Practical Protection for Personal Storage in the Cloud

Neal H. Walfield Paul T. Stanton John Linwood Griffin Randal Burns
Johns Hopkins University
{neal,pauls,jlg,randal}@cs.jhu.edu

ABSTRACT

We present a storage management framework for Web 2.0 services that places users back in control of their data. Current Web services complicate data management due to data lock-in and lack usable protection mechanisms, which makes cross-service sharing risky. Our framework allows multiple Web services shared access to a single copy of data that resides on a personal storage repository, which the user acquires from a cloud storage provider. Access control is based on hierarchically, filtered views, which simplify cross-cutting policies, and enable least privilege management. We also integrate a *powerbox* [16], which allows applications to request additional authority at run time thereby enabling applications running under a least privilege regime to provide useful open and save as dialogs.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls; D.4.7 [**Organization and Design**]: Distributed systems

General Terms

Design, Security

Keywords

Access Control, Cloud, Web Services, Data Management

1. INTRODUCTION

Web 2.0 services expect to own the data that they use and expect users to only access that data via their interfaces. These expectations complicate data management and limit sharing. Data management is complicated because data can usually only be accessed via non-standard interfaces, which make data extraction difficult (*data lock-in*) [2], because data are dispersed among services, which makes finding data difficult (*data spew*), and because the assumption that a service is the sole user of data does not hold resulting in multiple copies of data, which become incoherent (*version drift*). The lack of standard interfaces and useful access control mechanisms limits sharing. Currently, a user enables one service (e.g., Facebook) to access data managed by another service (e.g., her Hotmail address book) by providing the former her credentials for the latter. This gives the first service full access to the second account including all of the user's data and the ability to impersonate the user. These are serious security concerns.

Web services offer users two convincing features over local applications: they facilitate user mobility by providing access to the same application state independent of device and location, and they facilitate sharing and collaboration. Currently, these advantages come at the cost of more difficult data management and poor security. This hurts Web 2.0's value proposition: these problems discourage the use of experimental or unreputed services, which translates to conservative user behavior and slow industry innovation.

Having services use shared storage would solve the data management issues, but current techniques lack adequate protection mechanisms. Services could use user-provided storage, which is possible today given inexpensive services, such as Amazon S3, Nirvanix, and Rackspace. These services, however, provide protection mechanisms that are difficult to use in a manner consistent with the principle of least privilege (POLP) [13]. S3 and Nirvanix provide ACLs, but these make access rights management in a dynamic environment an ongoing chore for users. This critique applies equally well to capabilities when they refer to files, e.g., as in Tahoe [18]. S3 allows the use of an external reference monitor, which could be used to realize any security policy, but this mechiams is too low-level for even power users.

In this paper, we implement a framework that solves the data management and security problems. Our framework relocates storage to a user-managed storage repository. A user provides services access to the storage area. Protection is based on hierarchical, filtered views of the name space. Similar filtering techniques have been used in managing access in databases [5], and CloudViews has proposed them for use in the cloud, albeit only among services in a single cloud [4]. To delegate access, a user creates a principal, associates a view with it, and provides the principal's credentials to the service. A view consists of access rights and a filter, for which we use regular expressions. A principal can later be granted additional authority or have some authority revoked by adding or removing views. We think that users can easily conceptualize views and that they lend themselves to be used in a manner that approximates POLP. Views also

EUROSEC '10, Paris, France

have the advantage that they allow access to objects when they come into existence. Further, views ensure consistent naming of objects in all services.

To further improve usability, our implementation includes a *powerbox* [16], which enables an application to request more authority at run-time. A powerbox runs with the user's authority on the user's computer. Using a powerbox in place of an application-implemented open or save-as dialog box allows the the user to use any file, not just those that the service is authorized to see, which is significantly constrained when its authority is managed consistent with the POLP. The powerbox simplifies management because, when the user selects a file in the powerbox, the powerbox delegates access to the application. Thus, the user's designation becomes an authorization.

