io (s)	Newsfeed (s)	Chat (s)
Safari	0.33	0.19	0.07
Firefox	0.34	0.20	0.12
Chrome	0.25	0.09	0.04
Android(Cr)	1.89	0.36	N/A
Android(FF)	2.81	0.98	0.83
iPhone	0.67	0.77	N/A

Table 5: Average time (in seconds) taken to load the basic Priv.io code after login, as well as the Newsfeed and Chat applications, in various browsers. Android Chrome and Mobile Safari do not support Flash, which is used to make cross domain request for Amazon SQS.

under a never-used-before policy (i.e., most sessions require no ABE operations).

6.2 User-perceived performance

We now examine the user-perceived performance of Priv.io. In evaluating Web services, the primary metric of interest is typically latency; we therefore focus on latency here. We are primarily concerned with three issues: *First*, what is the loading time of the basic Priv.io code when a user logs in, absent any applications? *Second*, what is the loading time of applications, both with and without subsequently loaded content? *Third*, what is the latency of sending application-level messages?

Priv.io loading time We first measure the time taken for users to log in and load the basic Priv.io code, absent any applications. To measure this, we disable all applications and measure the time taken from when a user clicks "Log in" until the Priv.io code is fully loaded. We run this experiment on all of the browsers listed above, clearing the browser cache after each experiment, and repeating the experiment 10 times. The average loading time is shown in Table 5, under the "Priv.io" column. We observe that the loading time is quite fast, between 250 and 340 ms on desktops and between 0.6 and 2.8 seconds on mobile devices.

Application loading time Next, we explore the loading time of applications. We record the time taken to load the Newsfeed and Chat applications once the Priv.io code is loaded; in each instance, the applications are empty and contain no user data (we explore the loading latency when user data is present below). The results are presented in Table 5, under the application columns. We observe that the loading time is consistent across different applications, and ranges from 90 to 200 ms on desktops to between 360 and 980 ms on mobile devices.

Next, we explore the loading time of applications when user data is present. To measure this, we use the Newsfeed application, and create users with varying amounts of Newsfeed content from varying numbers of friends. Specifically, we load up to 15 Newsfeed items (i.e., one "page" of Newsfeed items), with each friend providing three items (i.e., we create one user who loads three items from one friend, another user who loads three items each from two friends, and so on). The average loading time is presented in Figure 5. We observe that the loading time increases linearly with the number of content items loaded, and that the desktop brewers are substantially quicker in loading content, as expected. However, we observe that the loading time is reasonable in

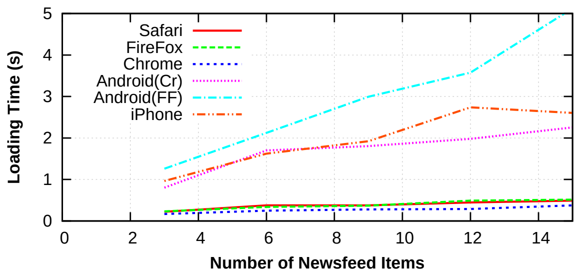


Figure 5: Average Newsfeed loading time with varying amounts of content, when content is loaded from multiple friends. We observe that the loading time increases linearly with the amount of content, as expected.

all cases: below 515 ms for desktops and below 5.1 seconds for all mobile devices.

Message latency Finally, we examine the latency of sending application-level messages. To do so, we use the Chat application, measuring the time for a user to send a message to a friend, and for the friend to reply. Both sender and receiver logged in on the same machine and browsers (note that the message itself must be delivered via the cloud provider's servers). We repeat this experiment in different browsers, and for each browser, we send 10 round trip messages and calculate the average. We find that the round trip time varies from an average of 637 ms on Chrome to 1.3 seconds on Safari[12], indicating that cloud providers' messaging can easily be used for human-timescale communication.

Overall, our results indicate that Priv.io is practical on the desktop Web browsers of today, with most user-facing loading times on the order of a second. However, mobile devices present challenges for Priv.io, as their lower computational resources result in higher latencies. Our results show that Priv.io does work on these devices, and as they become more powerful, accessing Priv.io from them will become more practical.

6.3 Small-scale deployment

We have deployed Priv.io on a small scale within our department. Unfortunately, it is difficult to measure the primary benefits of Priv.io to our users: improvements in privacy and control over data. As of this writing, 28 graduate students and professors have joined Priv.io and are using the Newsfeed and Chat applications. There were a total of 88 friendships recorded, for an average of 3.82 friends per user. Our users have accessed the service using a variety of operating systems, browsers, and desktop/mobile devices (a total of 23 different `User-Agents`). In total, our users have posted 221 items to Priv.io, most of which are comments in the Newsfeed application.

7. RELATED WORK

Enhancing Web browsers Over the past decade, Web browsers have become significantly more advanced. Researchers have explored using process-based models to iso-

[12]This latency could be further reduced if Amazon's SQS supported Cross-Origin Resource Sharing, which would eliminate the need for a Flash-based work-around.

late misbehaving Web pages [39], have examined moving beyond the same-origin policy of privilege separation [1], built systems that allow third-party code to execute while providing security and privacy guarantees [14, 24], and implemented iframe-based sandboxing [23]. We leverage many of these advances and techniques in the design of Priv.io.

