Appeared in ***USENIX Security 2012***

Social Networking with Frientegrity:
Privacy and Integrity with an Untrusted Provider

Ariel J. Feldman, Aaron Blankstein, Michael J. Freedman, and Edward W. Felten

Princeton University

Abstract

Today's social networking services require users to trust the service provider with the confidentiality and integrity of their data. But with their history of data leaks and privacy controversies, these services are not always deserving of this trust. Indeed, a malicious provider could not only violate users' privacy, it could *equivocate* and show different users divergent views of the system's state. Such misbehavior can lead to numerous harms including surreptitious censorship.

In light of these threats, this paper presents Frientegrity, a framework for social networking applications that can be realized with an *untrusted* service provider. In Frientegrity, a provider observes only encrypted data and cannot deviate from correct execution without being detected. Prior secure social networking systems have either been decentralized, sacrificing the availability and convenience of a centralized provider, or have focused almost entirely on users' privacy while ignoring the threat of equivocation. On the other hand, existing systems that are robust to equivocation do not scale to the needs social networking applications in which users may have hundreds of friends, and in which users are mainly interested the latest updates, not in the thousands that may have come before.

To address these challenges, we present a novel method for detecting provider equivocation in which clients collaborate to verify correctness. In addition, we introduce an access control mechanism that offers efficient revocation and scales logarithmically with the number of friends. We present a prototype implementation demonstrating that Frientegrity provides latency and throughput that meet the needs of a realistic workload.

1. Introduction

Popular social networking sites have hundreds of millions of active users [20]. They have enabled new forms of communication, organization, and information sharing; or, as Facebook's prospectus claims, they exist "to make the world more open and connected" [60]. But by now, it is widely understood that these benefits come at the cost of having to trust these centralized services with the privacy of one's social interactions. The history of these services is rife with unplanned data disclosures (*e.g.*, [22, 40]), and these services' centralization of information makes them attractive targets for attack by malicious insiders and outsiders. In addition, social networking sites face pressure from government agencies world-wide to release information on demand, often without search warrants [24]. Finally and perhaps worst of all, the behavior of service providers themselves is a source of users' privacy concerns. Providers have repeatedly changed their privacy policies and default privacy settings, and have made public information that their users thought was private [46, 47].

Less recognized, however, is the extent to which users trust social networking sites with the *integrity* of their data, and the harm that a malicious or compromised provider could do by violating it. Prior work on secure social networking has focused primarily on privacy and largely neglected integrity, or at most employed digital signatures on users' individual messages [5, 53, 54, 56]. But a malicious provider could be more insidious. For example, bloggers have claimed that Sina Weibo, a Chinese microblogging site, tried to disguise its censorship of a user's posts by hiding them from the user's followers but still showing them to the user [51]. This behavior is an example of server *equivocation* [34, 39], in which a malicious service presents different clients with divergent views of the system state. We argue that to truly protect users' data, a secure social networking service should defend against this sort of attack.

To address the security concerns surrounding social networking, numerous prior works (*e.g.*, [5, 17, 56]) have proposed decentralized designs in which the social networking service is provided not by a centralized provider, but by a collection of federated nodes. Each node could either be a service provider of a user's choice or the user's own machine or those of her friends. We believe that decentralization is the wrong, or at least an insufficient, approach, however, because it leaves the user with an unenviable dilemma: either sacrifice availability, reliability, and convenience by storing her data on her own machine, or entrust her data to one of several providers that she probably does not know or trust any more than she would a centralized provider.

In light of these problems, we present Frientegrity, a framework for building social networking services that protects the *privacy* and *integrity* of users' data from a

potentially malicious provider, while preserving the availability, reliability, and usability benefits of centralization. Frientegrity supports familiar social networking features such as "walls," "news feeds," comment threads, and photos, as well as common access control mechanisms such as "friends," "friends-of-friends," and "followers." But in Frientegrity, the provider's servers only see encrypted data, and clients can collaborate to detect server equivocation and other forms of misbehavior such as failing to properly enforce access control. In this way, Frientegrity bases its confidentiality and integrity guarantees on the security of users' cryptographic keys, rather than on the service provider's good intentions or the correctness of its complex server code. Frientegrity remains highly scalable while providing these properties by spreading system state across many shared-nothing servers [52].

To defend against server equivocation, Frientegrity enforces a property called *fork* consistency* [33]. A fork*-consistent system ensures that if the provider is honest, clients see a strongly-consistent (linearizable [27]) ordering of updates to an object (*e.g.*, a wall or comment thread). But if a malicious provider presents a pair of clients with divergent views of the object, then the provider must prevent the clients from ever seeing each other's subsequent updates lest they identify the provider as faulty.

Prior systems have employed variants of fork* consistency to implement network file systems [33, 34], key-value stores [7, 38, 50], and group collaboration systems [21] with untrusted servers. But these systems assumed that the number of users would be small or that clients would be connected to the servers most of the time. As a result, to enforce fork* consistency, they presumed that it would be reasonable for clients to perform work that is linear in either the number of users or the number of updates ever submitted to the system. But these assumptions do not hold in social networking applications in which users have hundreds of friends, clients are Web browsers or mobile devices that connect only intermittently, and users typically are interested only in the most recent updates, not in the thousands that may have come before.

To accommodate these unique scalability challenges, we present a novel method of enforcing fork* consistency in which clients *collaborate* to detect server equivocation. This mechanism allows each client to do only a small portion of the work required to verify correctness, yet is robust to collusion between a misbehaving provider and as many as f malicious users, where f is a predetermined security parameter per object.

Access control is another area where social networking presents new scalability problems. A user may have hundreds of friends and tens of thousands of friends-of-friends (FoFs) [19]. Yet, among prior social networking systems that employ encryption for access control (*e.g.*, [5, 9, 37]), many require work that is linear in the number of friends, if not FoFs, to revoke a friend's access (*i.e.*, to "un-friend"). Frientegrity, on the other hand, supports fast revocation of friends and FoFs, and also gives clients a means to efficiently verify that the provider has only allowed writes from authorized users. It does so through a novel combination of *persistent authenticated dictionaries* [12] and *key graphs* [59].

To evaluate the scalability of Frientegrity, we implemented a prototype that simulates a Facebook-like service. We demonstrate that Frientegrity is capable of scaling with reasonable performance by testing this prototype using workloads with tens of thousands of updates per object and access control lists containing hundreds of users.

Roadmap In §2, we introduce Frientegrity's goals and the threat model against which it operates. §3 presents an overview of Frientegrity's architecture using the task of fetching a "news feed" as an example. §4 delves into the details of Frientegrity's data structures and protocols for collaboratively enforcing fork* consistency on an object, establishing dependencies between objects, and enforcing access control. §5 discusses additional issues for untrusted social networks such as friend discovery and group administration. We describe our prototype implementation in §6 and then evaluate its performance and scalability in §7. We discuss related work in §8 and then conclude.

2. System Model

In Frientegrity, the service provider runs a set of servers that store *objects*, each of which corresponds to a familiar social networking construct such as a Facebook-like "wall", a comment thread, or a photo or album. Clients submit updates to these objects, called *operations*, on behalf of their users. Each operation is encrypted under a key known only to a set of authorized users, such as a particular user's friends, and not to the provider. Thus, the role of the provider's servers is limited to storing operations, assigning them a canonical order, and returning them to clients upon request, as well as ensuring that only authorized clients can write to each object. To confirm that servers are fulfilling this role faithfully, clients collaborate to verify their output. Whenever a client performs a read, it checks whether the response is consistent with the responses that other clients received.

2.1 Goals

Frientegrity should satisfy the following properties:

Broadly applicable: If Frientegrity is to be adopted, it must support the features of popular social networks such as Facebook-like walls or Twitter-like feeds. It must also support both the symmetric "friend" and "friend-of-friend" relationships of services like Facebook and the asymmetric "follower" relationships of services like Twitter.

Keeps data confidential: Because the provider is untrusted, clients must encrypt their operations before submitting them to the provider's servers. Frientegrity must ensure that all and only the clients of authorized users can obtain the necessary encryption keys.

Detects misbehavior: Even without access to objects' plaintexts, a malicious provider could still try to forge or alter clients' operations. It could also equivocate and show different clients inconsistent views of the objects. Moreover, malicious users could collude with the provider to deceive other users or could attempt to falsely accuse the provider of being malicious. Frientegrity must guarantee that as long as the number of malicious users with permission to modify an object is below a predetermined threshold, clients will be able to detect such misbehavior.

Efficient: Frientegrity should be sufficiently scalable to be used in practice. In particular, a client that is only interested in the most recent updates to an object should not have to download and check the object in its entirety just so that the it can perform the necessary verification. Furthermore, because social networking users routinely have hundreds of friends and tens of thousands of friends-of-friends [19], access control list changes must be performed in time that is better than linear in the number of users.