Our S4 prototype consists of approximately 4,000 lines of Python. It extends S3's REST interface and is mostly backwards compatible. The S3 programs that we tested work without modification.

2. SCENARIOS

We use two running examples. The first involves Alice, an avid user of social networking services. The second considers Bob who creates documents, which mash up data from multiple sources. He sends these documents to his colleagues as attachments using Hotmail.

Alice uses multiple social networking services including Facebook and Last.fm. She regularly updates her profile picture on all of her social networking services. Currently, she has to log in to each service. She finds this repetitive. Alice also uses Hotmail. Alice would like her social networking services to automatically identify potential connections using her address book. She often adds new contacts to her address book and would like those changes to propagate promptly. Likewise, she would like her services to automatically update her Hotmail address book based on contact information that her connections provide.

Bob creates documents using Google Docs. He often includes photos, many of which he stores on Flickr. To incorporate a file stored on Flickr, Bob has to download the file to his local computer and then upload it to Google Docs. He would prefer to be able to access the file directly from Google Docs. The copying also creates problems: if he modifies the copy in Google Docs, he has to manually synchronize the changes with Flickr. Sometimes, he forgets. The problem becomes worse when multiple copies have different sets of edits, which he merges manually.

3. TOWARD PER USER REPOSITORIES

Web services expect that they own the data they use and that users access data through the provided interfaces. These expectations lead to an architecture that makes data management hard and sharing difficult and dangerous.

3.1 Data Management Issues

Most Web services do not make accessing and modifying user data easy; they view their copy as the sole authoritative copy. This is illustrated in Figure 1. This results in a form of data lock-in [2] in which users have a limited ability to access and modify their data. Consider how Alice updates her profile picture on her social networking services: she logs in to each service, navigates to the change-profile-picture

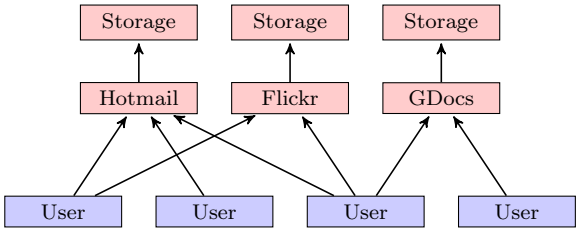

Figure 1: Web 2.0 services today: each service has its own storage, which users cannot access directly.

dialog, and then uploads a picture stored on her local hard drive. There is no way for Alice to designate a picture on the Web as being her current profile picture and have all services automatically update her picture when it changes.

Accessing and modifying data is not only hard for users, it is also hard for programmers. To write a program to change Alice's profile picture on her social networking services would require writing code specific to each service—services have their own API. As a consequence, tools tend to support only the most popular services. Although a standard API would help, applications would still need to use distributed version control algorithms for updating shared state, which is non-trivial even if ignoring access control.

This attitude toward user data also results in data spew: the inability to maintain a coherent view of data. If Alice wants to use a photo for her profile picture, she has to remember which service stores it. This is not a problem if all photos are managed by a single service. However, if she uses multiple photo management services, the data type is no longer an indicator for the service.

Another problem that can occur is version drift, which is when multiple copies of a file become out of sync. This arises because users must manually synchronize changes across all copies. When Bob copies a picture from Flickr to Google Docs and then edits it, he has to manually update the other copy. If he forgets and later the copy on Flickr is changed, e.g., it is tagged, he now has to manually merge the changes.

3.2 Security Issues

Web services lack adequate protection mechanisms for sharing data with other services in a manner consistent with POLP. Consider again Alice, who would like Facebook to monitor her Hotmail address book. She could use Facebook's Friend Finder. However, it requires her to enter her Hotmail user name and password: there is no way for her to just give Friend Finder access to her address book; her credentials give Friend Finder the authority to not only access her address book but also to read her email and impersonate her by sending email from her account. Alice's only available protection mechanism is a simple binary decision: give Friend Finder access to all of her Hotmail authority or none. This decision can be rephrased as: participate and be vulnerable or be safe but do not participate [15]. Although Alice might trust Facebook, this lack of a fine-grained sharing API impedes experimentation with unreputed services. Even if Hotmail supported fine-grained access control, this is only one service. All other services would need to be changed to support similar APIs. This problem could instead be addressed by the relatively few storage providers.

