Confidant: Protecting OSN Data without Locking It Up

Dongtao Liu[1], Amre Shakimov[1], Ramón Cáceres[2],
Alexander Varshavsky[2], and Landon P. Cox[1]

[1] Duke University
[2] AT&T Labs

Abstract. Online social networks (OSNs) are immensely popular, but participants are increasingly uneasy with centralized services' handling of user data. Decentralized OSNs offer the potential to address user's anxiety while also enhancing the features and scalability offered by existing, centralized services. In this paper, we present Confidant, a decentralized OSN designed to support a scalable application framework for OSN data without compromising users' privacy. Confidant replicates a user's data on servers controlled by her friends. Because data is stored on trusted servers, Confidant allows application code to run directly on these storage servers. To manage access-control policies under weakly-consistent replication, Confidant eliminates write conflicts through a lightweight cloud-based state manager and through a simple mechanism for updating the bindings between access policies and replicated data.

Keywords: Decentralization, Onlie Social Networks, Peer-to-peer, Cloud.

1 Introduction

Online social networks (OSNs) such as Facebook, MySpace, and Twitter have enhanced the lives of millions of users worldwide. Facebook has surpassed 700 million active users per month and already attracts 32% of global Internet users every day, with the average user spending 32 minutes each day on the site [8, 9]. The aggregate volume of personal data shared through Facebook is staggering: across all users, Facebook receives nearly 30 billion new items each month. Such high levels of participation should not be surprising: OSNs are fun, useful, and free.

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

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

Several decentralized OSN architectures have been proposed [1, 16, 22] that assume the servers where OSN data is stored are untrusted: data is encrypted before it is stored and plaintext is only viewable by client machines with the appropriate decryption keys. The appeal of this approach is that encrypted data can be stored anywhere, including peer-to-peer DHTs, cloud services such Amazon S3, or even existing OSNs such as Facebook.

However, OSNs have also evolved into large-scale platforms for third-party applications, and we observe that decentralized OSNs that rely on untrusted storage fundamentally limit the kinds of applications that can be built on top of an OSN: if storage servers cannot be trusted with plaintext data, application code that accesses OSN data can execute only on trusted clients. In the worst case, application code must download all relevant data to a client machine, and then decrypt and operate on the data locally. For mobile and desktop clients alike, this can lead to bandwidth, storage, and compute scalability problems.

A partial solution is for OSN designers to anticipate in advance how applications might want to use users' data and require clients to maintain an index of pre-defined features such as key words over the encrypted data [6, 7, 16, 20]. Unfortunately, these techniques limit applications to searching over those pre-defined features, and cannot scalably support richer interactions with the data such as trend spotting or image-based search.

The central question of this paper is: how can decentralized OSNs support a scalable, general-purpose application framework? To answer this question we present the design and implementation of *Confidant*. Confidant's approach to decentralized OSNs is to use information from the social graph to store users' data in plaintext on machines that they trust. The intuition behind our approach is that since a user's friends' machines already have read access to her OSN data, why not allow them to serve her data as well?

Confidant participants use commodity machines such as personal desktop PCs or enterprise workstations as storage servers, and create replicas on machines controlled by a small subset of their friends (e.g., ten friends). Trust relationships between users can be exposed to Confidant in many ways, such as by mining the social graph of existing OSNs (e.g., the level of interaction between users or the number of overlapping friends), or by asking users to manually identify friends to host their data. Once a user has selected her replicas, an application running on behalf of a Confidant user may be authorized to remotely execute sandboxed scripts on these trusted machines.

It is important to note that serving OSN data from personal machines does not introduce new risks to data confidentiality. Centralized and decentralized OSNs alike must store credentials on personal machines (e.g., web cookies, OAuth tokens, or cryptographic keys), and if an attacker compromises a personal machine it will have access to any of the data authorized by the credentials stored on that machine. At the same time, as with other decentralized OSNs, users can feel safe knowing that any data entrusted to Confidant will not be leaked by a centralized OSN behind their back.

The main challenge for Confidant is managing OSN data and access-control policies under weakly-consistent replication. As in prior work [17, 27], Confidant relies on eventual consistency among replicas and treats objects and access policies as immutable first-class data items. However, managing data shared within an OSN presents a different set of challenges than those addressed previously. First, OSN users commonly create new OSN data from multiple clients and should not have to deal with the inconvenience of manually resolving conflicts. Confidant eliminates conflicts without compromising data confidentiality or integrity by serializing a user's updates through a highly available and lightweight state manager hosted in the cloud. In addition, OSN users who inadvertently mis-share an item must be allowed to recover from their mistake without perturbing the policies protecting other data items. Confidant addresses this issue through flexible rebinding of access policies to data items.