2.2 Detecting Server Equivocation

To prevent a malicious provider from forging or modifying clients' operations without detection, Frientegrity clients digitally sign all their operations with their users' private keys. But as we have discussed, signatures are not sufficient for correctness, as a misbehaving provider could still equivocate about the history of operations.

To mitigate this threat, Frientegrity employs *fork* consistency* [33].[1] In fork*-consistent systems, clients share information about their individual views of the history by embedding it in every operation they send. As a result, if clients to whom the provider has equivocated ever communicate, they will discover the provider's misbehavior. The provider can still *fork* the clients into disjoint groups and only tell each client about operations by others in its group, but then it can never again show operations from one group to the members of another without risking detection. Furthermore, if clients are occasionally able to exchange views of the history out-of-band, even a provider which forks the clients will not be able to cheat for long.

[1]Fork* consistency is a weaker variant of an earlier model called *fork consistency* [39]. They differ in that under fork consistency, a pair of clients only needs to exchange one message to detect server equivocation, whereas under fork* consistency, they may need to exchange two. Frientegrity enforces fork* consistency because it permits a one-round protocol to submit operations, rather than two. It also ensures that a crashed client cannot prevent the system from making progress.

Ideally, to mitigate the threat of provider equivocation, Frientegrity would treat all of the operations performed on all of the objects in the system as a single, unified history and enforce fork* consistency on that history. Such a design would require establishing a total order on all of the operations in the system regardless of the objects to which they belonged. In so doing, it would create unnecessary dependencies between unrelated objects, such as between the "walls" of two users on opposite sides of the social graph. It would then be harder to store objects on different servers without resorting to either expensive agreement protocols (*e.g.*, Paxos [31]) or using a single serialization point for all operations.

Instead, like many *scale-out* services, objects in Frientegrity are spread out across many servers; these objects may be indexed either through a directory service [1, 23] or through hashing [15, 30]. The provider handles each object independently and only orders operations with respect to the other operations in the same object. Clients, in turn, exchange their views of each object to which they have access separately. Thus, for efficiency, Frientegrity only enforces fork* consistency on a per-object basis.

There are situations, however, when it is necessary to make an exception to this rule and specify that an operation in one object happened after an operation in another. Frientegrity allows clients to detect provider equivocation about the order of such a pair of operations by supplying a mechanism for explicitly entangling the histories of multiple objects (see §3.4).

2.3 Threat Model

Provider: We assume that the provider may be actively malicious. It may not only attempt to violate the confidentiality of users' social interactions, but also may attempt to compromise their integrity through either equivocation or by directly tampering with objects, operations, or access control lists (ACLs).

Although Frientegrity makes provider misbehavior detectable, it does not prevent a malicious provider from denying service, either by blocking all of a client's updates or by erasing the encrypted data it stores. To mitigate this threat, clients could replicate their encrypted operations on servers run by alternate providers. Furthermore, if provider equivocation creates inconsistencies in the system's state, clients can resolve them using fork-recovery techniques, such as those employed by SPORC [21]. We argue, however, that because Frientegrity allows provider misbehavior to be detected quickly, providers will have an incentive to avoid misbehaving out of fear of legal repercussions or damage to their reputations.

The provider does not have access to the contents of objects or the contents of the individual operations that clients upload, because they are encrypted under keys that it does not know. In addition, because users' names are

also encrypted, the provider can only identify users by pseudonyms, such as the hash of the public keys they use within the system. Nevertheless, we do not seek to hide social relationships: we assume that the provider can learn the entire pseudonymous social graph, including who is friends with whom and who interacts with whom, by analyzing the interconnections between objects and by keeping track of which pseudonyms appear in which objects, (*e.g.*, by using social network deanonymization techniques [4, 43]).

Preventing the provider from learning the social graph is likely to be impossible in practice because even if users used a new pseudonym for every new operation, the provider would still be able to infer a great deal from the size and timing of their operations. After all, in most social networking applications, the first thing a user does when she signs in is check a "news feed" which is comprised of her friends' most recent updates. In order to construct the news feed, she must query each of her friend's feed objects in succession, and in so doing reveal to the provider which feed objects are related.

Users and Clients: We assume that users may also be malicious and may use the clients they control to attempt to read and modify objects to which they do not have access. In addition, malicious users may collude with the provider or with other users to exceed their privileges or to deceive honest users. They may also attempt to falsely accuse the provider of misbehavior. Finally, we assume that some clients may be controlled by Sybil users, created by the provider to subvert the clients' defenses against server equivocation.

Frientegrity's security is based on the assumption, however, that among the users which have access to a given object, no more than some constant f will be malicious (Byzantine faulty). We believe that this assumption is reasonable because a user can only access an object if she has been explicitly invited by another user with administrator privileges for the object (*e.g.*, Alice can only access Bob's wall if he explicitly adds her as a friend). As we describe in §4.1, this assumption allows clients to collaborate to detect provider misbehavior. If a client sees that at least $f + 1$ other users have vouched for the provider's output, the client can assume that it is correct.

Client code: We assume the presence of a code authentication infrastructure that can verify that the application code run by clients is genuine. This mechanism might rely on code signing or on HTTPS connections to a trusted server (different from the untrusted service provider used as part of Frientegrity's protocols).

3. System Overview

As discussed above, to ensure that the provider is behaving correctly, Frientegrity requires clients to verify the output that they receive from the provider's servers. As a result, whenever clients retrieve the latest updates to an object, the provider's response must include enough information to make such verification possible. In addition, the provider must furnish the key material that allows authorized clients with the appropriate private keys to decrypt the latest operations. Thus, when designing Frientegrity's protocols and data structures, our central aim was to ensure that clients could perform the necessary verification and obtain the required keys *efficiently*.

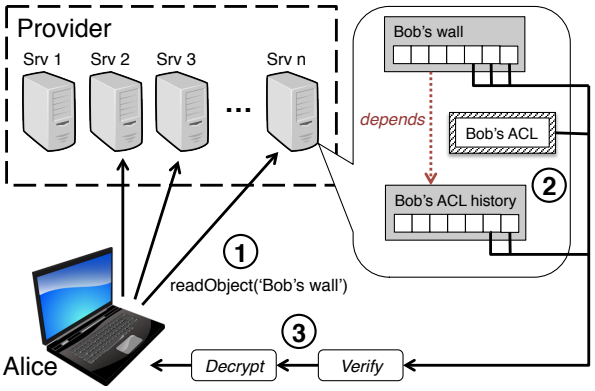


Figure 1: A client fetches a news feed in Frientegrity by reading the latest posts from her friends' walls, as well as information to verify, authenticate, and decrypt the posts.

To explain these mechanisms, we use the example of a user Alice who wants to fetch her "news feed" and describes the steps that her client takes on her behalf. For simplicity, in this and subsequent examples throughout the paper, we often speak of users, such as Alice, when we really mean to refer to the clients acting on their behalf.

3.1 Example: Fetching a News Feed

Alice's news feed consists of the most recent updates to the sources to which she is subscribed. In Facebook, for example, this typically corresponds to the most recent posts to her friends' "walls", whereas in Twitter, it is made up of the most recent tweets from the users she follows. At a high level, Frientegrity performs the following steps when Alice's fetches her news feed, as shown in Figure 1.

1. For each of Alice's friends, Alice's sends a readObject RPC to the server containing the friend's wall object.
2. In response to a readObject RPC for a friend Bob, a well-behaved server returns the most recent operations in Bob's wall, as well as sufficient information and key material for Alice to verify and decrypt them.
3. Upon receiving the operations from Bob's wall, Alice performs a series of verification steps aimed at detecting server misbehavior. Then, using her private key, she decrypts the key material and uses it to decrypt the operations. Finally, when she has verified and decrypted the recent wall posts from all her

friends, she combines them and optionally filters and prioritizes them according to a client-side policy.

For Alice to verify the response to each readObject, she must be able to check the following properties efficiently:

1. ***The provider has not equivocated about the wall's contents:*** The provider must return enough of the wall object to allow Alice to guarantee that history of the operations performed on the wall is fork* consistent.
2. ***Every operation was created by an authorized user:*** The provider must prove that each operation from the wall that it returns was created by a user who was authorized to do so at the time that the operation was submitted.
3. ***The provider has not equivocated about the set of authorized users:*** Alice must be able to verify that the provider did not add, drop, or reorder users' modifications to the access control list that applies to the wall object.
4. ***The ACL is not outdated:*** Alice must be able to ensure that the provider did not roll back the ACL to an earlier version in order to trick the client into accepting updates from a revoked user.

The remainder of this section summarizes the mechanisms with which Frientegrity enforces these properties.