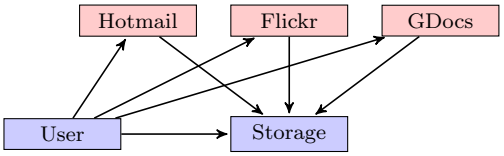

Figure 2: Web 2.0 services using a user-specified, user-controlled storage repository.

3.3 Per-User Repositories

A solution that solves the data management problems is one that centralizes a user's data with a storage provider, as shown in Figure 2. This creates a star topology: the user's storage repository is the hub, and services share data by way of the storage repository. A user authorizes services to access a file. When a service changes a file, it updates the copy on the storage repository. Services interested in changes to a file monitor that file. When they notice that a file has changed, they integrate the changes.

This architecture does not preclude services using their own storage to cache user data; it only requires that the user's repository hold the authoritative versions of the user's objects. This mean services must promptly integrate and propagate changes.

Preventing services from keeping local copies of data would hurt performance. For instance, if a service had to fetch data from a user's storage repository every time the file is requested, this would add bandwidth costs and latency. Also, services often precompute data to save time, e.g., Flickr stores a number of scaled versions of an image. Having to compute scaled versions on demand would be expensive.

3.4 Requirements

We distill these observations into a set of requirements for the protection mechanisms of the system:

1. A user can delegate access to just those objects that a service needs to realize the user's intents (POLP);
2. A user can revoke a service's subsequent access to any object at any time;
3. A service can transitively delegate access to an object to a third-party (e.g., Facebook can delegate access to an object to an application);
4. A principal can revoke a principal's access to an object if and only if it or a principal it dominates previously delegated access to that principal and that access was not subsequently revoked;
5. A principal can revoke another principal's access to an object without also revoking other principals' access (e.g., a user can revoke Google Doc's access to an object without also revoking Flickr's).
6. The API should be identical for all principals (*transparent interposition*);
7. When a principal revokes a delegation, any delegations based on that delegation are also revoked;
8. A user's explicit interactions with the security manager are minimized (usable security);
9. An object has the same name in all name spaces it appears in (consistent naming); and,
10. A user can delegate access to not-yet-existing objects.

The first seven requirements deal with protection and require fine-grained, transitive delegation and subsequent revocation. The remaining concern usability. The security of the system should be as invisible to the user as possible.

The consistent naming requirement arises because the names of the objects benefit users and are presented to the user by the service. If services have their own names for objects, it will be difficult for the user to find the objects in which he is interested. This is a form of data spew.

The requirement that allows delegating access to objects that do not yet exist enables useful security policies. Consider Bob who wants Flickr to access all of his public photos. He should be able to articulate this policy and whenever he uploads photos, they should be immediately accessible to Flickr; Bob should not have to implement this policy manually. Moreover, it is conceivable that the user does not even interact with the device when it is uploading the data: Bob's mobile phone might push photos when there is inexpensive connectivity. Any mechanism that enables future delegations must be designed carefully as a policy may inadvertently share data that should not be shared.

4. RELATED WORK

File-Oriented Access Control: Most common protection mechanisms are approximate realizations of *Lampson's Access Matrix* [8]. These include distributed file systems such as *NFS*, *AFS* [14], *SFS* [7], Amazon's *S3* [1], and *Nirvanix* [10], which provide ACLs, as well as *CapaFS* [12], *Tahoe* [18], and *Secure FS* [6], which are capability based.