To summarize, this paper makes the following contributions. Confidant's design represents the first decentralized OSN architecture to leverage trustworthy storage servers based on inter-personal relationships. This design choice allows Confidant to support a scalable, general-purpose application framework without relying on a centralized OSN to manage users' data. Confidant also provides an access-control scheme for weakly-consistent, replicated data that is tailored to the needs of OSN users. In particular, Confidant eliminates write conflicts and allows participants to recover from access-control mistakes by binding access policies to individual data items rather than to groups of items.

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

The rest of this paper is organized as follows: Section 2 gives a high-level overview of the Confidant architecture, Section 3 describes Confidant's design, Section 4 describes our prototype implementation, Section 5 presents an evaluation of our design and prototype implementation, Section 6 describes related work, and Section 7 provides our conclusions.

2 Overview

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

2.1 Architecture

Figure 1 shows the Confidant architecture, including paths for posting new data (steps 1-2), retrieving new data (steps 3-5), and running Confidant applications (steps 6-8). Keeping costs low is a critical design goal for Confidant, since existing OSNs such as Facebook are popular in large part because they are free. As a result, Confidant relies

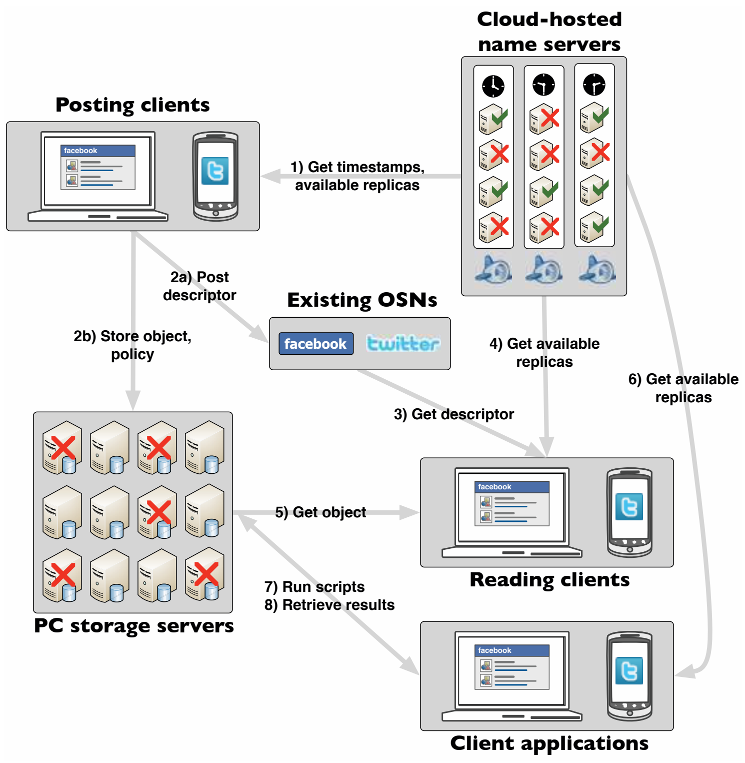


Fig. 1. Confidant architecture

on two low-cost forms of infrastructure: desktop and enterprise PC *storage servers*, and lightweight cloud-based *name servers*. PCs and workstations are a sunk cost for most users and free cloud services such as Google AppEngine and Heroku allow a user to execute a single-threaded server process while maintaining a small amount of persistent state. We assume that every Confidant user controls both a storage server and a name server.

A storage server hosts a user's OSN data and authorizes client read and write requests. Because storage servers can experience transient failures, a user may select a small number of her friends' servers to host *replicas* of her data. Each replica manages a copy of the user's data and participates in anti-entropy protocols to ensure eventual consistency. Storage servers also allow authorized users to run sandboxed application *scripts* directly on their hardware.

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

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

2.2 Trust and Threat Model

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

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

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

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

However, even if a user is trusted to preserve the confidentiality of their friend's data, their storage server might not be trusted to preserve the integrity of application scripts. A compromised storage server can corrupt script results by modifying a script's execution, injecting false data, or removing legitimate data. To reduce the likelihood of corrupted script results, we assume that users can identify a small number of friends whose storage servers will behave as expected. Based on several proposals to gauge the strength of social ties using the level of interaction between OSN users [3, 10, 25], it is reasonable to assume that users will find enough storage servers to act as replicas. For example, Facebook has reported that male users regularly interact with an average of seven friends, while women regularly interact with an average of ten friends [21].