3.2 Enforcing Fork* Consistency

Clients defend against provider equivocation about the contents of Bob's wall or any other object by comparing their views of the object's history, thereby enforcing fork* consistency. Many prior systems, such as BFT2F [33] and SPORC [21], enforced fork* consistency by having each client maintain a linear hash chain over the operations that it has seen. Every new operation that it submits to the server includes the most recent hash. On receiving an operation created by another client, a client in such systems checks whether the history hash included in the operation matches the client's own hash chain computation. If it does not, the client knows that the server has equivocated.

The problem with this approach is that it requires each client to perform work that is linear in the size of the entire history of operations. This requirement is ill suited to social networks because an object such as Bob's wall might contain thousands of operations dating back years. If Alice is only interested in Bob's most recent updates, as is typically the case, she should not have to download and check the entire history just to be able to detect server equivocation. This is especially true considering that when fetching a news feed, Alice must read all of her friends' walls, and not just Bob's.

To address these problems, Frientegrity clients verify an object's history collaboratively, so that no single client needs to examine it in its entirety. Frientegrity's collaborative verification scheme allows each client to do only a small portion of the work, yet is robust to collusion between a misbehaving provider and as many as f malicious users. When f is small relative to the number of users who have written to an object, each client will most likely only have to do work that is logarithmic, rather than linear, in the size of the history (as our evaluation demonstrates in §7.5). We present Frientegrity's collaborative verification algorithm in §4.1.

3.3 Making Access Control Verifiable

A user Bob's profile is comprised of multiple objects in addition to his wall, such as photos and comment threads. To allow Bob to efficiently specify the users allowed to access all of these objects (*i.e.*, his friends), Frientegrity stores Bob's friend list all in one place as a separate *ACL*. ACLs store users' pseudonyms in the clear, and every operation is labeled with the pseudonym of its creator. As a result, a well-behaved provider can reject operations that were submitted by unauthorized users. But because the provider is untrusted, when Alice reads Bob's wall, the provider must *prove* that it enforced access control correctly on every operation it returns. Thus, Frientegrity's ACL data structure must allow the server to construct efficiently-checkable proofs that the creator of each operation was indeed authorized by Bob.

Frientegrity also uses the ACL to store the key material with which authorized users can decrypt the operations on Bob's wall and encrypt new ones. Consequently, ACLs must be designed to allow clients with the appropriate private keys to efficiently retrieve the necessary key material. Moreover, because social network ACLs may be large, ACL modifications and any associated rekeying must be efficient.

To support both efficiently-checkable membership proofs and efficient rekeying, Frientegrity ACLs are implemented as a novel combination of *persistent authenticated dictionaries* [12] and *key graphs* [59]. Whereas most prior social networking systems that employ encryption required work linear in the number of friends to revoke a user's access, all of Frientegrity's ACL operations run in logarithmic time.

Even if it convinces Alice that every operation came from someone who was authorized by Bob at some point, the provider must still prove that it did not equivocate about the history of changes Bob made to his ACL. To address this problem, Frientegrity maintains an *ACL history* object, in which each operation corresponds to a change to the ACL and which Alice must check for fork* consistency, just like with Bob's wall. Frientegrity's ACL data structure and how it interacts with ACL histories are further explained in §4.3.

3.4 Preventing ACL Rollbacks

Even without equivocating about the contents of either Bob's wall or his ACL, a malicious provider could still give Alice an outdated ACL in order to trick her into accepting operations from a revoked user. To mitigate this threat, operations in Bob's wall are annotated with *dependencies* on Bob's ACL history (the red dotted arrow in Figure 1). A dependency indicates that a particular operation in one object *happened after* a particular operation in another object. Thus, by including a dependency in an operation that it posts to Bob's wall, a client forces the provider to show anyone who later reads the operation an ACL that is at least as new as the one that the client observed when it created the operation. In §4.2, we explain the implementation of dependencies and describe additional situations where they can be used.

4. System Design

Clients interact with Frientegrity primarily by reading and writing objects and ACLs via the following four RPCs: [2]

- readObject(objectID, k, [otherOps]). Returns the k most recent operations in object objectID, and optionally, a set of additional earlier operations from the object (otherOps). But as we explain in the previous section, the provider must also return enough operations from the object to allow the client to verify that the provider has not equivocated and proofs from the ACL that show that every operation came from an authorized user. In addition, it must return key material from the ACL that allows the client to decrypt the object.
- writeObject(objectID, op). Submits the new operation op to object objectID. Every new operation is signed by the user that created it. To allow clients to enforce fork* consistency, it also includes a compact representation of the submitting client's view of object's state. (This implies that the client must have read the object at least once before submitting an update.)
- readACL(aclID, [userToAdd] [userToRemove]). Returns ACL aclID and its corresponding ACL history object. As an optimization, the client can optionally specify in advance that it intends to add or remove particular users from the ACL so that the provider only has to return the portion of the ACL that the client needs to change.
- writeACL(aclID, aclUpdate). Submits an update to ACL aclID. Only administrator users (*e.g.*, the owner of a Facebook-like profile) can modify the ACL. The objects to which the ACL applies are encrypted under a key that is shared only among currently authorized users. Thus, to add a user, the client must update the ACL so that it includes the encryption of this shared key under the new user's public key. To remove a user, the ACL must be updated with a new shared key encrypted such that all remaining users can retrieve it. (See §4.3.3.)

[2] For brevity, we omit RPCs for creating new objects and ACLs and for adding new users to the system.

The remainder of this section describes how Frientegrity makes these RPCs possible. It discusses the algorithms and data structures underlying object verification (§4.1), dependencies between objects (§4.2), and verifiable access control §4.3).

4.1 Making Objects Verifiable

4.1.1 Object Representation

Frientegrity's object representation must allow clients to compare their views of the object's history without requiring any of them to have the entire history. Representing an object as a simple list of operations would be insufficient because it is impossible to compute the hash of a list without having all of the elements going back to the first one. As a result, objects in Frientegrity are represented as *history trees*.

A history tree, first introduced by Crosby *et al.* [11] for tamper-evident logging, is essentially a versioned Merkle tree [41]. Like an ordinary Merkle tree, data (in this case, operations) are stored in the leaves, each internal node stores the hash of the subtree below it, and the hash of the root covers the tree's entire contents. But unlike a static Merkle tree, a history tree allows new leaves (operations) to be added to the right side of the tree. When that occurs, a new version of the tree is created and the hashes of the internal nodes are recomputed accordingly.

This design has two features that are crucial for Frientegrity. First, as with a Merkle tree, subtrees containing unneeded operations can be omitted and replaced by a stub containing the subtree's hash. This property allows a Frientegrity client which has only downloaded a subset of an object's operations to still be able to compute the current history hash. Second, if one has a version j history tree, it is possible to compute what the root hash would have been as of version $i < j$ by pretending that operations $i + 1$ through j do not exist, and by then recomputing the hashes of the internal nodes.

Frientegrity uses history trees as follows. Upon receiving a new operation via a writeObject RPC, the server hosting the object adds it to the object's history tree, updates the root hash, and then digitally signs the hash. This server-signed hash is called a *server commitment* and is signed to prevent a malicious client from later falsely accusing the server of cheating.

When Alice reads an object of version i by calling readObject, the server responds with a pruned copy of the object's history tree containing only a subset of the opera-

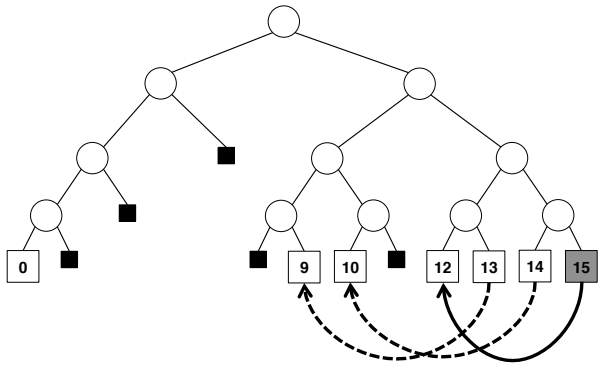


Figure 2: A pruned object history that a provider might send to a client. Numbered leaves represent operations and filled boxes represent stubs of omitted subtrees. The solid arrow represents the last operation's prevCommitments. Dashed arrows represent other prevCommitments.

tions, along with C_i, the server commitment to version i of the object. If Alice then creates a new operation, she shares her view of the history with others by embedding C_i in the operation's prevCommitment field. If Bob later reads the object, which by then has version $j \geq i$, he can compare the object he receives with what Alice saw by first computing what the root hash would have been at version i from his perspective and then comparing it to the prevCommitment of Alice's operation. If his computed value C'_i does not equal C_i, then he knows the server has equivocated.