These systems require that access be delegated one file or directory at a time, which does not match how users conceive of security policies. Consider Bob. He would like to give Flickr access to his public photos. Granting access on a file-by-file basis is tedious and requires constant care: he must update his security policy whenever he adds new public photos. Using directories, Bob could partition his files and grant Flickr access to a directory. This separation is inconvenient if the security policy does not match how he wants to structure his name space: Bob may want to place each photo set in its own directory, but does not deem all photos in a set to be either public or private.

An important secondary concern is delegation. ACL systems do not track modifications to an object's ACL. Without this feature, such systems cannot simultaneously support transitive delegation and revocation (requirements 3 and 4): a principal that can modify an ACL can increase its own authority and revoke *any* other principal's access to the object. Capabilities do not have this problem. However, to support the revocation requirements, the capability system must support *caretakers* [11] (proxies that indirect access to the object and thus enable fine-grained revocation). Neither *CapaFS* [12], *Tahoe* [18], nor *Secure FS* [6] support this.

Another secondary concern is the ability to work across administrative domains, i.e., with remote principals [9]. This requires global principals. *SFS* overcomes this by identifying users through user ids and self-certifying hostnames [7]. Capability systems using sparse capabilities solve this naturally. However, if capabilities are exposed to the user, it raises a usability concern: annotating a capability and handing it to the delegatee are separate steps. If the user does not rigorously perform both, e.g., by giving the same capability to multiple principals, the security monitor will not have an up-to-date view of the delegations, which will lead to confusion and mistakes.

View-Oriented Access Control: View-based access control has been used in the database community [5]. The idea is that a user grants permission to execute a database query statement, rather than to specific tables or columns. The resulting view is compact and enables access to data as it becomes available (requirement 10). This is the approach that we take. The only distributed file system, which we know of, that uses this technique is *CloudViews* [4]. It uses cryptographically signed views. We extend CloudViews by enabling modification of existing views without interaction with the delegatee and transitive delegation. We also address usability concerns for non-technical users and consider cross-cloud issues (CloudViews considers sharing only among services in the same cloud).

Powerbox: A powerbox, first introduced in *CapDesk* [16], runs with the user's authority and allows a service to display open and save-as dialog boxes that have access to all of the user's files, not just those that the service can access. Unlike for a Web server, for which it is often possible to know *a priori* exactly what authority it needs, the authority that a word processor needs is dynamic—it depends on the user's intents at the moment. To be consistent with POLP, a word processor should only have access to those files the user currently wants to edit. To allow a program to run with just that authority and yet still be able to display useful open and save-as dialog boxes, an application makes an RPC to the powerbox, which displays an open or save-as dialog box on its behalf but with the user's full authority. When the user selects a file, the powerbox returns a capability referencing the object to the application.

The powerbox has been integrated into *Secure FS* [6] to enable applications that run in the browser to access local files. They do not support a distributed file system or remote applications.

Malicious Storage Provider: *Tahoe* protects against a malicious storage provider by including an encryption key, which is never exposed to the storage provider, in the capability [18]. Since the storage provider does not know the real names of files or have access to file metadata, it appears this design may be incompatible with view-based access control.

5. S4

In this section, we describe a framework and API called S4 based on hierarchical, filtered views. We think that the model is easy for users to understand and that it satisfies our protection and security requirements. S4 extends Amazon S3's API and is mostly compatible with it. For instance, the same authentication mechanism is used (private key signatures), and the same interface for accessing objects. Given the limited space, we focus on our enhancements to the protection-related functionality.

Our S4 prototype consists of approximately 4,000 lines of Python (according to David A. Wheeler's SLOCCount). It supports all of the access control mechanisms described in this paper and includes much of the powerbox functionality.

S4 builds on S3's REST interface and is mostly backwards compatible with it (the only major missing piece is S3's ACL support). Services written to use this interface work with S4 without modification; taking advantage of the powerbox requires explicit support from Web services.

```
principal.create (pet_name) → (credentials)
principal.delete (child)
principal.list ()
    → ((credentials, pet_name, view...), ...)
view := <rights, filter...>
```

Table 1: Principal Management Methods

5.1 Principals

An S4 user first obtains an account from a storage provider. The storage provider gives the user authentication credentials for the account's primary principal, which has access to all of the user's storage and objects.