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

3 Design

In this section, we describe the Confidant design.

3.1 Cryptographic State

Confidant encodes component roles and trust relationships using techniques described by work on Attribute-Based Access Control (ABAC) [26]. We do not support the full power of an ABAC system, and have adopted a subset of techniques (e.g., independently rooted certificate chains and signed attribute-assignments) that are appropriate to our decentralized OSN.

Principals in Confidant are defined by a public-key pair, and *users* are defined by a public-key pair called a *root key pair*. Users generate their own root keys, and distribute their root public key out of band (e.g., through their Facebook profile or via email). A user's root public key is distributed as a self-signed certificate called a *root certificate*; the root private key is kept in a secure, offline location. Through her root key-pair, a user also issues certificates for her storage server (*storage certificate*), name server (*name certificate*), and any clients under her control (*client certificate*): these certificates describe the principal's role (i.e., storage server, name server, or client) and its public key. For each certificate signed by a user's root key pair, the matching private key is only stored on the component, and expiration dates are set to an appropriate period of time. Users also generate *replica certificates* with their root key pair for any storage servers controlled by others who are authorized to serve their objects. All certificates are distributed out of band through a service such as Facebook or via email.

Users encode their inter-personal relationships through *groups*. A group is defined by four pieces of state: 1) a unique user who owns the group, 2) a list of users making up the group's membership, 3) an attribute string, and 4) a secret key. Group owners are the only users who can update a group's state. Group memberships grow monotonically; "removing" a member requires an owner to create a new group with the previous membership minus the evicted member.

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

Group keys are generated on the group owner's storage server and distributed as social attestations [22]; attestations are signed using the key pair of the owner's storage server. Social attestations in Confidant are nearly identical to those in Lockr, except that Confidant attestations also enumerate their group's membership. Within a social attestation, group members are represented by the string description and public key found in their root certificate. If new members are added to the group, the group owner distributes a new social attestation to members reflecting the larger group size. New and updated attestations are distributed epidemically among storage servers; clients periodically synchronize their set of attestations with their owner's storage server. Although there is no bound on the time for a client to receive an attestation, the key embedded in an existing attestation can remain valid even if the membership enumerated in the attestation becomes stale.

Access policies specify the groups that are allowed to view an object, and are represented by logical expressions in disjunctive normal form (e.g., $(g_0 \wedge g_1) \vee (g_2 \wedge g_3)$), where each literal describes a group, and each conjunction indicates a set of group keys that could be used for authorization.

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

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

3.2 Objects and Access Policies

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

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

$$\{owner, seq, acl\}$$

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

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

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

$$\{owner, seq, typ, t, len, \langle data \rangle\}$$

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

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

Like objects, access policies are immutable, although the binding between objects and polices can change according to the following rule: *an object is protected by the policy with the greatest sequence number that refers to the object*. Application of this rule allows clients to add and remove permissions using a single, simple mechanism. If a client wants to bind a new access policy to an old object, it increments its user's logical clock and creates a new policy using the new sequence number. To invalidate an object, a client can issue a new access policy with a null expression. If two objects are protected by the same logical expression, they will require separate access policies. Note that because the binding between objects and policies can change the acl included in an object descriptor is meant only as a hint [12], and may not reflect the current protection scheme for the object.

There are two potential drawbacks of not aggregating policies across objects: the overhead of storing and transferring extra policy items, and the added complexity of bulk policy changes. However, both drawbacks are minor concerns, given the relatively small number of items that individual users are likely to generate. According to Facebook, the average user creates only 70 data items every month [9]. Even for bulk policy rebindings covering years' worth of user data, iterating through all of a user's items several times should be reasonably fast. As a result, the flexibility to bind arbitrary policy expressions to items at any point in the item's lifetime outweigh the drawbacks.

Confidant's approach to object protection is similar to recent work on managing access policies in Cimbiosys [17, 27]. Both systems manage data as a weakly-consistent replicated state store, and both treat objects and policies as first-class data items. Despite the similarities, there are several important differences between Confidant and Cimbiosys.