Modifying existing Web services In parallel, many groups have explored ways to provide greater user privacy by re-architecting existing Web services. Developers have built subscription-based services such as app.net [5], which promise to not show ads in exchange for a yearly fee; unfortunately these simply replace one centralized provider with another. Systems have also been built that guarantee sandboxing of third-party applications [49], but these do not address hiding information from the service provider. Finally, researchers have developed approaches that enable sharing of provider-hosted content among different providers and with the user's local machine [21]; however, these do not address the issue of privacy from the centralized provider.

Others have explored retaining existing centralized providers, but hiding certain information from the provider. For example, researchers have explored encrypting uploaded content [22, 38], encrypting social relationships [47], and keeping data on user-managed devices [51]. It is unclear whether existing providers are amenable to these solutions (as they directly impact the providers' revenue stream), and deploying them independently risks users being banned by the provider.

Building new Web services Researchers have also explored new approaches that operate via the Web. For example, Persona [10] (which inspired our design, and in particular, our approach for encrypting content) stores encrypted user data on user-contracted storage services. Similar approaches include Vis-à-Vis [43] (storing data on group-based EC2 machines), Confidant [27] (storing data on friends' machines) and others [30] (storing data on users' home routers). Unfortunately, all of these solutions require client-side changes in order to work, and assume that they have a less-restricted model of computation than is available to JavaScript within the browser. In contrast, Priv.io uses many aspects of these systems' design, but does so without requiring any client-side changes and supports potentially untrusted third-party applications.

Others have explored separating Web-based services from user data. For example, W5 [26] proposed an architecture that separates Web service developers from providers that execute service code and host user data in a secure manner. While the vision of Priv.io and W5 are similar, to the best of our knowledge, W5 has not been deployed nor has any providers become available. BStore [13], provides a generic file system-like interface for Web applications, allowing users flexibility in the location of their data. Priv.io stores data on cloud providers using a number of techniques that were proposed in BStore. However, BStore is focused on providing file storage, while Priv.io also deals with challenges of sharing data with others, supporting third-party applications, and demonstrating that existing services can be replicated in a confederated manner.

Non-Web approaches Finally, researchers have presented systems that implement services in a decentralized fashion. These include PeerSoN [11], Diaspora* [18], Safebook [15], Contrail [45], and others [2]. While similar to Priv.io in goals, all of these approaches require client software to be downloaded, and also generally face challenges in ensuring availability [7]. Others have designed protocols [48] that allow users to host their data on dedicated, secure servers of their choosing. A more detailed overview of the tradeoffs of decentralized architectures is provided in [44].

8. CONCLUSION

We presented Priv.io, a new approach to building Web-based services using a confederated architecture. In Priv.io, each user is responsible for providing the resources necessary to support their use of the service; this is accomplished by contracting with cloud providers (for storage, bandwidth, and messaging) and by using the user's Web browser (for computation). As a result, in Priv.io, users retain control of their own data, users are not required to reveal their information to any centralized entity, and users enjoy a highly reliable and available service. We demonstrated that implementing many popular services with Priv.io is both practical and affordable: Most users would pay less than \$0.95 per month, and Priv.io works today on the latest versions of common Web browsers as well as (more slowly) on Android and iOS mobile devices.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Ben Zhao, for their helpful comments. We also thank the Priv.io users and beta testers for their hard work and patience. This research was supported by NSF grants IIS-0964465, CNS-1054233, CNS-1319019, and an Amazon Web Services in Education Grant.

9. REFERENCES

[1] D. Akhawe, P. Sazena, and D. Song. Privilege Separation in HTML5 Applications. *USENIX ATC*, Boston, MA, 2012.

[2] J. Anderson, C. Diaz, J. Bonneau, and F. Stajano. Privacy-Enabling Social Networking Over Untrusted Networks. *WOSN*, Barcelona, Spain, 2009.

[3] Amazon EC2 Pricing. `http://aws.amazon.com/ec2/pricing`.

[4] Amazon S3 Pricing. `http://aws.amazon.com/s3/pricing`.

[5] app.net. `http://join.app.net`.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Securing Frame Communication in Browsers. *USENIX Security*, San Jose, CA, 2008.

[7] C. Blake and R. Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. *HotOS*, Lihue, HI, 2003.

[8] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. *OSDI*, Vancouver, Canada, 2010.

[9] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-Policy Attribute-Based Encryption. *IEEE S&P*, Oakland, CA, 2007.

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

[11] S. Buchegger, D. Schiöberg, L. H. Vu, and A. Datta. PeerSoN: P2P Social Networking—Early Experiences and Insights. *SNS*, Nuremberg, Germany, 2009.

[12] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. *ICWSM*, Washington, D.C., 2010.

[13] R. Chandra, P. Gupta, and N. Zeldovich. Separating Web Applications from User Data Storage with BStore. *WebApps*, Boston, MA, 2010.

[14] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen. Virtual Browser: a Virtualized Browser to Sandbox Third-party JavaScripts with Enhanced Security. *CCS*, Chicago, IL, 2010.