A user only shares his credentials with his security monitor and powerbox; other programs are given separate credentials and less authority. To do this, a user creates a subordinate principal by issuing the `principal.create` RPC to the storage server. This RPC causes the storage server to create a new principal that is subordinate to the principal that issued the RPC (thus, any principal can create subordinate principals). The method includes a so-called `pet name` parameter. This allows a user to associate a private, memorable identifier with the principal's public key, which is hard for a human to remember [17]; the storage server does not assign the pet name any semantic meaning. The server's response includes the credentials for the new principal, which the user passes to the service by copying and pasting them into a Web form.

The `principal.delete` RPC deletes a principal. Only a principal's parent may execute this RPC. When a principal is deleted, its authorization credentials are invalidated. Further, any principals subordinate to it are also removed; this RPC removes the subtree rooted at the principal.

A principal can enumerate its children using the `principal.list` RPC. This RPC is invoked on the principal and information about subordinate principals is returned. For each subordinate principal, its access key, its parent-assigned pet name, and any associated views are returned.

The principal management methods are shown in Table 1.

5.2 Hierarchical, Filtered Views

A principal can delegate access to a subordinate principal by associating a view with the principal. A view is a filter on the parent's name space and thus its authority. Specifically, it is a set of access rights and a set of regular expressions. Instead of regular expressions, Unix globs or SQL select statements could be used. Those files that are accessible to the parent and match all the regular expressions are made accessible to the child with the specified access rights. A view is installed using the `principal.delegate` RPC. It is possible to have more than one view associated with a principal. Initially, a subordinate principal has no authority: it has no associated views and can neither access nor create any objects.

Access control is based on hierarchical evaluation and filtering, which enables creating increasingly restricted views of a name space. The following algorithm is used for access checks: for each view, determine if the access rights include the requested access type and if the name matches all of the regular expressions. If no view satisfies this test, access is denied. Otherwise, the same procedure is carried out using each of the principal's ancestors. If all of the principal's ancestors may access the object with the requested permis-

```
principal.delegate (child, view...)
principal.revoke (child, view...)
```

Table 2: View Management Methods

sion, access is granted. The intuition is that a parent can only grant as much access as it has, not more. Thus, when a parent indicates that a child can access all objects, this means that the child can access all objects that the parent can access, not that the child can access all files in the file system, which is access that the parent cannot grant.

A possible issue with this design is that it does not allow cross-hierarchy delegations. For instance, Alice cannot directly delegate access to Bob's principal. Instead, she must create a new Alice-Bob principal. We do not think that this is a limitation in practice; we expect most sharing to do be done via the Web services. Note that since principals are represented to users as strings, it is possible to make the string a URL (similar to a webkey [3]), which when entered in a browser loads an interface to browse the accessible files.

Permission checks can be optimized. Whether using regular expressions as we do in S4, Unix globs or SQL query statements, it should be possible to parallelize their evaluation. Also, it should be possible to cache compiled forms near the data structure associated with the principal. Indeed, all filters can be combined thereby avoiding having to walk the tree. The trade-off in this case is that when a principal's views are updated, it, as well as all of its descendants, must have their cached compilations invalidated. We expect that this is infrequent relative to permission checks.

To revoke access, a principal uses the `principal.revoke` method. The indicated view must match an existing view exactly; views cannot be modified in place. To modify a view, it must be revoked and a new view must be installed. This is not atomic, however, if atomicity is required, a `principal.filter_replace` method could be provided.

The view management are summarized in Table 2.

Describing Filters: We do not expect most users to describe policies using regular expressions. Instead, users interact with the security monitor or powerbox, which presents a graphical user interface. The GUI allows users to describe their policy using graphical elements (e.g., a file selector), which the monitor translates to an appropriate regular expression.

