Privacy and Integrity are Possible in the Untrusted Cloud

Ariel J. Feldman [#], Aaron Blankstein [#], Michael J. Freedman [*], Edward W. Felten [*]

[#] *University of Pennsylvania*
[*] *Princeton University*

Abstract

From word processing to online social networking, user-facing applications are increasingly being deployed in the cloud. These cloud services are attractive because they offer high scalability, availability, and reliability. But adopting them has so far forced users to cede control of their data to cloud providers, leaving the data vulnerable to misuse by the providers or theft by attackers. Thus, users have had to choose between trusting providers or forgoing cloud deployment's benefits entirely.

In this article, we show that it is possible to overcome this trade-off for many applications. We describe two of our recent systems, SPORC [13] and Frientegrity [12], that enable users to benefit from cloud deployment without having to trust providers for confidentiality or integrity. In both systems, the provider only observes encrypted data and cannot deviate from correct execution without detection. Moreover, for cases when the provider does misbehave, SPORC introduces a mechanism, also applicable to Frientegrity, that enables users to recover.

SPORC is a framework that enables a wide variety of collaborative applications such as collaborative text editors and shared calendars with an untrusted provider. It allows concurrent, low-latency editing of shared state, permits disconnected operation, and supports dynamic access control even in the presence of concurrency. Frientegrity extends SPORC's model to online social networking. It introduces novel mechanisms for verifying the provider's correctness and access control that scale to hundreds of friends and tens of thousands of posts while still providing the same security guarantees as SPORC. By effectively returning control of users' data to the users themselves, these systems do much to mitigate the risks of cloud deployment.

1 Introduction

From word processing and calendaring to online social networking, the applications on which end users depend are increasingly being deployed in the cloud. The appeal of cloud-based services is well known. They offer high scalability and availability along with global accessibility and often the convenience of not requiring end users to install any software other than a Web browser. Furthermore, they can enable multiple users to edit shared state concurrently, such as in real-time collaborative text editors.

But by now, it is also well understood that these benefits come at the cost of having to trust the cloud provider with the privacy users' data. The recent history of user-facing cloud services is rife with unplanned data

Copyright 2012 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

disclosures [19], [26], [14], [38], and these services' very centralization of information makes them attractive targets for attack by malicious insiders and outsiders. In addition, providers face pressure from government agencies world-wide to release information on demand, often without search warrants [17], [20], [32]. Finally and perhaps worst of all, providers themselves often have an economic incentive to voluntarily disclose data that their users thought was private. They have repeatedly weakened their privacy policies and default privacy settings to promote new services [29], [30], [31], [39], and frequently stand to gain by selling users' information to marketers.

Less recognized, however, is the extent to which users trust providers with the *integrity* of their data, and the harm that a malicious or compromised provider could do by violating it. A misbehaving provider could corrupt users' information by adding, dropping, modifying, or forging portions of it. 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 [34]. This behavior is an example of provider *equivocation* [25], [22], in which a malicious service presents different clients with divergent views of the system state.

In sum, the emerging class of user-facing cloud services currently requires users to cede control of their data to third-party providers. Users are forced to trust the providers' promises — and perhaps legal and regulatory measures — that have so far failed to adequately safeguard users' data. Thus, users are currently faced with a dilemma: either forgo the advantages of cloud deployment or subject their data to a myriad of new threats.

1.1 Our Approach

In this article, we argue that for many applications, it is possible to overcome this strict trade-off. Rather than forcing users to depend on cloud providers' good behavior, we propose that cloud services should be redesigned so that users can benefit from cloud deployment without having to trust the providers for confidentiality or integrity. Cloud applications should be designed under the assumption that the provider may be actively malicious, and the services' privacy and security guarantees should be rooted in cryptographic keys known only to the users rather than in the providers' promises.

In our recent SPORC [13] and Frientegrity [12] works, we present frameworks for building a wide variety of end user services with untrusted providers. In both systems, the provider observes only encrypted data and cannot deviate from correct execution without detection. Moreover, for cases when the provider does misbehave, SPORC introduces a mechanism, also applicable to Frientegrity, that enables users to recover. It allows users to switch to a new provider and repair any inconsistencies that the provider's equivocation may have caused.