[15] L. A. Cutillo and R. Molva. Safebook: A Privacy-Preserving Online Social Network Leveraging on Real-Life Trust. *IEEE Communications*, 43(12), 2009.

[16] Ciphertext Policy Attribute-Based Encryption. `http://acsc.cs.utexas.edu/cpabe`.

[17] Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`.

[18] Diaspora*. `http://www.joindiaspora.com/`.

[19] Facebook Statistics. `http://on.fb.me/UtWB0`.

[20] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao. Detecting and Characterizing Social Spam Campaigns. *IMC*, Melbourne, Victoria, Australia, 2010.

[21] R. Geambasu, C. Cheung, A. Moshchuk, S. D. Gribble, and a. H. M. Levy. Organizing and Sharing Distributed Personal Web-Service Data. *WWW*, Beijing, China, 2008.

[22] S. Guha, K. Tang, and P. Francis. NOYB: Privacy in Online Social Networks. *WOSN*, Seattle, WA, 2008.

[23] HTML5 Specification. `http://bit.ly/3h8KZG`.

[24] L. Ingram and M. Walfish. TreeHouse: JavaScript sandboxes to help Web developers help themselves. *USENIX ATC*, Boston, MA, 2012.

[25] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, IETF, 2000.

[26] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A World Wide Web Without Walls. *HotNets*, Atlanta, GA, 2007.

[27] D. Liu, A. Shakimov, R. Cáceres, A. Varshavsky, and L. P. Cox. Confidant: Protecting OSN Data without Locking It Up. *Middleware*, Lisbon, Portugal, 2011.

[28] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Growth of the Flickr Social Network. *WOSN*, Seattle, WA, 2008.

[29] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. *IMC*, San Diego, CA, 2007.

[30] M. Marcon, B. Viswanath, M. Cha, and K. P. Gummadi. Sharing Social Networking Content from Home: A Measurement-driven Feasibility Study. *NOSSDAV*, Vancouver, Canada, 2011.

[31] R. Miller. Ma.gnolia Data is Gone for Good. `http://bit.ly/tbFup`.

[32] C. B. Myers. Google to finally shut down Google Buzz, along with Google Labs. *The Next Web*, 2011. `http://tnw.co/pptfQp`.

[33] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. *NDSS*, San Diego, CA, 2007.

[34] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. *NDSS*, San Diego, CA, 2009.

[35] G. Ou. Facebook slashes the quality of "HD" videos. `http://bit.ly/bvx86h`.

[36] P. Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. *CRYPTO*, Santa Barbara, CA, 2003.

[37] S. N. Porter. A password extension for improved human factors. *Comp. & Sec.*, 1(1), 1982.

[38] priv.ly. `http://priv.ly`.

[39] C. Reis, A. Barth, and C. Pizano. Browser security: Lessons from Google Chrome. *CACM*, 52(8), 2009.

[40] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *W2SP*, Oakland, CA, 2010.

[41] M. Roth, A. Ben-David, D. Deutscher, G. Flysher, I. Horn, A. Leichtberg, N. Leiser, Y. Matias, and R. Merom. Suggesting Friends Using the Implicit Social Graph. *KDD*, Washington, D.C., 2010.

[42] Representational State Transfer. `http://en.wikipedia.org/wiki/Representational_state_transfer`.

[43] A. Shakimov, H. Lim, R. Cáceres, L. P. Cox, K. Li, D. Liu, and A. Varshavsky. Vis-à-Vis: Privacy-Preserving Online Social Networking via Virtual Individual Servers. *COMSNETS*, Bangalore, India, 2011.

[44] A. Shakimov, A. Varshavsky, L. P. Cox, and R. Cáceres. Privacy, Cost, and Availability Tradeoffs in Decentralized OSNs. *WOSN*, Barcelona, Spain, 2009.

[45] P. Stuedi, I. Mohomed, M. Balakrishnan, Z. M. Mao, V. Ramasubramanian, D. Terry, and T. Wobber. Contrail: Enabling Decentralized Social Networks on Smartphones. *Middleware*, Lisbon, Portugal, 2011.

[46] A. Tootoonchian, K. K. Gollu, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: Social Access Control for Web 2.0. *WOSN*, Seattle, WA, 2008.

[47] A. Tootoonchian, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: Better Privacy for Social Networks. *CoNEXT*, Rome, Italy, 2009.

[48] tent.io. `http://tent.io`.

[49] B. Viswanath, E. Kıcıman, and S. Sariou. Keeping Information Safe from Social Networking Apps. *WOSN*, Helsinki, Finland, 2012.

[50] R. van Zwol. Flickr: Who is Looking? *WI*, Silicon Valley, CA, 2007.

[51] C. Wilson, T. Steinbauer, G. Wang, A. Sala, H. Zheng, and B. Y. Zhao. Privacy, Availability and Economics in the Polaris Mobile Social Network. *HotMobile*, Phoenix, AZ, 2011.

[52] A. Zakai. Emscripten: an LLVM-to-JavaScript compiler. *SPLASH*, Portland, OR, 2011.

[53] Zynga. `http://zynga.com`.