Visualizing Filters: Figure 3 shows a screenshot of a security manager, which we have implemented. The manager is a user interface for managing principals and access rights. To better enable the user to understand what a principal may or may not access, when the user selects a principal or a regular expression, the security manager highlights the accessible objects. We use red to mean that the principal has no access, green to mean that the principal has some access, and yellow to mean that it may access one or more objects below the directory. Our implementation does not currently provide a way to distinguish between read and write access. This could be done using an icon. The interface shows only the files that a principal can access; it does not show files that a principal may create or that could become accessible. While this would be valuable, it is unclear how to display this information.

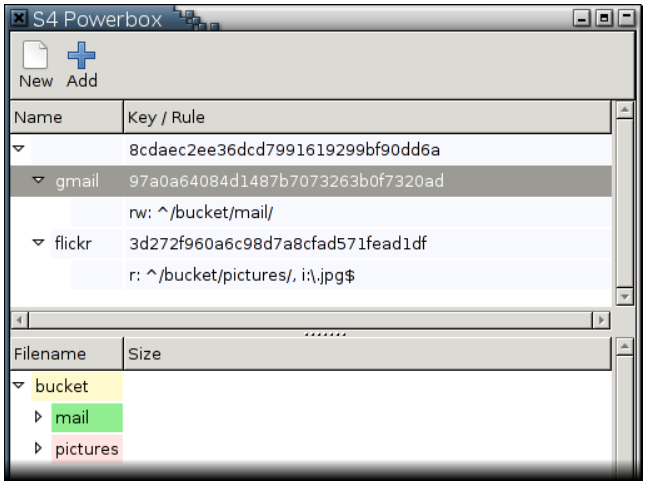

Figure 3: Screenshot of the security manager. The user has two subordinate principals. The gmail principal is selected. Objects that it may access are in green (here, `mail`), those that is may not are in red (`pictures`). Yellow indicates directories which contain some accessible files (`bucket`).

5.3 Powerbox

We integrated the powerbox paradigm into our system as follows. When a user connects to the Internet, his powerbox registers with his storage provider using the `principal.powerbox_register` RPC. When a service requires the use of the powerbox, for instance, when the user clicks on an open or save-as icon, the service invokes the `principal.powerbox_invoke` RPC. The storage server forwards this request using the `principal.powerbox_invoke` upcall to the powerbox clients associated with the nearest ancestor principal that has at least one registered powerbox. When the powerbox receives this message, it displays an appropriate dialog box, which includes the requesting principal's pet name (which helps prevent a service from spoofing its identity) and the service's message.

Multiple powerbox clients may be registered simultaneously. This is useful when a user has multiple devices that may be simultaneously connected to the Internet. In this case, all devices show the dialog box. When one powerbox responds, the storage provider issues the `principal.powerbox_close` upcall to all of the other powerbox instances indicating that they should close their corresponding dialog. Multiple requests can be extant at any given time. The `handle` parameter allows the parties to differentiate the invocations.

In the case where there is no powerbox registered, the storage server replies to the service with an error message to prevent it from waiting forever. It may be that the user forgot to start the powerbox client. In this case, the service can display a message to the user indicating that to proceed, the powerbox application must be started.

The powerbox methods are summarized in Table 3.

5.4 Updating Shared State

Given that ad hoc sharing is the norm in S4, a mechanism is needed for ensuring that updates are applied atomically. S4 clients can reuse S3's mechanism for updating shared

Client:
  `principal.powerbox_register ()`
  `principal.powerbox_reply (handle, name)`
Service:
  `principal.powerbox_invoke (msg, options)` → `handle`
Service and Storage Provider:
  `principal.powerbox_close (handle)`

Table 3: Powerbox Methods

state: when storing a file in S3, the store can be made conditional on the current content's MD5 hash. A service can ensure atomic updates by reading the current version, applying updates, and then performing a conditional store. If the conditional store fails, the service can retry. This approach guarantees atomic updates for a single file. It can, however, result in starvation.

