Privacy and Integrity in the Untrusted Cloud

Ariel Joseph Feldman

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Advisor: Edward W. Felten

June 2012

© Copyright 2012 by Ariel Joseph Feldman.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/us/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Abstract

Cloud computing has become increasingly popular because it offers users the illusion of having infinite computing resources, of which they can use as much as they need, without having to worry about how those resources are provided. It also provides greater scalability, availability, and reliability than users could achieve with their own resources. Unfortunately, adopting cloud computing has required users to cede control of their data to cloud providers, and a malicious provider could compromise the data's confidentiality and integrity. Furthermore, the history of leaks, breaches, and misuse of customer information at providers has highlighted the failure of government regulation and market incentives to fully mitigate this threat. Thus, users have had to choose between trusting providers or forgoing cloud computing's benefits entirely.

This dissertation aims to overcome this trade-off. We present two systems, SPORC and Frientegrity, that enable users to benefit from cloud deployment *without* having to trust the cloud provider. Their security is rooted not in the provider's good behavior, but in the users' cryptographic keys. 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. It allows users to switch to a new provider and repair any inconsistencies that the provider's misbehavior may have caused.

SPORC is a framework for building a wide variety of user-facing applications from collaborative word processing and calendaring to email and instant messaging 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. Thus, they clear the way for greater adoption of cloud applications.

Acknowledgements

During my time at Princeton, I have benefited from the support, insight, encouragement, and friendship of many people. Not only did they make this dissertation possible, they also made my years in graduate school some of the best of my life.

First, I wish to thank my advisor, Ed Felten. When I initially applied to Princeton, I asked to be considered for its masters program. But Ed knew that what I really wanted was a Ph.D., even before I did, and took the unusual step of getting me admitted to the doctoral program instead. Throughout my graduate career, I have benefited immensely from his insight and his uncanny ability to quickly distill the essence of any complex issue. I have also gained much from his advice — he has always seemed to know the right thing to do in any situation, academic or otherwise.

I also thank Mike Freedman who, as an unofficial second advisor, pointed me towards secure distributed systems. His help not only led to the most productive period of my graduate career, it also resulted in the work that comprises this dissertation.

I have been particularly lucky that my colleagues in my research group and in the Center for Information Technology Policy as a whole were also my friends. Joe Calandrino, Will Clarkson, Ian Davey, Deven Desai, Shirley Gaw, Alex Halderman, Joe Hall, Nadia Heninger, Josh Kroll, Tim Lee, David "The Admiral" Robinson, Steve Schultze, Harlan Yu, the late Bill Zeller, and honorary member Jeff Dwoskin (who also provided this excellent dissertation template) made our lab a free-wheeling place where we were just as likely to play practical jokes and make horrible puns as we were to collaborate on interesting and relevant research. I especially wish to acknowledge Bill's contributions and loyal friendship. He was a coauthor of the SPORC work presented in this dissertation, but he also enriched our lives with his brilliance, humor, generosity, and incisive wit. I miss him dearly.

In addition, I wish to thank Aaron Blankstein, coauthor of this dissertation's Frientegrity work, for his insights, hard work, and friendship. My Princeton experience

was also enhanced by the friendship and research ideas of Tony Capra, Forrester Cole, Wyatt Lloyd, Haakon Ringberg, Sid Sen, Daniel Schwartz-Narbonne, Jeff Terrace, and Yi Wang. Beyond Princeton, I thank my summer mentors, Umesh Shankar of Google and Josh Benaloh of Microsoft Research for their valuable guidance. Special thanks to Yotam Gingold. Although Yotam and I graduated Brown the same year, he began his CS Ph.D. program two years ahead of me, and his advice and example have been invaluable as I navigate the challenges of academia.

I thank my committee — Andrew Appel, Brian Kernighan, and Dave Walker along with Ed and Mike — for their constructive comments throughout the dissertation process. Thanks as well to Melissa Lawson and Laura Cummings-Abdo for their help with all things administrative and for making the CS Department and CITP run smoothly. I am also grateful for the financial support that made this dissertation's research possible: Princeton's Upton Fellowship and grants from Microsoft, Google, and the Office of Naval Research.

Most importantly, I would like to thank my family. I would not have been able to reach this milestone without the education that my parents, Ella and Arthur, provided for me, along with their boundless advice, support, and encouragement. Thanks also to my late grandparents Charles, Lillian, Zecharia, and Rina, as well as to my aunts, uncles, cousins, and many others for their support and their keen interest in my progress over the years. Finally, I would like to thank Racquel for the happiness that she has brought to my life over the past two years. Despite all of the times that I have been stressed or unavailable while finishing this dissertation, she has always supported me. Her love, understanding, and belief in me, even when I doubted myself, has allowed me to achieve this goal.