First, Confidant serializes all of a user's updates by assigning new items a unique sequence number from the user's name server. This eliminates the complexity and inconvenience of automatically or manually resolving concurrent updates. Avoiding the pain of handling conflicts is an important consideration for OSNs. Experience with the Coda file system found that most write conflicts were caused by users updating their data from multiple clients [14], which mirrors the common behavior of OSN users accessing services from both a PC and mobile device. Confidant's name servers create a single point of failure, but we believe that this is an appropriate tradeoff given inconvenience of handling conflicts and the high availability of cloud services such as Google AppEngine.

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

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

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

3.3 Name Servers

As described in Section 2.1, each user runs a name server within a low-cost cloud service such as Google AppEngine. Name servers manage two pieces of state: a list of IP addresses corresponding to online replicas and a logical clock. Entries in the list of IP addresses also include an expiration time, and become invalid if not updated in time. Name servers also maintain a list of storage servers authorized to act as replicas.

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

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

Table 1. Storage-server messages

Message	Format
Store request	$\{\{g_0, g_1, \ldots g_n\}, cert_C, \{replicas, obj, ap, rand, \{hash(obj, ap) rand\}_{K_C^-}\}_{gk_0, gk_1, \ldots gk_n}\}$
Policy update	$\{cert_C, replicas, ap, rand, \{hash(ap), rand\}_{K_C^-}\}$
Retrieve request	$\{owner, seq, \{g_0, g_1, \ldots g_n\}\}$
Retrieve response 1	$\{\{g_0, g_1, \ldots g_n\}, cert_R, \{obj, ap, rand, \{hash(obj, ap), rand\}_{K_R^-}\}_{gk_0, gk_1, \ldots gk_n}\}$
Retrieve response 2	$\{cert_R, ap, rand, \{hash(ap), rand\}_{K_R^-}\}$

using their signed certificates, but once the session key has been established it is used to authenticate future requests.

A name server's logical clock is used to assign sequence numbers to items and to help replicas synchronize when they come back online. The value of the logical clock increases monotonically. When a client wants to assign a sequence number to a new item, it requests an increment; the name server responds by adding one to the existing value and returning the new value of the clock. Since only clients under the user's control should be allowed to advance the clock, increment requests are authenticated. As with entry-refresh requests, clients and name servers initially establish a session key with their signed certificates, and use the session keys to authenticate future requests.

3.4 Storage Servers

Each Confidant user runs a storage server that contains plaintext copies of all of her objects and access policies. A storage server may also act as a *replica* for another user if the other user trusts the server's owner to 1) read all of her objects, 2) enforce access policies, and 3) preserve the integrity of any application scripts run on the server. As explained in Section 2.2, it is reasonable to assume that users can identify on the order of ten trustworthy replicas.

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

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

Consistency. We apply an eventual consistency model to data stored within a replica set and rely on epidemic propagation to synchronize storage servers [5, 11]. Because objects and access policies are immutable, ensuring a consistent ordering of updates across replicas is not material. We are only concerned with whether the set of data items stored on behalf of a data owner is consistent with the set of all items the data owner has created. Servers achieve eventual consistency by applying standard anti-entropy techniques, which are described in greater detail in the Confidant Technical Report [13].

Updating and retrieving items. The messages used to update and retrieve items are listed in Table 1.

To add a new object, a client first retrieves a list of online replicas and two new sequence numbers (one for the object and one for the object's access policy). To store the new items, a client connects to the first server in the list returned by the name server and submits the store-request message described in Table 1. The header corresponds to a conjunction, $\{g_0, g_1, \ldots g_n\}$, from the the access policy ap; this indicates which group keys, $gk_0, gk_1, \ldots gk_n$, are used to protect the message in transit. The client also sends the replica its client certificate, $cert_C$.

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

Once a server has unpacked and verified a store request, it commits the new items to its local database and returns control to the client. The storage server is now responsible for propagating the update to the other replicas listed in the $replicas$ field of the message payload. We rely on anti-entropy to spread new items to the rest of the replica set in the face of network partitions and server failures. Storage servers' local database and the protocol for authorizing retrieve requests are described in more detail in the Confidant Technical Report [13].