SPORC makes it possible to build applications from collaborative word processors and calendars to email and instant messaging systems that are robust in the face of a misbehaving provider. It allows low-latency editing of shared state, permits disconnected operation, and supports dynamic access control even in the presence of concurrency. Frientegrity extends SPORC's model to online social networking. It introduces novel mechanisms for verifying the provider's correctness and access control that scale to hundreds of friends and tens of thousands of posts while still providing the same security guarantees as SPORC. By effectively returning control of users' data to the users themselves, these systems do much to mitigate the risks of cloud deployment. Thus, they may clear the way for greater adoption of cloud applications.

Road map This article introduces our approach to untrusted, centralized cloud services.[1] Section 2 presents a set of concepts and assumptions that is common to both of our systems. It includes a description of our threat and deployment models and a discussion of *fork* consistency*, a technique for defending against server equivocation. In Sections 3 and 4, we summarize the design, implementation, and evaluation of SPORC and Frientegrity respectively. Finally, in Section 5 we conclude and highlight directions for future work.

[1] For a more complete presentation and related work, see [13], [12], [11].

2 System Model

Traditionally, the cloud provider maintains state that is shared among a set of users. When users read or modify the state, their client devices submit requests to the provider's servers on their behalf. The provider's servers handle these potentially concurrent requests, possibly updating the shared state in the process, and then return responses to the users' clients. Because the shared state is stored on the provider's servers either in plaintext form or encrypted under keys that the provider knows, the provider must be trusted.

In SPORC and Frientegrity, however, users' shared state is encrypted under keys that the provider does not know. As a result, instead of having the servers update the shared state in response to clients' requests, all of the code that handles plaintext runs on the clients. Clients submit encrypted updates, known as *operations*, to the servers, and the servers' role is mainly limited to storing the operations and assigning them a global, canonical order. The servers perform a few other tasks such as rejecting operations that are invalid or that came from unauthorized users. But because the provider is untrusted, the clients must check the servers' output to ensure that they have performed their tasks faithfully.

2.1 Why Have a Centralized Provider?

Because both of our systems limit the provider's role mainly to ordering and storing encrypted operations, one might wonder why they use a centralized provider at all. Indeed, many peer-to-peer group collaboration and social networking systems have been proposed (*e.g.*, [36], [18], [2], [37], [8]). But decentralized schemes have at least two major disadvantages. First, they leave an end 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 federated providers that she probably does not know or trust any more than she would a centralized provider. Second, they are a poor fit for applications in which an online user needs a timely notification that her operation has been committed and will not later be overridden by an operation from another user who is currently offline. For example, to schedule a meeting room, an online user should be able to quickly determine whether her reservation succeeded. Yet in a decentralized system, she could not know the outcome of her request until she had heard from at least a quorum of other users' client. By contrast, a centralized provider can leverage the benefits of cloud deployment—high availability and global accessibility—to achieve timely commits.

2.2 Detecting Provider Equivocation

To prevent a malicious provider from forging or modifying clients' operations without detection, SPORC and Frientegrity clients digitally sign all their operations with their users' private keys. But as we have discussed, signatures are insufficient because a misbehaving provider could still equivocate about the history of operations.

To mitigate this threat, both systems enforce *fork* consistency* [23].[2] 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.

[2]Fork* consistency is a weaker variant of an earlier model called *fork consistency* [25]. 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. Our systems enforce 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.

2.3 Threat Model

Provider We assume that a provider may be actively malicious and may attempt to equivocate or simply deny service. Because clients cannot directly prevent such misbehavior, SPORC and Frientegrity deter it by allowing clients to detect it quickly and providing them with a means to switch to a new provider and repair any inconsistencies in the system's state that the misbehavior may have caused. Both systems prevent the provider from observing the plaintext of users' shared state, and in Frientegrity users are only known to the provider by pseudonym. Nevertheless, a provider may be able to glean some information via traffic analysis and social network deanonymization techniques [1], [27]. A full mitigation of these attacks is beyond the scope of this work.