4.1.2 Verifying an Object Collaboratively

But how many operations' prevCommitments does a client need to check in order to be confident that the provider has not misbehaved? Clearly, if the client checks every operation all the way back to the object's creation, then using a history tree provides no advantage over using a hash chain. Consequently, in Frientegrity, each client only verifies a suffix of the history and trusts others to check the rest. If we assume that there are at most f malicious users with write access to an object, then as long as at least $f + 1$ users have vouched for a prefix of the history, subsequent clients do not need to examine it.

To achieve this goal, every client executes the following algorithm to verify an object that it has fetched. In response to a readObject RPC, the provider returns a pruned object history tree that includes all of the operations the client requested along with any additional ones that the client will need to check in order to verify the object. Because the provider knows the client's verification algorithm, it can determine a priori which operations the client will need. For simplicity, the algorithm below assumes that only one user needs to vouch for a prefix of the history in order for it to be considered trustworthy (*i.e.*, $f = 0$). We relax this assumption in the next section.

1. Suppose that Alice fetches an object, and the provider replies with the pruned object shown in Figure 2. Because the object has version 15, the provider also sends Alice its commitment C_{15}. On receiving the object, she checks the server's signature on C_{15}, recomputes the hashes of the internal nodes, and then verifies that her computed root hash C'_{15} matches C_{15}. Every operation she receives is signed by the user that created it, and so she verifies these signatures as well.
2. Alice checks the prevCommitment of the last operation (op_{15}), which in this case is C_{12}.[3] To do so, Alice computes what the root hash would have been if op_{12} were the last operation and compares her computed value to C_{12}. (She must have op_{12} to do this.)
3. Alice checks the prevCommitment of every operation between op_{12} and op_{15} in the same way.
4. Frientegrity identifies every object by its first operation.[4] Thus, to make sure that the provider did not give her the wrong object, Alice checks that op_0 has the value she expects.

4.1.3 Correctness of Object Verification

The algorithm above aims to ensure that at least one honest user has checked the contents and prevCommitment of every operation in the history. To see how it achieves this goal, suppose that op_{15} in the example was created by the honest user Bob. Then, C_{12} must have been the most recent server commitment that Bob saw at the time he submitted the operation. More importantly, however, because Bob is honest, Alice can assume that he would have never submitted the operation unless he had already verified the entire history up to op_{12}. As a result, when Alice verifies the object, she only needs to check the contents and prevCommitments of the operations after op_{12}. But how was Bob convinced that the history is correct up to op_{12}? He was persuaded the same way Alice was. If the author of op_{12} was honest, and op_{12}'s prevCommitment was C_i, then Bob only needed to examine the operations from op_{i+1} to op_{12}. Thus, by induction, as long as writers are honest, every operation is checked even though no single user examines the whole history.

Of course in the preceding argument, if any user colludes with a malicious provider, then the chain of verifications going back to the beginning of the history is broken. To mitigate this threat, Frientegrity clients can tolerate up to f malicious users by looking back in the history until they find a point for which at least $f + 1$ different users

[3] An operation's prevCommitment need not refer to the immediately preceding version. This could occur, for example, if the operation had been submitted concurrently with other operations.

[4] Specifically, an objectID is equal to the hash of its first operation, which contains a client-supplied random value, along with the provider's name and a provider-supplied random value.

have vouched. Thus, in the example, if $f = 2$ and op_{13}, op_{14}, and op_{15} were each created by a different user, then Alice can rely on assurances from others about the history up to op_9, but must check the following operations herself.

Frientegrity allows the application to use a different value of f for each type of object, and the appropriate f value depends on the context. For example, for an object representing a Twitter-like feed with a single trusted writer, setting $f = 0$ might be reasonable. By contrast, an object representing the wall of a large group with many writers might warrant a larger f value.

The choice of f impacts performance: as f increases, so does the number of operations that every client must verify. But when f is low relative to the number of writers, verifying an object requires logarithmic work in the history size due to the structure of history trees. We evaluate this security vs. performance trade-off empirically in §7.5.

4.2 Dependencies Between Objects

Recall that, for scalability, the provider only orders the operations submitted to an object with respect to other operations in the same object. As a result, Frientegrity only enforces fork* consistency on the history of operations within each object, but does not ordinarily provide any guarantees about the order of operations across different objects. When the order of operations spanning multiple objects is relevant, however, the objects' histories can be entangled through *dependencies*. A dependency is an assertion of the form ⟨ srcObj, srcVers, dstObj, dstVers, dstCommitment ⟩, indicating that the operation with version srcVers in srcObj happened after operation dstVers in dstObj, and that the server commitment to dstVers of dstObj was dstCommitment.

Dependencies are established by authorized clients in accordance with a policy specified by the application. When a client submits an operation to srcObj, it can create a dependency on dstObj by annotating the operation with the triple ⟨ dstObj, dstVers, dstCommitment ⟩. If another client subsequently reads the operation, the dependency serves as evidence that dstObj must have at least been at version dstVers at the time the operation was created, and the provider will be unable to trick the client into accepting an older version of dstObj.

As described in §3.4, Frientegrity uses dependencies to prevent a malicious provider from tricking clients into accepting outdated ACLs. Whenever a client submits a new operation to an object, it includes a dependency on the most recent version of the applicable ACL history that it has seen.[5] Dependencies have other uses, however. For example, in a Twitter-like social network, every retweet could be annotated with a dependency on the original tweet to which it refers. In that case, a provider that wished to suppress the original tweet would not only have to suppress all subsequent tweets from the original user (because Frientegrity enforces fork* consistency on the user's feed), it would also have to suppress all subsequent tweets from all the users who retweeted it.

Frientegrity uses *Merkle aggregation* [11] to implement dependencies efficiently. This feature of history trees allows the attributes of the leaf nodes to be aggregated up to the root, where they can be queried efficiently. In Frientegrity, the root of every object's history tree is annotated with a list of the other objects that the object depends on, along with those objects' most recent versions and server commitments. To prevent tampering, each node's annotations are included in its hash, so that incorrectly aggregated values will result in an incorrect root hash.

4.3 Verifiable Access Control

4.3.1 Supporting Membership Proofs

When handling a readObject RPC, Frientegrity ACLs must enable the provider to construct proofs that demonstrate to a client that every returned operation was created by an authorized user. But to truly demonstrate such authorization, such a proof must not only show that a user was present in the ACL at *some* point in time, it must show that the user was in the ACL at the time the operation was created (*i.e.*, in the version of the ACL on which the operation depends). As a result, an ACL must support queries not only on the current version of its state, but on previous versions as well. The abstract data type that supports both membership proofs and queries on previous versions is known as a *persistent authenticated dictionary (PAD)*. Thus, in Frientegrity, ACLs are PADs.

To realize the PAD abstract data type, an ACL is implemented as a binary search tree in which every node stores both an entry for a user and the hash of the subtree below it.[6] To prove that an entry u exists, it suffices for the provider to return a pruned tree containing the search path from the root of the tree to u, in which unneeded subtrees in the path are replaced by stubs containing the subtrees' hashes. If the root hash of the search path matches the previously-known root hash of the full tree, a client can be convinced that u is in the ACL.

To support queries on previous versions of their states, ACLs are copy-on-write. When an administrator updates the ACL before calling writeACL, it does not modify any

[5] The annotation can be omitted if the prior operation in the object points to the same ACL history version.

[6] Our ACL construction expands on a PAD design from Crosby *et al.* [12] that is based on a *treap* [2]. A treap is a randomized search tree that is a cross between a tree and a heap.In addition to a key-value pair, every node has a priority, and the treap orders the nodes both according to their keys and according to the heap property. If nodes' priorities are chosen pseudorandomly, the tree will be balanced in expectation.

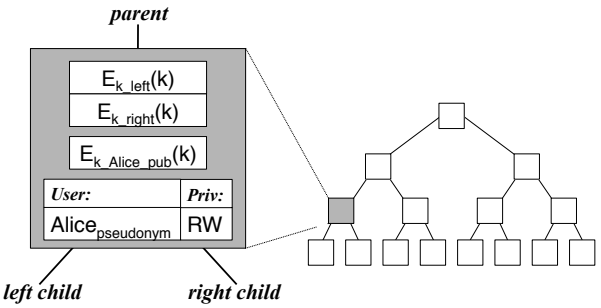


Figure 3: ACLs are organized as trees for logarithmic access time. Figure illustrates Alice's entry in Bob's ACL.

nodes directly. Instead, the administrator copies each node that needs to be changed, applies the update to the copy, and then copies all of its parents up to the root. As a result, there is a distinct root for every version of the ACL, and querying a previous version entails beginning a search at the appropriate root.