Application framework. Our primary motivation for leveraging social relationships to select replicas is to enable scalable, general-purpose applications without sacrificing data confidentiality. Prior decentralized OSNs assumed that storage servers were not trusted to view plaintext data, which limited the class of operations that storage servers could perform on OSN data to feature-based searches (e.g., key-word [6, 7, 20] or location-based [16] search). Unfortunately, many popular and emerging OSN applications such as Twitter's trending topics and face.com's face-recognition service require iterating over a large corpus of plaintext data. Services such as these require a general-purpose distributed programming framework like MapReduce [4]. However, unless storage servers are trusted to view plaintext data, such applications can only be implemented by downloading an entire encrypted corpus to a client, where it must be decrypted and analyzed. This approach will not scale for clients executing on resource-limited desktop PCs and mobile devices.

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

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

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

4 Implementation

We have implemented a Confidant prototype based on the design described in Section 3. This section describes our client, name server, and storage server implementations. We also describe three applications that we have implemented on top of Confidant.

4.1 Client

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

We derive several benefits from interoperating with Facebook. One, users continue using their existing Facebook accounts, thus leveraging their considerable investment in creating social connections there and learning how to use the many popular features. Two, we take advantage of Facebook as a reliable medium for storing object descriptors and distributing them throughout the social graph. For more details on our client implementation, please see the Confidant Technical Report [13].

Facebook remains an untrusted service that should not have access to users' secret keys or sensitive data. Our browser extension modifies Facebook's default behavior by listening for browser events such as document loads, form submits, and button clicks.

When these events happen, our handling functions are triggered to run prior to Facebook's original functionality. For example, when a user wants to share a status update or post a wall message, the browser extension intercepts the control flow. Once in control, it contacts the user's Confidant name server to retrieve sequence numbers for the new object and policy as well as the IP address of the available replicas for storing these items. It then creates an object and policy with the appropriate descriptor and sends the items to a replica. Finally, the extension substitutes the original content from the Facebook page with the object descriptor to be sent to Facebook.

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

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

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

4.2 Name Server

As described in Section 3.3, each user runs a lightweight name server that maintains a list of available replicas. We host this server in the Google AppEngine. AppEngine is a highly available and low-cost cloud-computing infrastructure where users can run applications written in several languages; we used Python. Google does not charge for running applications until they exceed a free-resource quota. We engineered our name server so that its resource utilization should remain comfortably within AppEngine's free quota.

4.3 Storage Server

As described in Section 3.4, each user also runs a storage server with multiple roles: it maintains the primary copy of the user's sensitive data; it maintains a replica of the data belonging to a subset of the user's friends; it arbitrates and serves requests for this data; and it runs application scripts submitted for execution by the user's friends.

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

4.4 Applications

As described in Section 3.4, Confidant allows application scripts to execute on the distributed collection of storage servers. We have implemented the three representative applications described below:

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

Trending-Topics Script: The *trending-topics script* calculates the most frequently used words within a user's friends' status updates. This script prepares a dictionary object $[''word1'' : ''count'', ''word2'' : ''count'', ...]$ using the available status updates and wall messages on each storage server. We eliminate common words that should not be interpreted as trends such as *"I"*, *"me"*, *"am"*, and *"you"*. A client downloads these pre-processed objects from their friends' replicas and merges them into a single list. The client then sorts the list by value to produce the most trending keywords.

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

4.5 Untrusted Storage Server

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

5 Evaluation

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

- How does the performance of our application framework compare to approaches that rely on untrusted storage servers?
- How many trusted friends will Confidant users need to ensure that at least one replica is always available?

Note that the Confidant Technical Report also describes the results from experiments measuring the latency of creating and retrieving photo and text objects [13]. We have omitted these results to focus on scalability and availability.

5.1 Application Performance

In this section, we compare the performance of Confidant's application framework to the performance of approaches that rely on untrusted storage servers. We measured the end-to-end latencies of the face-detection, keyword-search, and trending-topics scripts described in Section 4.4 and compared them to the latencies of fetching data encrypted under AES from an untrusted server and processing it locally. We refer to the latter approach as *Encrypted*.

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

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

To populate the remote servers for our trending-topics and keyword-search experiments, we used status updates collected from Facebook by researchers at UCSB [18]. We used images from publicly available datasets from computer vision research groups at CMU, MIT and Cal Tech for evaluating the face-detection script's performance. Our image corpus contained 70 images altogether. We tuned the parameters of the face-detection algorithm so that it was reasonably accurate and required about 500 ms to process a 30Kb picture. It should be noted that we did not evaluate the accuracy of our face detection algorithm and, thus, we do not report how many of the returned pictures were false-positives. The average size of a picture used for the evaluation is 30 KB, with sizes ranging from 8Kb to 200Kb.