6. DISCUSSION

Scenarios: To determine whether our design meets our requirements, we return to our example scenarios. We assume that Alice and Bob's services have been updated to use S4 and that Alice and Bob have migrated their data to personal storage providers.

When Alice or Bob wants to give a service access to their respective storage repository, they log in to their security manager and create a new principal for that service. The security monitor returns a password capability (an unguessable string), which Alice or Bob copy and paste into a dialog the service displays. This string contains all of the information the service needs to log in and access the user's storage and to use the powerbox. Initially, a principal has no authority to access the user's storage. Authority can be bootstrapped using the powerbox. For instance, Hotmail may use the powerbox to ask for access to the user's address book, and Flickr may ask for access to files matching the pattern `*.jpg`. In the latter case, the user may add the additional limitation that Flickr is authorized to access files under a certain sub-directory.

One of Alice's desires was to be able to update her profile picture in one place and have all services that use a profile picture automatically start using that picture. To support this, such services can be changed to use a user-specified file, which has a reasonable default. Alice then needs to authorize the relevant services to access this object. This needs to be done just once for each service. Again, this can be done by way of the powerbox. Alternatively, Alice could interact with the security monitor. Either way, when Alice changes the file, services will start using the file when they notice that the file has changed.

Alice also wants a single global address book. Again, we can imagine that all services either use a single file or a directory with a single file for each entry. A standard default location should again be established. Access can again be authorized using the powerbox. From then on, each service monitors the address book for changes, and automatically saves changes there.

We now turn to Bob, who wants to be able to access his photos on Flickr from Google Docs. Bob can now upload his files directly to his personal storage repository and enable Flickr to access them immediately and Google Docs on a need-to-use basis by way of the powerbox. Note that this requires no more interactions with the program than opening a file in a program that has access to all of the user's files. Our use of regular expressions on the file name does not enable Bob to perform access control based on tags and thus he is unable to only grant Flickr access to those files tagged as being public without renaming them. He does not have to move the files to a different directory, however: he can just add a differentiating token to the file name.

Another of Bob's concerns is version drift. Since both Google Docs and Flickr refer to his store for authoritative copies of data, all updates are made to the authoritative copy. Version drift cannot occur.

Finally, Bob wants to be able to send documents he worked on in Google Docs from Hotmail without first having to download the documents. Since Hotmail and Google Docs both use his store for accessing his data, this is not a problem. Moreover, it can again be done in a manner consistent with POLP and without any explicit interactions with the security monitor.

Now, consider the case where Alice decides to stop using a social networking service. Alice can prevent that service from further accessing her data by entering the security monitor and deleting the principal. There is no way to stop the service from using copies of data that it cached, but it cannot access subsequent updates.

Business Incentives: It is questionable whether established players, such as Facebook, would want to support a system such as S4: they would relinquish control of data. Nevertheless, we think that users are interested in solving the problems that we have identified and might migrate to a service that gives them more control over their data. Further, the possibility to better control data might extend the market to privacy-sensitive individuals. Alternatively, a company such as Facebook might be interested in becoming a storage provider in the S4 model, as this would allow them to potentially increase their importance as a focal point of user activity and thereby reap the benefits of being a hub.

7. FUTURE WORK

We identify four areas of future work: richer filters, protection from undesired changes, a push mechanism for propagating changes, and protection against financial attacks.

S4 applies filters to the file name, however, many files, such as photos, contain tags. Filtering based on tags is provided by many services, so it is well understood. It would allow Bob to more easily articulate his policy that only photos tagged `public` should be made available to Flickr. As tags are harder than file names to see, care must be taken particularly with tags that the user did not set; a tag's provenance should be considered.