4.3.2 Preventing Equivocation about ACLs

To authenticate the ACL, it is not enough for an administrator to simply sign the root hash of every version, because a malicious provider could still equivocate about the history of ACL updates. To mitigate this threat, Frientegrity maintains a separate *ACL history* object that stores a log of updates to the ACL. An ACL history resembles an ordinary object, and clients check it for fork* consistency in the same way, but the operations that it contains are special ModifyUserOps. Each version of the ACL has a corresponding ModifyUserOp that stores the root hash as of that version and is signed by an administrator.

In summary, proving that the posts on a user Bob's wall were created by authorized users requires three steps. First, for each post, the provider must prove that the post's creator was in Bob's ACL by demonstrating a search path in the appropriate version of the ACL. Second, for each applicable version of Bob's ACL, the provider must provide a corresponding ModifyUserOp in Bob's ACL history that was signed by Bob. Finally, the provider must supply enough of the ACL history to allow clients to check it for fork* consistency, as described in §4.1.2.

4.3.3 Efficient Key Management and Revocation

Like many prior systems designed for untrusted servers (*e.g.*, [5, 21, 25, 37]), Frientegrity protects the confidentiality of users' data by encrypting it under a key that is shared only among currently authorized users. When any user's access is revoked, this shared key must be changed. Unfortunately, in most of these prior systems, changing the key entails picking a new key and encrypting it under the public key of the remaining users, thereby making revocation expensive.

To make revocation more efficient, Frientegrity organizes keys into *key graphs* [59]. But rather than maintaining a separate data structure for keys, Frientegrity stores keys in same ACL tree that is used for membership proofs. As shown in Figure 3, each node in Bob's ACL not only contains the pseudonym and privileges of an authorized user, such as Alice, it is also assigned a random AES key k. k is, in turn, encrypted under the keys of its left and right children, k_{left} and k_{right}, and under Alice's public key k_{Alice_pub}.[7] This structure allows any user in Bob's ACL to follow a chain of decryptions up to the root of the tree and obtain the root key k_{Bob_root}. As a result, k_{Bob_root} is shared among all of Bob's friends and can be used to encrypt operations that only they can access. Because the ACL tree is balanced in expectation, the expected number of decryptions required to obtain k_{Bob_root} is logarithmic in the number of authorized users. More significantly, this structure makes revoking a user's access take logarithmic time as well. When a node is removed, only the keys along the path from the node to the root need to be changed and reencrypted.

4.3.4 Supporting Friends-of-Friends

Many social networking services, including Facebook, allow users to share content with an audience that includes not only their friends, but also their "friends-of-friends" (FoFs). Frientegrity could be extended naively to support sharing with FoFs by having Bob maintain a separate key tree, where each node corresponded to a FoF instead of a friend. This approach is undesirable, however, as the size of the resulting tree would be quadratic in the number of authorized users. Instead, Frientegrity stores a second FoF key k' in each node of Bob's ACL. Similar to the friend key k, k' is encrypted under the FoF keys of the node's left and right children, k'_{left} and k'_{right}. But instead of being encrypted under k_{Alice_pub}, k' is encrypted under k_{Alice_root}, the root key of Alice's ACL. Thus, any of Alice's friends can decrypt k' and ultimately obtain k'_{Bob_root}, which can be used to encrypt content for any of Bob's FoFs.

The FoF design above assumes, however, that friend relationships are symmetric: Bob must be in Alice's ACL in order to obtain k_{Alice_root}. To support asymmetric "follower-of-follower" relationships, such as Google+ "Extended Circles," Frientegrity could be extended so that a user Alice maintains a separate public-private key pair $\langle k_{Alice\_FoF\_pub}, k_{Alice\_FoF\_priv} \rangle$. Alice could then give $k_{Alice\_FoF\_priv}$ to her followers by encrypting it under k_{Alice_root}, and she could give $k_{Alice\_FoF\_pub}$ to Bob. Finally, Bob could encrypt k' under $k_{Alice\_FoF\_pub}$.

[7] To lower the cost of changing k, k is actually encrypted under an AES key k_{user} which is, in turn, encrypted under k_{Alice_pub}.

5. Extensions

5.1 Discovering Friends

Frientegrity identifies users by pseudonyms, such as the hashes of their public keys. But to enable users to discover new friends, the system must allow them to learn other users' real names under certain circumstances. In Frientegrity, we envision that the primary way a user would discover new friends is by searching through the ACLs of her existing friends for FoFs that she might want to "friend" directly. To make this possible, users could encrypt their real names under the keys that they use to share content with their FoFs. A user Alice's client could then periodically fetch and decrypt the real names of her FoFs and recommend them to Alice as possible new friends. Alice's client could rank each FoF according to the number of mutual friends that Alice and the FoF share by counting the number of times that the FoF appears in an ACL of one of Alice's friends.

Frientegrity's design prevents the provider from offering site-wide search that would allow any user to locate any other users by their real names. After all, if any user could search for any other user by real name, then so could Sybils acting on behalf of a malicious provider. We believe that this limitation is unavoidable, however, because there is an inherent trade-off between users' privacy and the effectiveness of site-wide search even in existing social networking systems.[8] Thus, a pair of users who do not already share a mutual friend must discover each other, by exchanging their public keys out-of-band.

5.2 Multiple Group Administrators

As we describe in §4.3, when a user Alice reads another user Bob's wall, she verifies every wall post by consulting Bob's ACL. She, in turn, verifies Bob's ACL using Bob's ACL history, and then verifies each relevant ModifyUserOp by checking for Bob's signature. To support features like Facebook Groups or Pages, however, Frientegrity must be extended to enable multiple users to modify a single ACL and to allow these administrators be added and removed dynamically. But if the set of administrators can change, then, as with ordinary objects, a user verifying the ACL history must have a way to determine that every ModifyUserOp came from a user who was a valid administrator at the time the operation was created. One might think the solution to this problem is to have another ACL and ACL history just to keep track of which users are administrators at any given time. But this proposal merely shifts the problem to the question of who is authorized to write to *these* data structures.

Instead, we propose the following design. Changes to the set of administrators would be represented as special ModifyAdminOp. Each ModifyAdminOp would be included in the ACL history alongside the ModifyUserOp, but would also have a pointer to the previous ModifyAdminOp. In this way, the ModifyAdminOp would be linked together to form a separate *admin history*, and clients would enforce fork* consistency on this history using a linear hash chain in the manner of BFT2F [33] and SPORC [21]. When a client verifies the ACL history, it would download and check the entire *admin history* thereby allowing it to determine whether a particular user was an administrator when it modified the ACL history. Although downloading an entire history is something that we have otherwise avoided in Frientegrity, the cost of doing so here likely is low: Even when the set of regular users changes frequently, the set of administrators typically does not.

5.3 Dealing with Conflicts

When multiple clients submit operations concurrently, conflicts can occur. Because servers do not have access to the operations' plaintexts, Frientegrity delegates conflict resolution to the clients, which can employ a number of strategies, such as last-writer-wins, operational transformation [21], or custom merge procedures [55]. In practice, however, many kinds of updates in social networking systems, such as individual wall posts, are *append* operations that are inherently commutative, and thus require no special conflict resolution.

5.4 Public Feeds with Many Followers

Well-known individuals and organizations often use their feeds on online social networks to disseminate information to the general public. These feeds are not confidential, but they would still benefit from a social networking system that protected their integrity. Such feeds pose scalability challenges, however, because they can have as many as tens of millions of followers.

Fortunately, Frientegrity can be readily adapted to support these feeds efficiently. Because the object corresponding to such a feed does not need to be encrypted, its ACL does not need to store encryption keys. The ACL is only needed to verify that every operation in the object came from an authorized writer. As a result, the size of the object's ACL need only be proportional to the number of users with *write* access to the object, which is often only a single user, rather than to the total number of followers.

Popular feeds would also not prevent applications from using dependencies to represent retweets in the manner described in §4.2. Suppose that Alice retweets a post from the feed of a famous individual, such as Justin Bieber.

[8] For example, in 2009, Facebook chose to weaken users' privacy by forcing them to make certain information public, such as their genders, photos, and current cities. It adopted this policy, which it later reversed, so that it would be easier for someone searching for a particular user to distinguish between multiple users with the same name [46].

Then, in such a design, the application would establish a dependency from Alice's feed to Justin Bieber's. But because dependencies only modify the source object (in this case Alice's feed), they would not impose any additional performance penalty on reads of Justin Bieber's feed. Thus, even if Justin Bieber's posts are frequently retweeted, Frientegrity could still serve his feed efficiently.

6. Implementation

To evaluate Frientegrity's design, we implemented a prototype that simulates a simplified Facebook-like service. It consists of a server that hosts a set of user profiles and clients that fetch, verify, and update them. Each user profile is comprised of an object representing the user's "wall," as well as an ACL and ACL history object representing the user's list of friends. The wall object is made up of operations, each of which contains an arbitrary byte string, that have been submitted by the user or any of her friends. The client acts on behalf of a user and can perform RPCs on the server to read from and write to the walls of the user or user's friends, as well as to update the user's ACL. The client can simulate the work required to build a Facebook-like "news feed" by fetching and verifying the most recent updates to the walls of each of the user's friends in parallel.