Confidant achieves compute scalability by off-loading computational tasks to trusted remote servers, while the Encrypted approach must first download data to a client and decrypt it before processing it locally. To isolate the effect of computational scalability on end-to-end latency, during our experiments the Confidant clients and Encrypted clients only differed in how much processing they performed rather than the volume of data they received. For the keyword-search experiments, we partitioned 1,000 status updates across the remote storage servers. For the trending-topics experiments, remote Confidant scripts collectively returned dictionaries based on 1,000 status updates to the client. For the face-detection experiments, we assigned each remote server 10 photos. Finally, for our experiments, we conservatively assumed that each of the client's friends had only one replica available.

Figure 2 shows the average over 20 trials of each script's latency under Confidant divided by the latency to perform the same task under the Encrypted scheme. Please note the logarithmic scale of the y-axis. Standard deviations for all averages were less than 2%. For 100 friends, Confidant was between 3 and 30 times faster (trending topics and face detection, respectively). While all three scripts performed better under Confidant, the face-detection script benefited the most from parallel remote execution due to the computational complexity of detecting faces in images. Trending topics

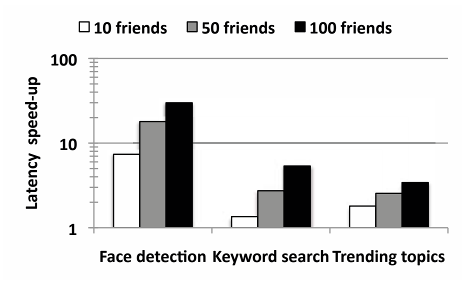


Fig. 2. Script performance

benefits the least due to the computational complexity of merging dictionaries from many sources that must be performed at the client. Nonetheless, these results demonstrate the performance and scalability benefits of Confidant's application framework.

5.2 Availability

Confidant relies on replicated storage servers to ensure that clients can successfully submit and retrieve items. The availability of a user's replica set is a function of two properties: the number of trusted friends a user has (i.e., the size of the replica set), and the availability of individual replica-set members.To explore the impact of these properties on the rates at which clients can successfully read and write, we simulated Confidant using two types of traces. To characterize how many friends users have and how they interact with their friends,we used a trace of the Facebook graph and wall messages collected at MPI [24]. This trace captures links and interactions between users and consists of approximately 60,000 users, connected by over 0.8 million links, with an average node degree of 25.6.

To characterize storage-server availability, we used the well-known Gnutella trace [19] and Microsoft-workstation trace [2]. The Microsoft-workstation trace is much more forgiving and machines have a much higher average online time than hosts in the Gnutella trace. It should be noted that there were many IP addresses listed in the Gnutella trace that were never online, and we removed all such nodes. This left us with approximately 15,000 hosts over 60 hours. From the Microsoft trace, we used roughly 52,000 hosts' records over 35 days.

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

We included the most-active Facebook users from the MPI traces because they generated the most write events for our simulation. Of course, there is a correlation between activity and connectedness, as our increased average node degree demonstrates. This correlation biases in our favor since the resulting graph includes a larger fraction of highly-connected users. However, even with the higher average node degree of 44.8, this is still substantially lower than the average of 130 reported by Facebook [9].

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

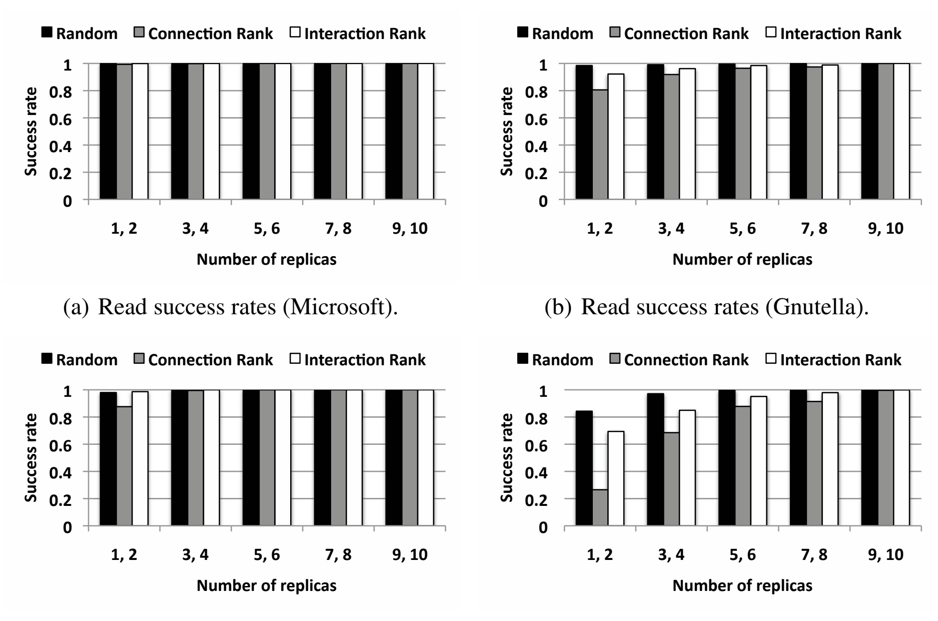
We assumed that reads and writes were generated by an always-available client such as a mobile device. We simulated writes using interactions from the MPI trace. For a write event, if at least one member of a client's replica set was available, the write was considered a success and was added to a read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