Users and clients Both systems assume that users may also be malicious and colluding with a malicious provider. As a result, users cannot impersonate other users, decrypt or modify shared state or participate in the consistency protocol unless they have been invited by an authorized user. Frientegrity extends these protections to defend against malicious authorized users. It guarantees fork* consistency as long as the number of misbehaving users with access to a given shared item is less than a predetermined constant.

2.4 Deployment Assumptions

Provider Like traditional providers, providers in our systems would likely employ many servers to support large numbers of users and distinct shared items. Both systems enable such scalability by allowing the provider to partition the systems' state into logically distinct portions that can be managed independently on different servers. In SPORC, each collaboratively-edited piece of state, called a *document*, is entirely self-contained, leading naturally to a shared-nothing architecture [35]. In Frientegrity, each element of each user's social networking profile (*e.g.*, "walls," photo albums, comment threads) can reside on a different servers, and the system minimizes the number of costly dependencies between them. We assume that all of the operations performed on a given shared item are stored and ordered by a single server. But, for reliability, the provider could perform these tasks with multiple servers in a primary/backup or even in a Byzantine fault tolerant configuration.

Users and clients We aim to make realistic assumptions about users' clients. We assume that each user may connect to the provider from multiple client devices (*e.g.*, a laptop, a tablet, and a mobile phone), and that each device has its own separate view of the history of operations. In addition, we do not assume that clients retain any state other than the user's key pair which is used to sign every operation that she creates. We do not even assume that multiple clients are ever online simultaneously, and so our protocols do not rely on consensus among users or clients for correctness.

3 SPORC

SPORC, our first system, is a *generic* collaboration service that makes it possible to build a wide variety of applications such as word processing, calendaring, and instant messaging with an untrusted service provider. It enables a set of authorized users to concurrently edit a shared *document* as well as modify its access control list in real time, all without allowing the provider to compromise the document's confidentiality or integrity. It also allows users to work offline and only later synchronize their changes.

SPORC achieves these properties through a novel combination of *fork* consistency* and *operational transformation (OT)* [9]. Whereas fork* consistency enables clients to detect provider equivocation, OT provides clients with a mechanism to resolve conflicts that result from concurrent edits to the document without having to resort to locking. Perhaps most interestingly, however, OT also allows clients that have detected a malicious fork to switch to a new provider and repair the damage that the fork may have caused. In this way, the same

mechanism that allows SPORC clients to merge correct concurrent operations also enables them to recover from a malicious provider's attacks.

Operational transformation OT provides a general set of rules for synchronizing shared state between clients. In OT, the application defines the set of operations from which all modifications to the document are constructed. When clients generate new operations, they apply them locally before sending them to others. To deal with the conflicts that these optimistic updates inevitably incur, each client *transforms* the operations it receives from others before applying them to its local state. If all clients transform incoming operations appropriately, OT guarantees that they will eventually converge to a consistent, reasonable state.

Central to OT is an application-specific *transformation function* $T(\cdot)$ that allows two clients whose states have diverged by a pair of conflicting operations to return to a consistent state. $T(op_1, op_2)$ takes two operations as input and returns a pair of transformed operations (op_1', op_2'), such that if the party that initially did op_1 now applies op_2', and the party that did op_2 now applies op_1', the conflict will be resolved.

For example [28], suppose Alice and Bob both begin with the same local state "ABCDE", and then Alice applies $op_1 =$ 'del 4' locally to get "ABCE", while Bob performs $op_2 =$ 'del 2' to get "ACDE". If Alice and Bob exchanged operations and executed each others' naively, then they would end up in inconsistent states (Alice would get "ACE" and Bob "ACD"). To avoid this problem, the application supplies the following transformation function that adjusts the offsets of concurrent delete operations:

$$T(\text{del } x, \text{del } y) = \begin{cases} (\text{del } x-1, \text{del } y) & \text{if} \quad x > y \\ (\text{del } x, \text{del } y-1) & \text{if} \quad x < y \\ (\text{no-op}, \text{no-op}) & \text{if} \quad x = y \end{cases}$$

Thus, after computing $T(op_1, op_2)$, Alice will apply $op_2' =$'del 2' as before but Bob will apply $op_1' =$ 'del 3', leaving both in the consistent state "ACE".