Our prototype is implemented in approximately 4700 lines of Java code (per SLOCCount [58]) and uses the protobuf-socket-rpc [16] library for network communication. To support the history trees contained in the wall and ACL history objects, we use the reference implementation provided by Crosby *et al.* [13].

Because Frientegrity requires every operation to be signed by its author and every server commitment to be signed by the provider, high signature throughput is a priority. To that end, our prototype uses the Network Security Services for Java (JSS) library from Mozilla [42] to perform 2048-bit RSA signatures because, unlike Java's default RSA implementation, it is written in native code and offers significantly better performance. In addition, rather than signing and verifying each operation or server commitment individually, our prototype signs and verifies them in batches using *spliced signatures* [10, 13]. In so doing, we improve throughput by reducing the total number of cryptographic operations at the cost of a small potential increase in the latency of processing a single message.

7. Experimental Evaluation

Social networking applications place a high load on servers, and they require reasonably low latency in the face of objects containing tens of thousands of updates and friend lists reaching into the hundreds and thousands. This section examines how our Frientegrity prototype performs and scales under these conditions.

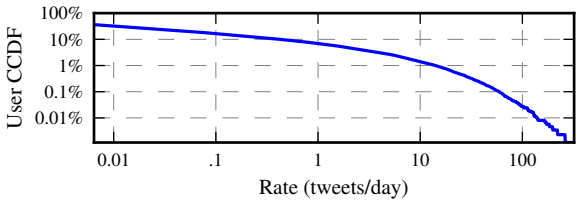


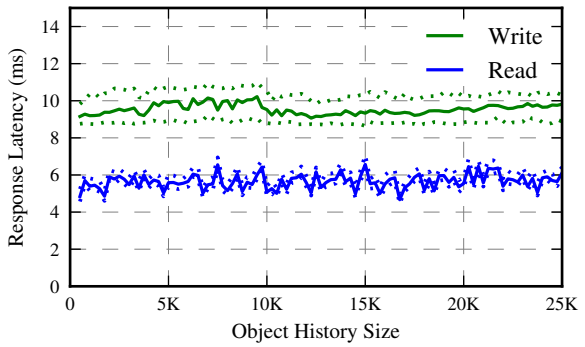
Figure 4: Distribution of post rates for Twitter users. 1% of users post at least 14 times a day, while 0.1% post at least 56 times a day.

All tests were performed on machines with dual 4-core Xeon E5620 processors clocked at 2.40 GHz, with 11 GB of RAM and gigabit network interfaces. Our evaluation ran with Oracle Java 1.6.0.24 and used Mozilla's JSS cryptography library to perform SHA256 hashes and RSA 2048 signatures. All tests, unless otherwise stated, ran with a single client machine issuing requests to a separate server machine on the same local area network, and all data is stored in memory. A more realistic deployment over the wide-area would include higher network latencies (typically an additional tens to low hundreds of milliseconds), as well as backend storage access times (typically in low milliseconds in datacenters). These latencies are common to any Web service, however, and so we omit any such synthetic overhead in our experiments.

7.1 Single-object Read and Write Latency

To understand how large object histories may get in practice, we collected actual social network usage data from Twitter by randomly selecting over 75,000 users with public profiles. Figure 4 shows the distribution of post rates for Twitter users. While the majority of users do not tweet at all, the most active users post over 200 tweets per day, leading to tens of thousands of posts per year.

To characterize the effect of history size on read and write latency, we measured performance of these operations as the history size varies. For each read, the client fetched an object containing the five most recent operations along with any other required to verify the object. As shown in Figure 5, write latency was approximately 10 ms (as it includes both a server and client signature in addition to hashing), while read latency was approximately 6 ms (as it includes a single signature verification and hashing). The Figure's table breaks down median request latency to its contributing components. As expected, a majority of the time was spent on public-key operations; a faster signature verification implementation or algorithm would correspondingly increase performance. While the latency here appears constant, independent of the history size, the number of hash verifications actually grows logarithmically with the history. This observed behavior arises because, at least up to histories of 25,000



Read	Server Data Fetches	0.45 ms	7.5%
	Network and Data Serialization	1.06 ms	17.5%
	Client Signature Verification	3.55 ms	58.8%
	Other (incl. Client Decrypt, Hashing)	0.98 ms	16.3%
	Total Latency	6.04 ms	
Write	Client Encryption	0.07 ms	0.7%
	Client Signature	4.45 ms	41.7%
	Network and Data Serialization	0.64 ms	6.0%
	Server Signature	4.31 ms	40.4%
	Other (incl. Hashing)	1.21 ms	11.3%
	Total Latency	10.67 ms	

Figure 5: Read and write latency for Frientegrity as the object history size increases from 0 to 25000. Each data point represents the median of 1000 requests. The dots above and below the lines indicate the 90th and 10th percentiles for each trial. The table breaks down the cost of a typical median read and write request.

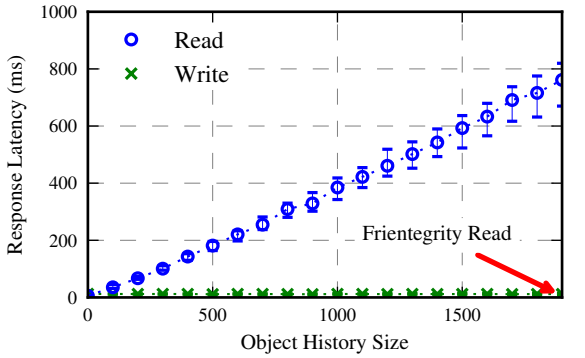


Figure 6: Latency for requests in a naive implementation using hash chains. The red arrow indicates the response time for Frientegrity read requests at an object size of 2000. Each data point is the median of 100 requests. The error bars indicate the 90th and 10th percentiles.

operations, the constant-time overhead of a public-key signature or verification continues to dominate the cost.

Next, we performed these same microbenchmarks on an implementation that verifies object history using a hash chain, rather than Frientegrity's history trees. In this experiment, each client was stateless, and so it had to perform a complete verification when reading an object. This verification time grows linearly with the object history size, as shown in Figure 6. Given this linear growth in latency, verifying an object with history size of 25,000 operations would take approximately 10 s in the implementation based on a hash chain compared to Frientegrity's 6 ms.

Object	Signatures	7210 B
	History Tree Hashes	640 B
	Dependency Annotations	224 B
	Other Metadata	1014 B
ACL	ACL PAD	453 B
	Signatures in ACL History	1531 B
	Hashes in ACL History Tree	32 B
	Other Metadata	226 B
Total Overhead		11300 B

Table 1: Sources of network overhead of a typical read of an object's five most recent updates.

The performance of hash chains could be improved by having clients cache the results of previous verifications so they would only need to verify subsequent operations. Even if clients were stateful, however, Figure 4 shows that fetching the latest updates of the most prolific users would still require hundreds of verifications per day. Worse still, following new users or switching between client devices could require tens of thousands of verifications.

7.2 Network Overhead

When network bandwidth is limited, the size of the messages that Frientegrity sends over the network can impact latency and throughput. To understand this effect, we measure the overhead that Frientegrity's verification and access control mechanisms add to an object that is fetched. Table 1 provides a breakdown of the sources of overhead in a read of the five most recent operations in an object. The object is comprised of 100 operations all created by a single writer. We assume that the ACL that applies to the object only contains a single user and his associated encrypted key and that the ACL history object contains only two operations (an initial operation and the operation that added the single user).

As shown in Table 1, the total overhead added by Frientegrity is 11,300 B, which would add approximately 90 ms of download time on a 1 Mbps link. Not surprisingly, the majority of the overhead comes from the signatures on individual operations and in prevCommitments. The object history tree contains 14 signatures, and the ACL history contains another four. Together, this many 2048-bit RSA bare signatures would require 4068 bytes, but because Frientegrity employs spliced signatures, they require additional overhead in exchange for faster signing and verification.

7.3 Latency of Fetching a News Feed

To present a user with a news feed, the client must perform one readObject RPC for each of the user's friends, and so we expect the latency of fetching a news feed to scale linearly with the number of friends. Because clients can hide network latency by pipelining requests to the server, we expect the cost of decryption and verification to dominate.

To evaluate the latency of fetching a news feed, we varied the number of friends from 1 to 50. We repeated each experiment 500 times and computed the median of the trials. A linear regression test on the results showed an overhead of 3.557 ms per additional friend (with a correlation coefficient of 0.99981). As expected, the value is very close to the cost of client signature verification and decryption from Figure 5.