Reads were not captured in the MPI trace so we generated them synthetically. Clients chose a random time between 30 minutes and 60 minutes, and after that time passed attempted to retrieve the objects in their read queue. This read rate is consistent with the reported behavior of Facebook users [9]. Read failures occurred if a client tried to read an object while no member of the replica set was available, or none of the online replica servers had the object at that time. Otherwise, the read was considered a success. After clearing its read queue, clients chose another random period before they read again.

Figure 3(a) and Figure 3(c) show the read and write success rates for simulations using the Microsoft availability trace, while Figure 3(b) and Figure 3(d) show the read and write success rates using the Gnutella trace. As expected, nodes with more replicas had higher read and write success rates, regardless of how Facebook users were assigned host availability records. The overall higher read success rates across experiments are partially an artifact of failed write attempts being suppressed in the read results; if a node failed to perform a write, no other nodes attempted to read it.

One of the most striking features of our results is how much worse nodes with fewer replicas fared when low connectivity was correlated with low availability. This is because nodes with low connectivity were penalized twice. Not only did these users have access to fewer replicas, but their own server was assigned low availability as well. This combination of factors led to significantly lower success rates than for nodes with low connectivity under the other schemes or nodes with more replicas.

Unsurprisingly, read and write success rates under the Gnutella trace are much worse than under the Microsoft-workstation trace. The Gnutella trace likely provides a close to worst-case scenario for host availability since it captures client-process uptime rather than machine uptime. On the other hand, results from the Microsoft-workstation trace are likely close to a best-case scenario since the trace captures machine uptime in a



(a) Read success rates (Microsoft). (b) Read success rates (Gnutella).

(c) Write success rates (Microsoft). (d) Write success rates (Gnutella).

Fig. 3. Simulation results

corporate environment. Reality likely lies somewhere in between. It is important to note that even under the less-forgiving availability of the Gnutella trace, users with ten replicas fared well. The worst case for ten replicas was under a connection-based mapping, which generated a write-success rate of 99.6%. Under the Microsoft-workstation trace, three or four replicas provided a write-success rate of 99.5% using the connection-based mapping. Users with more than four replicas under the Microsoft trace had perfect read and write rates for all mappings. All of these results bode well for Confidant. Facebook users have an average of 130 friends, and we suspect that it would be straightforward for most users to identify 5-10 trusted friends who are willing to host their data.

6 Related Work

FriendStore [23] is a backup system that also identifies trustworthy storage sites through inter-personal relationships. FriendStore leverages trustworthy remote storage servers to mitigate long-standing fairness problems in peer-to-peer systems. A primary difference between Confidant and FriendStore is the nature of the data they must manage. Backup data is not meant to be shared and thus FriendStore does not require the same level of complexity to manage access-control policies that Confidant does.

While Confidant leverages knowledge of the social graph to provide data privacy without compromising data processing in a decentralized OSN, SPAR [15] uses social information to improve the scalability of centralized OSNs such as Facebook or Twitter. By co-locating the data of socially proximate users on the same physical machine, SPAR can reduce the time to compose a user's update feed and eliminate network traffic. SPAR may also be useful for decentralized systems such as Confidant for reducing the number of storage servers clients have to communicate with.

As discussed Section 3.2, recent work on Cimbiosys [17, 27] comes closest to Confidant's scheme for managing replicated access policies. The two systems have a great deal in common, but their are two main differences. First, Confidant eliminates write conflicts by serializing a user's updates through her name server. Second, in Cimbiosys, policies are bound to immutable object attributes (labels), while in Confidant, policies are bound to individual data items. By binding policies at a finer granularity is more appropriate for OSNs, where users often want to change the permissions of a single item. However, Confidant can also enable efficient bulk policy changes due to the limited scale of user's personal OSN data sets.