Given this pairwise function, clients that diverge in arbitrarily many operations can return to a consistent state by applying it repeatedly. OT works in many settings, as operations, and the transforms on them, can be tailored to each application's requirements. For a collaborative text editor, operations may contain inserts and deletes of character ranges at specific cursor offsets, whereas for a key-value store, operations may contain lists of keys to update or remove.

3.1 SPORC Design

Architecture In SPORC, each client maintains its own copy of the document. When it performs a new operation, the client applies the operation to its local copy first and only later encrypts, digitally signs, and uploads the operation to one of the provider's servers. Upon receiving an encrypted operation, the server commits it to a place in the order of operations on the document and then broadcasts it to the other authorized clients. These clients verify the operation's signature and perform a consistency check (see below) to ensure that the provider has not equivocated about the order of operations. Finally, the clients decrypt the operation and use OT to transform the incoming operation into a form that be applied to their local states. An incoming operation may need to be transformed because the recipient's state may have diverged from the sender's. Other clients' operations may have been committed since the incoming operation was sent. Moreover, the recipient may have pending operations that it has applied locally but that have not yet been committed.

To enforce fork* consistency, each client maintains a *hash chain* over the history of committed operations it has received from the provider. For a set of operations $op_1, \ldots, op_n$, the value of the hash chain up to op_i is given by $h_i = H(h_{i-1}||H(op_i))$, where $H(\cdot)$ is a cryptographic hash function and $||$ denotes concatenation. When a client with history up to op_n submits a new operation, it includes h_n in its message. On receiving the operation, another client can check whether the included h_n matches its own hash chain computation over its local history up to op_n. If they do not match, the client knows that the provider has equivocated.

Access control SPORC allows the administrators of a document to grant and revoke users' access on the fly, but implementing dynamic access control raises several challenges. First, the system must provide an in-band mechanism for distributing encryption keys and supporting key changes when users are removed. Second, it must keep track of causal dependencies between operations and access control list (ACL) changes so that, for example, operations performed after a user has been expelled are guaranteed to be inaccessible to that user. Finally, it must prevent concurrent, conflicting access control changes from corrupting the ACL.

To address these challenges, ACL changes and key distribution are handled via special operations, ModifyUserOps, that are ordered along side ordinary document operations and that are subject to the same consistency guarantees. Adding a user entails submitting a ModifyUserOp containing the symmetric document encryption key encrypted under the new user's public key while removing a user involves picking a new document key for subsequent operations and submitting a ModifyUserOp containing the new key encrypted under the public keys of the remaining users. When creating any operation, including a ModifyUserOp, a client embeds a pointer in it to the last committed operation that the client has seen. Although the client may not know exactly where the operation will end up in the total order, SPORC ensures that it will never be ordered before this pointer, thereby ensuring causal consistency. SPORC also uses these pointers to detect and prevent concurrent, potentially conflicting ACL changes.

Fork recovery In normal operation, SPORC clients are constantly creating small forks between their histories whenever they optimistically apply operations locally, and are then later resolving these forks with OT. From the clients' perspective, a malicious fork caused by an equivocating provider looks similar to these small forks: a history with a common prefix followed by divergent suffixes after the fork point. Thus, SPORC can use OT to resolve a malicious fork in a similar way. To do so, clients essentially treat the operations after the fork as if they were new operations performed locally. A pair of forked clients can switch to a new provider, upload all of their common operations prior to the fork, and then resubmit the operations after the fork as if they were new. The new provider will assign these operations a new order and the clients can then use OT to resolve any conflicts just as they would in normal operation.

3.2 SPORC Implementation & Evaluation

The SPORC framework consists of a server and a client library written in Java that handle synchronization, consistency checking, and access control automatically. The application developer need only supply a data type for operations, a transformation function, and the client-side application. Notably, because the server only handles encrypted data, it is completely generic and need not contain any application-specific code. We used our framework to implement a Web-based, real-time collaborative text editor and a causally-consistent key-value store. In both cases, we were able to create applications robust to a misbehaving provider with only a few hundred lines of code. In particular, we were able to reuse the operations and transformation function from Google Wave [16], an OT-based groupware system with a trusted server, without modification.