Protection from changes from malicious or misbehaving services can be achieved using copy-on-write. Using such a mechanism, a user would enable copy-on-write for untrusted principals in his security monitor and the storage provider would not immediately propagate changes to the authoritative copy. Instead, after examining the changes, the user would manually promote changes. This has the disadvantage that the feature must be enabled before the damage occurs and requires constant user maintenance. Instead, periodic snapshots can be made, which only require user intervention when something goes wrong. This has the disadvantage that changes may need to be disentangled. This

mechanism also provides automatic backups.

Scanning files for changes can be bandwidth intensive and can add latency. A push mechanism solves both of these problems. This could be implemented as a simple per-service log, which the storage provider periodically pushes. The aggressiveness of the pushes could be made a function of the service's and the storage provider's latency requirements.

Finally, since bandwidth is not free, a mechanism needs to be provided to prevent financial attacks. This is particularly important because unlike programs that use a lot of CPU or network locally, there is no immediate feedback that a service is generating a lot of traffic. A solution is to enforce monetary allowances. If a service exceeded its allowance, access could be denied or rate limited, and the user could be informed using the powerbox or via email.

8. CONCLUSION

We have defined requirements for protection, usability and sharing in subsection 3.4 and built a system that meets these requirements. We focused on fine-grained delegation and usability in the sense that users should be able to concisely and directly express their desired security policies and these should require minimal maintenance. Our framework is based on hierarchical, filtered views. This enables users to concisely express cross-cutting concerns, such as delegating access to all JPEG files. This not only delegates access to all JPEG files independent of their location but also new JPEG files as they are added. To reduce the number of user interactions with the security monitor, we integrated a powerbox into our design, a mechanism that allows users to select a file in an open or save-as dialog box using their authority and not the more limited authority of the application.

9. REFERENCES

[1] Amazon web services simple storage service. `http://aws.amazon.com/s3/`, 2010.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb. 2009.

[3] T. Close. Web-key: Mashing with permission. In *Proceedings of Web 2.0 Security and Privacy*, Beijing, China, Apr. 2008.

[4] R. Geambasu, S. D. Gribble, and H. M. Levy. CloudViews: Communal data sharing in public clouds. In *Hot Cloud '09*, June 2009.

[5] P. P. Griffiths and B. W. Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems*, 1(3):242–255, 1976.

[6] F. Hsu and H. Chen. Secure file system services for Web 2.0 applications. In *CCSW '09*, pages 11–18, 2009.

[7] M. Kaminsky, G. Savvides, D. Mazières, and M. F. Kaashoek. Decentralized user authentication in a global file system. In *Proceedings of SOSP '03*, pages 60–73, October 2003.

[8] B. W. Lampson. Protection. *SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[9] S. Miltchev, J. M. Smith, V. Prevelakis, A. Keromytis, and S. Ioannidis. Decentralized access control in distributed file systems. *ACM Computing Surveys*, 40(3):1–30, 2008.

[10] Nirvanix Storage Delivery Network. `http://www.nirvanix.com/`, 2010.

[11] D. D. Redell. *Naming and Protection in Extendable Operating Systems*. PhD thesis, MIT, Cambridge, MA, USA, 1974.

[12] J. T. Regan and C. D. Jensen. Capability file names: separating authorisation from user management in an internet file system. In *Proceedings of USENIX Security Symposium '01*, pages 17–17, 2001.

[13] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, Sept. 1975.

[14] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computing Systems*, 7(3):247–280, 1989.

[15] M. Stiegler, A. H. Karp, K.-P. Yee, T. Close, and M. S. Miller. Polaris: virus-safe computing for windows xp. *Commun. ACM*, 49(9):83–88, 2006.

[16] M. Stiegler and M. Miller. A capability based client: The DarpaBrowser. Technical Report BAA-00-06-SNK, Combex, Inc., 2002.

[17] Z. Wilcox-O'Hearn. Names: Decentralized, secure, human-meaningful: Choose two. `http://zooko.com/distnames.html`, Sept. 2003.

[18] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proceedings of StorageSS '08*, pages 21–26, 2008.