Lockr [22] is an identity-management tool for OSNs that allows users to codify their relationships through social attestations. Confidant borrows this idea from Lockr and uses it to manage groups of users.

Persona [1] is one of many systems [6, 7, 16, 20] that assumes that remote storage servers are untrusted. We have articulated the advantages of locating trusted storage servers throughout this paper. However, it is worth noting that Confidant's collusion protections are weaker than those provided by Persona. Persona uses an attribute-based encryption scheme to defend against attacks in which colluding users combine keys to falsely claim membership in an intersection of groups. In Confidant we trade robustness to collusion attacks for a faster, simpler protection scheme based on common symmetric-key algorithms.

7 Conclusion

We have presented Confidant, a decentralized OSN designed to support a scalable application framework. The key insight behind Confidant is that friends who already have access to a user's data may be trusted to serve it as well. Trace-based simulations and experiments with a Confidant prototype demonstrate the feasibility of our approach.

Acknowledgements. The work by the co-authors from Duke University was supported by the National Science Foundation under award CNS-0916649, as well as by AT&T Labs and Amazon.

References

1. Baden, R., et al.: Persona: an online social network with user-defined privacy. In: SIGCOMM 2009 (2009)
2. Bolosky, W., et al.: Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. SIGMETRICS (2000)
3. Chun, H., et al.: Comparison of online social relations in volume vs interaction: a case study of cyworld. In: IMC 2008 (2008)

4. Dean, J., et al.: Mapreduce: simplified data processing on large clusters. Commun. ACM (2008)
5. Douglas, T., et al.: Managing update conflicts in bayou, a weakly connected replicated storage system. In: SOSP (1995)
6. Shi, E., et al.: Multi-dimensional range query over encrypted data. In: IEEE Symposium on Security and Privacy (2007)
7. Fabbri, D., et al.: Privatepond: Outsourced management of web corpuses. In: WebDB (2009)
8. Facebook site info from alexa.com, `http://www.alexa.com/`
9. Facebook statistics, `http://www.facebook.com/press/`
10. Gilbert, E., et al.: Predicting tie strength with social media. In: CHI 2009 (2009)
11. Golding, R.A.: A weak-consistency architecture for distributed information services. Computing Systems (1992)
12. Lampson, B.W.: Hints for computer system design. IEEE Software (1983)
13. Liu, D., Shakimov, A., Cáceres, R., Varshavsky, A., Cox, L.P.: Confidant: Protecting OSN Data without Locking it Up. Technical Report TR-2010-04, Duke University, Department of Computer Science, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA (August 2010)
14. Noble, B., Satyanarayanan, M.: An empirical study of a highly available file system. In: SIGMETRICS (1994)
15. Puhol, J., et al.: The little engine(s) that could: Scaling online social networks. In: SIGCOMM 2010 (2010)
16. Puttaswamy, K., Zhao, B.: Preserving privacy in location-based mobile social applications. In: HotMobile (2010)
17. Ramasubramanian, V., et al.: Cimbiosys: a platform for content-based partial replication. In: NSDI 2009 (2009)
18. Sala, A., et al.: Measurement-calibrated graph models for social network experiments. In: WWW 2010 (2010)
19. Saroiu, S., et al.: Measuring and analyzing the characteristics of napster and gnutella hosts. Multimedia Syst. (2003)
20. Song, D., et al.: Practical techniques for searches on encrypted data. In: IEEE Symposium on Security and Privacy (2000)
21. The Economist. Primates on facebook (February 2009)
22. Tootoonchian, A., et al.: Lockr: better privacy for social networks. In: CoNEXT 2009 (2009)
23. Tran, D., et al.: Friendstore: cooperative online backup using trusted nodes. In: SocialNets 2008 (2008)
24. Viswanath, B., et al.: On the evolution of user interaction in facebook. In: WOSN (2009)
25. Wilson, C., et al.: User interactions in social networks and their implications. In: EuroSys 2009 (2009)
26. Winsborough, W., et al.: Towards practical automated trust negotiation. In: Policy 2002 (2002)
27. Wobber, T., et al.: Policy-based access control for weakly consistent replication. In: EuroSys 2010 (2010)