To evaluate SPORC's practicality, we performed several microbenchmarks on our prototype implementation on a cluster of commodity machines connected by a gigabit LAN. Our experiments demonstrated that SPORC achieves sufficiently low latency and sufficient throughput to support the user-facing collaborative application for which it was designed. Under load, latency was under 34 ms with up to 16 clients and median server throughput reached 1600 ops/sec. Notably, latency was dominated by the cost of clients' 2048-bit RSA signatures, and thus it could be improved greatly by a faster algorithm such as ESIGN [22]. [3]

[3] Complete results can be found in the full paper [13].

4 Frientegrity

Our second system, Frientegrity, extends SPORC's confidentiality and integrity guarantees to online social networking. It supports the main features of popular social networking applications such as "walls," "news feeds," comment threads, and photos, as well as common access control mechanisms such as "friends," "friends-of-friends (FoFs)," and "followers." But as in SPORC, the provider only sees encrypted data, and clients can collaborate to detect equivocation and other misbehavior such as failing to properly enforce access control.

Frientegrity's design is shaped by social networking's unique scalability challenges. Prior systems that enforced variants of fork* consistency assumed that the number of users would be relatively small or that clients would be connected to the servers most of the time. As a result, to enforce 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 connect only intermittently, and users typically are interested only in the most recent updates, not in the thousands that may have come before. In addition, many previous proposals for secure social networking systems (*e.g.*, [2], [24], [5]) required work that is linear in the number of friends, if not FoFs, to revoke a friend's access (*i.e.*, to "un-friend"). But in real social networks, users may have hundreds of friends and tens of thousands of FoFs [10]. Finally, because popular networking providers have hundreds of millions of users, any consistency and access control mechanisms must be able to function even when the system's state is sharded across many servers.

4.1 Frientegrity Design

A Frientegrity provider runs a set of servers that store *objects*, each of which corresponds to a social networking construct such as a Facebook-like "wall". Like SPORC, clients submit encrypted operations on objects and the provider orders and stores them. The provider also ensures that only authorized clients (*e.g.*, those belonging to a user's friends) can write to each object. To confirm that the provider is fulfilling these roles faithfully, clients collaborate to verify any output that they receive from the provider. Whenever a client performs a read, the provider's response must include enough information to make verification possible. It must also contain the key material that allows a user with an appropriate private key to decrypt the object being read. But because Frientegrity must be scalable, the provider's responses must be structured to allow verification and key distribution to be performed *efficiently*.

For example, if Alice fetches the latest operations from Bob's wall object, the response must allow her to cheaply verify that: (1) the provider has not equivocated about the wall's contents (*i.e.*, enforcing fork* consistency), (2) every operation was created by an authorized user, (3) the provider has not equivocated about the set of authorized users, and (4) the ACL is not outdated.

Enforcing fork* consistency Many prior systems, including SPORC, used hash chains to enforce fork* consistency. But hash chains are a poor fit for social networking applications because verifying an object like Bob's wall would require downloading the entire history of posts and performing linear work on it even though the user is probably only interested in the most recent updates. This cost is further magnified because building a "news feed" requires a user to process all of her friends' walls.

As a result, Frientegrity represents an object's history as a *history tree* [4] rather than a list. A history tree allows multiple clients, each of which may have a different subset of the history, to efficiently compare their views of it. Processing time and the size of messages exchanged are logarithmic in the history size. With a history tree, a client can embed a compact representation of its view of the history [5] in every operation it creates, and clients which subsequently read the operation can compare their views to the embedded one.

[4] A history tree [6] is a growable Merkle hash tree that has been used previously for tamper-evident logging.

[5] *i.e.*, the history tree's current root hash signed by the provider.