Users in social networks may have hundreds of friends, however. In 2011, the average Facebook user had 190 friends, while the 90th percentile of users had 500 friends [19]. With Frientegrity's measured per-object overhead, fetching wall posts from all 500 friends would require approximately 1.8 s. In practice, we expect a social networking site to use modern Web programming techniques (*e.g.*, asynchronous Javascript) so that news feed items could be loaded in the background and updated incrementally while a user stays on a website. Even today, social networking sites often take several seconds to fully load.

7.4 Server Throughput with Many Clients

Social networks must scale to millions of active users. Therefore, to reduce capital and operational costs, it is important that a server be able to maximize throughput while maintaining low latency. To characterize a loaded server's behavior, we evaluated its performance as we increased the number of clients, all issuing requests to the same object. In this experiment, we ran multiple client machines, each with at most 4 clients. Each client issued 3000 requests sequentially, performing a 10 B write with a 1% probability and a read otherwise.

Figure 7 plots server throughput as the number of clients increases, as well as server latency as a function of load. We measured server latency from the time it received a request until the time that it started writing data back to its network socket. The server reached a maximal throughput of handling around 3500 requests per second, while median latency remained below 0.5 ms.

7.5 Effect of Increasing f

Frientegrity supports collaborative verification of object histories. The number of malicious clients that can be tolerated, f, has a large impact on client performance. As f increases, the client has to examine operations further back in the history until it finds $f + 1$ different writers. To understand this effect, we measured the read latency of a single object as f grows.

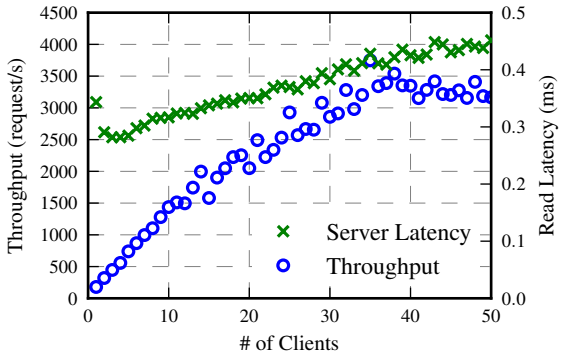


Figure 7: Server performance under increased client load. Each data point is the median of 5 runs.

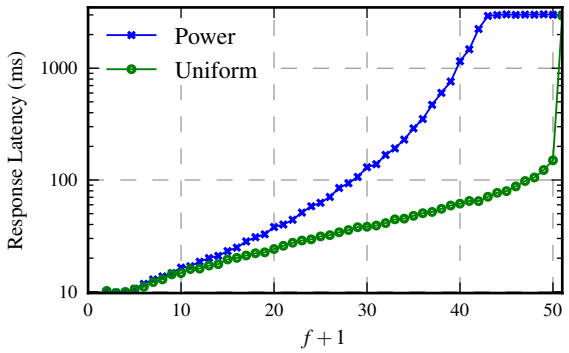


Figure 8: Performance implication of varying minimum set of trusted writers for collaborative verification.

In this experiment, 50 writers first issued 5000 updates to the same object. We evaluated two different workloads for clients. In *uniform*, each writer had a uniform probability (2%) of performing the write; in *power law*, the writers were selected from a power-law distribution with $\alpha = 3.5$ (this particular α was the observed distribution of chat activity among users of Microsoft messaging [32]). A client then issued a read using increasing values of f. Read latencies plotted in Figure 8 are the median of 100 such trials.

In the uniform distribution, the number of required verifications rises slowly. But as $f + 1$ exceeds the number of writers, the client must verify the entire history. For the power law distribution, however, as f increases, the number of required verifications rises more rapidly, and at $f = 42$, the client must verify all 5000 updates. Nevertheless, this experiment shows that Frientegrity can maintain good performance in the face of a relatively large number of malicious users. Even with f at nearly 30, the verification latency was only 100 ms.

7.6 Latency of ACL Modifications

In social networking applications, operations on ACLs must perform well even when ACL sizes reach hundreds

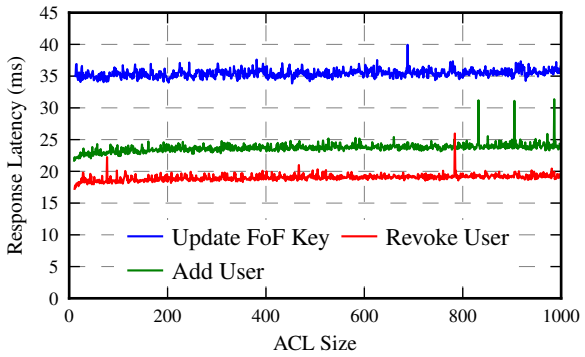


Figure 9: Latency of various ACL operations as a function of number of friends. Friend of Friend updates are measured as time to change a single user's FoF key. Each data point is the mean of 100 runs.

of users. When a user Alice updates her ACL, she first fetches it and its corresponding ACL history and checks that they are consistent. In response, Alice's friend Bob updates the key he shares with friends-of-friends (FoFs). To do so, he fetches and checks Alice's ACL in order retrieve her updated key. He then proceeds to fetch, check, and update his own ACL.

To evaluate the cost of these ACL operations, we measured Frientegrity's performance as two users, Alice and Bob, make changes to their ACLs. While Alice added and removed users from her ACL, Bob updated the key he shares with FoFs. We performed this experiment for different ACL sizes and plotted the results in Figure 9.

As expected, updating the key shared with FoFs was the most costly operation because it requires verifying two ACLs instead of one. Furthermore, adding a new user to an ACL took longer than removing one because it requires a public key encryption. Finally, we observed that although modifying an ACL entails a logarithmic number of symmetric key operations, the cost of these operations was dominated by constant number of public key operations required to verify and update the ACL history.

8. Related Work

Decentralized approaches: To address the security concerns surrounding social networking, numerous works have proposed decentralized designs, in which the social networking service is provided by a collection of federated nodes. In Diaspora [17], perhaps the most well known of these systems, users can choose to store their data with a number of different providers called "pods." In other systems, including Safebook [14], eXO [36], PeerSoN [6], porkut [44], and Confidant [35], users store their data on their own machines or on the machines of their trusted friends, and these nodes are federated via a distributed hash table. Still others, such as PrPl [48] and Vis-à-Vis [49], allow users' data to migrate between users' own machines and trusted third-party infrastructure. We have argued, however, that decentralization is an insufficient approach. A user is left with an unenviable dilemma: either sacrifice availability, reliability, scalability, and convenience by storing her data on her own machine, or entrust her data to one of several providers that she probably does not know or trust any more than she would a centralized provider.

Cryptographic approaches: Many other works aim to protect social network users' privacy via cryptography. Systems such as Persona [5], flyByNight [37], NOYB [25], and Contrail [53] store users' data with untrusted providers but protect its contents with encryption. Others, such as Hummingbird [9], Lockr [56], and systems from Backes *et al.* [3], Domingo-Ferrer *et al.* [18] and Carminati *et al.* [8] attempt to hide a user's social relationships as well, either from the provider or from other users. But, they do not offer any defenses against the sort of traffic analysis we describe in §2.3 other than decentralization. Unlike Frientegrity, in many of these systems (*e.g.*, [5, 9, 37]), "un-friending" requires work that is linear in the number of a user's friends. The scheme of Sun *et al.* [54] is an exception, but it does not support FoFs. EASiER [28] aims to achieve efficient revocation via broadcast encryption techniques and a reencrypting proxy, but when deployed in the DECENT [29] distributed social network, it appears to perform poorly for reasons that are unclear. All of these systems, however, focus primarily on protecting users' privacy while largely neglecting the integrity of users' data. They either explicitly assume that third parties are "honest-but-curious" (*e.g.*, [9, 37]), or they at most employ signatures on individual messages. None deal with the prospect of provider equivocation, however.

Defending against equivocation: Several systems have addressed the threat of server equivocation in network file systems [33, 34], key-value stores [7, 38, 50], and group collaboration [21] by enforcing fork* consistency and related properties. But to enforce fork* consistency, they require clients to perform work that is linear in either the number of users or the number of updates ever submitted to the system. This overhead is impractical in social networks with large numbers of users and in which users typically are interested only in the latest updates.

FETHR [45] is a Twitter-like service that defends against server equivocation by linking a user's posts together with a hash chain as well as optionally entangling multiple users' histories. But besides not supporting access control, it lacks a formal consistency model. Thus, unless a client verifies a user's entire history back to the beginning, FETHR provides no correctness guarantees.