Thus, when Alice reads the tail of Bob's wall, she can compare her view of the history with the one embedded in the most recent operation, which perhaps was created by Charlie. If Alice trusts Charlie, then she only has to directly check the operations that were committed since the last operation that Charlie observed. [6] Similarly, before uploading his operation, Charlie only had to check the operations after some earlier snapshot. In this way, no single client needs to examine every operation, and yet by collaborating, clients can verify an object's entire history. Moreover, we can defend against collusion between a misbehaving provider and up to f malicious users by having clients look farther back in the history until they find a point that $f + 1$ clients have vouched for.

Making access control verifiable Bob's profile may be comprised of multiple objects under a single ACL. A well-behaved provider can reject operations from unauthorized users. But because it is untrusted, it must *prove* that it enforced access control correctly on every operation it returns in response to Alice's read. Thus, Frientegrity's ACL data structure must allow the provider to construct efficiently-checkable membership proofs. The ACL must also enable authorized clients to efficiently retrieve the keys necessary to decrypt the relevant objects. Moreover, because social network ACLs may be large, ACL changes and rekeying must be efficient.

To meet these requirements, we represent ACLs with a tree-like data structure that is a novel combination of a *persistent authenticated dictionary* [7] and a *key graph* [40] in which each node is a "friend." A given user's membership proof is simply a path from the root to that user's node and requires space and verification time that is logarithmic in the number of users. In addition, each node stores its user's symmetric key, and the keys are organized so that a user who can decrypt her own node key can follow a chain of decryptions up the tree and obtain the root key which is shared among all authorized users. As a result, adding or removing a user only requires a logarithmic number of keys to be changed along the path from the user's node to the root. Notably, adding or removing a FoF still only requires logarithmic work in the the number of *friends*, not FoFs. Finally, to prevent the provider from equivocating about the history changes to the ACL itself, root hashes of successive versions of the ACL tree are stored in their own fork*-consistent *ACL history* object.

Preventing ACL rollbacks Ideally, Frientegrity would treat every operation performed on every object as a single history and enforce fork* consistency on that history. But, doing so would create extensive, and often unnecessary, dependencies between objects, thereby making it difficult to spread objects across multiple servers without resorting to expensive agreement protocols (*e.g.*, Paxos [21]). Thus, for scalability, Frientegrity orders operations and enforces fork* consistency on each object independently. Weakening consistency across objects leads to new attacks, however. For example, even without equivocating about the contents 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, Frientegrity supports *dependencies* between objects which specify that an operation in one object *happened after* an operation in another. To establish a dependency from object A to object B, a client adds a new operation to A annotated with a compact representation of the client's current view of B. In so doing, the client forces the provider to show anyone who later reads the operation a view of B that is at least as new as the one the client observed. With this mechanism, rollback attacks on Bob's ACL can be defeated by annotating operations in Bob's wall with dependencies on his ACL. Dependencies are also useful in other applications. For example, in a Twitter-like system, every retweet could have a dependency on the tweet to which it refers. In that case, a provider wishing 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), the provider would also have to suppress all subsequent tweets from everyone who retweeted it.

[6] The provider may have committed other operations after Charlie's operation was uploaded but before his operation was committed.

4.2 Frientegrity Implementation & Evaluation

To evaluate Frientegrity's design, we implemented a prototype that simulates a simplified Facebook-like service. Like SPORC, we evaluated our prototype on a cluster of commodity machines connected by a gigabit LAN. We conducted a series of experiments that measured latency, throughput, and network overhead. We found that Frientegrity's method of enforcing fork* consistency outperformed a hash chain-based design by a substantial margin. Whereas Frientegrity achieved latency under 10 ms even for objects comprised of 25,000 operations, a hash chain-based implementation had latency approaching 800 ms for objects containing only 2000 operations. Frientegrity also demonstrated good scalability with large numbers of friends. The latency of modifying an ACL containing up to 1000 friends was below 25 ms. [7]

5 Conclusion & Future Work

SPORC and Frientegrity enable a wide variety of cloud services with untrusted providers by employing a two-pronged strategy. First, to protect the confidentiality of users' information, both systems ensure that providers' servers only observe encrypted data. As a result, not only do they stop the provider from misusing users' data itself, they also prevent users' data from being stolen by malicious insiders or outsiders. Second, to protect the data's integrity, both systems' give clients enough information to check the provider's behavior. Thus, clients can quickly detect any deviation from correct execution, including complex equivocation about the system state.

Nevertheless, due to the diversity of cloud applications, their use cases, and their threat models, our systems can only be considered a small part of a comprehensive mitigation of the risks of cloud deployment. Much work remains, and we discuss two directions for future work here. First, Frientegrity shows that, even within our threat model of an untrusted provider, one size does not fit all. To scale to the demands of online social networking, Frientegrity uses different data structures and a more complex client-server protocol than SPORC. Future work may attempt to extend our systems' guarantees to new applications, but might employ different mechanisms. In so doing, it may shed light on general techniques for developing efficient systems with untrusted parties. Second, although our systems support many useful features, some, such as cross-object search, automatic language translation, and contextual advertising, remain difficult to implement efficiently because the provider cannot manipulate plaintext. Finding practical ways to at least partially support these capabilities would go a long way in spurring the adoption of systems like ours. These solution may well involve algorithms that operate on encrypted data (*e.g.*, [33], [4], [3]) even if fully homomorphic encryption [15] remains impractical.

Acknowledgements

We thank Andrew Appel, Matvey Arye, Christian Cachin, Jinyuan Li, Wyatt Lloyd, Siddhartha Sen, Alexander Shraer, and Alma Whitten for their insights. We also thank the anonymous reviewers of this article and of the prior works upon which it is based. This research was supported by NSF CAREER grant CNS-0953197, an ONR Young Investigator Award, and a gift from Google.

References

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

[2] 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.

[7] Complete results can be found in the full paper [12].

[3] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. *CRYPTO*, pages 535–552, Aug. 2007.

[4] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *Proc. TCC*, Mar. 2006.

[5] 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/.

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

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

[8] Diaspora. Diaspora project. http://diasporaproject.org/. Retrieved Apr. 23, 2012.

[9] C. Ellis and S. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2):399–407, 1989.

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

[11] A. J. Feldman. *Privacy and Integrity in the Untrusted Cloud*. PhD thesis, Princeton University, 2012.

[12] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten. Social networking with Frientegrity: Privacy and integrity with an untrusted provider. In *Proc. USENIX Security*, Aug. 2012.

[13] 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.

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

[15] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. http://crypto.stanford.edu/craig.

[16] Google. Google Wave federation protocol. http://www.waveprotocol.org/federation. Retrieved Apr. 23, 2012.

[17] Google. Transparency report. https://www.google.com/transparencyreport/governmentrequests/userdata/. Retrieved Apr. 23, 2012.

[18] M. Handley and J. Crowcroft. Network text editor: A scalable shared text editor for MBone. In *Proc. SIGCOMM*, Oct. 1997.

[19] J. Kincaid. Google privacy blunder shares your docs without permission. *TechCrunch*, Mar. 2009.

[20] D. Kravets. Aging 'privacy' law leaves cloud e-mail open to cops. *Wired Threat Level Blog*, Oct. 2011.

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

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

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

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

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

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

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

[28] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proc. UIST*, Nov. 1995.

[29] K. Opsahl. Facebook's eroding privacy policy: A timeline. *EFF Deeplinks Blog*, Apr. 2010.

[30] R. Sanghvi. Facebook blog: New tools to control your experience, Dec. 2009.

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

[32] D. J. Solove. A taxonomy of privacy. *University of Pennsylvania Law Review*, 154(3):477–560, Jan. 2006.

[33] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proc. IEEE S & P*, May 2000.

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

[35] M. Stonebraker. The case for shared nothing. *IEEE DEB*, 9(1):4–9, 1986.

[36] 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.

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

[38] U.S. Federal Trade Commission. FTC accepts final settlement with twitter for failure to safeguard personal information. http://www.ftc.gov/opa/2011/03/twitter.shtm, Mar. 2011.

[39] J. Vijayan. 36 state ags blast google's privacy policy change. *Computerworld*, Feb. 2012.

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