9. Conclusion and Future Work

In designing Frientegrity, we sought to provide a general framework for social networking applications built around an untrusted service provider. The system had to both preserve data confidentiality and integrity, yet also remain efficient, scalable, and usable. Towards these goals, we present a novel method for detecting server equivocation in which users collaborate to verify object histories, and more efficient mechanisms for ensuring fork* consistency based on history trees. Furthermore, we provide a novel mechanism for efficient access control by combining persistent authenticated dictionaries and key graphs.

In addition to introducing these new mechanisms, we evaluate a Frientegrity prototype on synthetic workloads inspired by the scale of real social networks. Even as object histories stretch into the tens of thousands and access control lists into the hundreds, Frientegrity provides response times satisfactory for interactive use, while maintaining strong security and integrity guarantees.

Like other social networking systems that store users' encrypted data with an untrusted provider [5, 25, 37, 53], Frientegrity faces the problem of how such third-party infrastructure would be paid for. It has been suggested that providers would not accept a business model that would prevent them from mining the plaintext of users' data for marketing purposes. Whether this is so has not been well studied. Although there has been some work on privacy-preserving advertising systems[26, 57], the development of business models that can support privacy-preserving services hosted with third-party providers largely remains future work.

Acknowledgments We thank Andrew Appel, Matvey Arye, Wyatt Lloyd, and our anonymous reviewers for their insights and helpful comments. This research was supported by funding from NSF CAREER Award #0953197, an ONR Young Investigator Award, and a gift from Google.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM TOCS*, 14(1), 1996.

[2] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. FOCS*, Oct. 1989.

[3] M. Backes, M. Maffei, and K. Pecina. A security API for distributed social networks. In *Proc. NDSS*, Feb. 2011.

[4] L. Backstrom, C. Dwork, and J. Kleinberg. Wherefore Art Thou R3579X? Anonymized social networks, hidden patterns, and structural steganography. In *Proc. WWW*, May 2007.

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

[6] S. Buchegger, D. Schiöberg, L. hung Vu, and A. Datta. PeerSoN: P2P social networking early experiences and insights. In *Proc. SNS*, Mar. 2009.

[7] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *Proc. DSN*, June 2009.

[8] B. Carminati and E. Ferrari. Privacy-aware collaborative access control in web-based social networks. In *Proc. DBSec*, July 2008.

[9] E. D. Cristofaro, C. Soriente, G. Tsudik, and A. Williams. Hummingbird: Privacy at the time of twitter. Cryptology ePrint Archive, Report 2011/640, 2011. `http://eprint.iacr.org/`.

[10] S. A. Crosby and D. S. Wallach. High throughput asynchronous algorithms for message authentication. Technical Report CS TR10-15, Rice University, Dec. 2010.

[11] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *Proc. USENIX Security*, Aug. 2009.

[12] S. A. Crosby and D. S. Wallach. Super-efficient aggregating history-independent persistent authenticated dictionaries. In *Proc. ESORICS*, Sept. 2009.

[13] S. A. Crosby and D. S. Wallach. Reference implementation of history trees and spliced signatures. `https://github.com/scrosby/fastsig`, Dec. 2010.

[14] L. A. Cutillo, R. Molva, T. Strufe, and T. Darmstadt. Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine*, 47(12):94–101, Dec. 2009.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, Oct. 2007.

[16] S. Deo. protobuf-socket-rpc: Java and python protobuf rpc implementation using TCP/IP sockets (version 2.0). `http://code.google.com/p/protobuf-socket-rpc/`, May 2011.

[17] Diaspora. Diaspora project. `http://diasporaproject.org/`. Retrieved April 23, 2012.

[18] J. Domingo-Ferrer, A. Viejo, F. Sebé, and írsula González-Nicolás. Privacy homomorphisms for social networks with private relationships. *Computer Networks*, 52:3007–3016, Oct. 2008.

[19] Facebook, Inc. Anatomy of facebook. `http://www.facebook.com/notes/facebook-data-team/anatomy-of-facebook/10150388519243859`, Nov. 2011.

[20] Facebook, Inc. Fact sheet. `http://newsroom.fb.com/content/default.aspx?NewsAreaId=22`. Retrieved April 23, 2012.

[21] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proc. OSDI*, Oct. 2010.

[22] Flickr. Flickr phantom photos. `http://flickr.com/help/forum/33657/`, Feb. 2007.

[23] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. SOSP*, Oct. 2003.

[24] Google, Inc. Transparency report. https://www.google.com/transparencyreport/governmentrequests/userdata/. Retrieved April 23, 2012.

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

[26] S. Guha, B. Cheng, and P. Francis. Privad: Practical privacy in online advertising. In *Proc. NSDI*, Mar. 2011.

[27] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12 (3), 1990.

[28] S. Jahid, P. Mittal, and N. Borisov. EASiER: Encryption-based access control in social networks with efficient revocation. In *Proc. ASIACCS*, Mar. 2011.

[29] S. Jahid, S. Nilizadeh, P. Mittal, N. Borisov, and A. Kapadia. DECENT: A decentralized architecture for enforcing privacy in online social networks. In *Proc. SESOC*, Mar. 2012.

[30] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. STOC*, May 1997.

[31] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2): 133–169, 1998.

[32] J. Leskovec and E. Horvitz. Planetary-scale views on a large instant-messaging network. In *Proc. WWW*, Apr. 2008.

[33] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. NSDI*, Apr. 2007.

[34] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. OSDI*, Dec. 2004.

[35] D. Liu, A. Shakimov, R. Cáceres, A. Varshavsky, and L. P. Cox. Confidant: Protecting OSN data without locking it up. In *Proc. Middleware*, Dec. 2011.

[36] A. Loupasakis, N. Ntarmos, and P. Triantafillou. eXO: Decentralized autonomous scalable social networking. In *Proc. CIDR*, Jan. 2011.

[37] M. M. Lucas and N. Borisov. flyByNight: mitigating the privacy risks of social networking. In *Proc. WPES*, Oct. 2008.

[38] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proc. OSDI*, Oct. 2010.

[39] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proc. PODC*, July 2002.

[40] J. P. Mello. Facebook scrambles to fix security hole exposing private pictures. *PC World*, Dec. 2011.

[41] R. C. Merkle. A digital signature based on a conventional encryption function. *CRYPTO*, pages 369–378, 1987.

[42] Mozilla Project. Network security services for Java (JSS). https://developer.mozilla.org/En/JSS. Retrieved April 23, 2012.

[43] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *Proc. IEEE S & P*, May 2009.

[44] R. Narendula, T. G. Papaioannou, and K. Aberer. Privacy-aware and highly-available OSN profiles. In *Proc. WETICE*, June 2010.

[45] D. R. Sandler and D. S. Wallach. Birds of a FETHR: Open, decentralized micropublishing. In *Proc. IPTPS*, Apr. 2009.

[46] R. Sanghvi. Facebook blog: New tools to control your experience. https://blog.facebook.com/blog.php?post=196629387130, Dec. 2009.

[47] E. Schonfeld. Watch out who you reply to on google buzz, you might be exposing their email address. *TechCrunch*, Feb. 2010.

[48] S.-W. Seong, J. Seo, M. Nasielski, D. Sengupta, S. Hangal, S. K. Teh, R. Chu, B. Dodson, and M. S. Lam. PrPl: A decentralized social networking infrastructure. In *Proc. MCS*, June 2010.

[49] A. Shakimov, H. Lim, R. Caceres, L. P. Cox, K. Li, D. Liu, and A. Varshavsky. Vis-à-Vis: Privacy-preserving online social networking via virtual individual servers. In *Proc. COMSNETS*, Jan. 2011.

[50] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proc. CCSW*, Oct. 2010.

[51] S. Song. Why I left Sina Weibo. http://songshinan.blog.caixin.cn/archives/22322, July 2011.

[52] M. Stonebraker. The case for shared nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986.

[53] P. Stuedi, I. Mohomed, M. Balakrishnan, Z. M. Mao, V. Ramasubramanian, D. Terry, and T. Wobber. Contrail: Enabling decentralized social networks on smartphones. In *Proc. Middleware*, Dec. 2011.

[54] J. Sun, X. Zhu, and Y. Fang. A privacy-preserving scheme for online social networks with efficient revocation. In *Proc. INFOCOM*, Mar. 2010.

[55] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP*, Dec. 1995.

[56] A. Tootoonchian, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: Better privacy for social networks. In *Proc. CoNEXT*, Dec. 2009.

[57] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Proc. NDSS*, Feb. 2010.

[58] D. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/. Retrieved April 23, 2012.

[59] C. K. Wong, M. Gouda, and S. S. Lam. Secure group communications using key graphs. *IEEE/ACM TON*, 8(1): 16–30, 1998.

[60] M. Zuckerberg. Facebook S-1: Letter from Mark Zuckerberg. http://sec.gov/Archives/edgar/data/1326801/000119312512034517/d287954ds1.htm#toc287954_10, Feb. 2012.