To my parents, Ella and Arthur.

Contents

Abstract . . . iii

Acknowledgements . . . v

List of Figures . . . xii

1 Introduction **1**

1.1 The Risks of Cloud Deployment . . . 3

1.2 The Limits of Regulation and Market Incentives . . . 5

1.3 Our Approach . . . 6

1.4 Contributions and Road Map . . . 7

2 Background **10**

2.1 Detecting Provider Equivocation . . . 11

2.2 Deployment Assumptions . . . 15

2.2.1 Provider . . . 15

2.2.2 Users and Clients . . . 15

2.3 Threat Model . . . 16

2.4 Design Challenges . . . 19

3 SPORC **21**

3.1 Introduction . . . 21

3.1.1 Contributions and Road Map . . . 22

3.2 System Model . . . 23

3.2.1 Goals . . . 24
3.2.2 Operational Transformation . . . 25
3.2.3 The Benefits of a Centralized Provider . . . 28
3.3 System Design . . . 29
3.3.1 System Overview . . . 29
3.3.2 Operations . . . 32
3.3.3 The Server's Limited Role . . . 33
3.3.4 Sequence Numbers and Hash Chains . . . 33
3.3.5 Resolving Conflicts with OT . . . 34
3.4 Membership Management . . . 36
3.4.1 Encrypting Document Operations . . . 37
3.4.2 Barrier Operations . . . 38
3.5 Extensions . . . 39
3.5.1 Checkpoints . . . 39
3.5.2 Checking for Forks Out-of-Band . . . 41
3.5.3 Recovering from a Fork . . . 42
3.6 Implementation . . . 44
3.6.1 Variants . . . 44
3.6.2 Building SPORC Applications . . . 45
3.7 Experimental Evaluation . . . 46
3.7.1 Latency . . . 47
3.7.2 Server throughput . . . 49
3.7.3 Client time-to-join . . . 50
3.8 Related Work . . . 51
3.9 Conclusion . . . 54

4 Frientegrity 55

4.1 Introduction . . . 55

4.1.1 Road Map 57
4.2 System Model 58
4.2.1 Goals 58
4.2.2 Scaling Fork* Consistency 59
4.2.3 The Limits of User Anonymity 60
4.2.4 Defending Against Malicious Users 61
4.3 System Overview 62
4.3.1 Example: Fetching a News Feed 62
4.3.2 Enforcing Fork* Consistency 64
4.3.3 Making Access Control Verifiable 65
4.3.4 Preventing ACL Rollbacks 66
4.4 System Design 67
4.4.1 Making Objects Verifiable 67
4.4.2 Dependencies Between Objects 72
4.4.3 Verifiable Access Control 73
4.5 Extensions 77
4.5.1 Discovering Friends 77
4.5.2 Multiple Group Administrators 78
4.5.3 Dealing with Conflicts 79
4.5.4 The Possibility of Fork Recovery 79
4.6 Implementation 80
4.7 Experimental Evaluation 82
4.7.1 Single-object Read and Write Latency 82
4.7.2 Latency of Fetching a News Feed 85
4.7.3 Server Throughput with Many Clients 86
4.7.4 Effect of Increasing f 87
4.7.5 Latency of ACL Modifications 88

4.8 Related Work 89
4.9 Conclusion 91

5 Conclusion **93**

5.1 Future Work 95
5.2 Final Remarks 97

Bibliography **99**

List of Figures

3.1 SPORC architecture and synchronization steps 30

3.2 SPORC latency, unloaded and loaded 47

3.3 SPORC server throughput as a function of payload size 49

3.4 SPORC client time-to-join given a variable length history 51

4.1 Frientegrity overview 63

4.2 A pruned object history in Frientegrity 70

4.3 An ACL node in Frientegrity 75

4.4 Distribution of post rates for Twitter users 83

4.5 Read and write latency for Frientegrity as object size increases 84

4.6 Latency of a naive implementation of Frientegrity using hash chains . 85

4.7 Frientegrity server performance under increasing client load 86

4.8 Varying the minimum set of trusted writers in Frientegrity 87

4.9 Latency of Frientegrity ACL operations 89

Chapter 1

Introduction

The past decade has seen the rise of *cloud computing* [75], an arrangement in which businesses and individual users utilize the hardware, storage, and software of third-party companies called *cloud providers* instead of running their own computing infrastructure. Cloud computing offers users the illusion of having infinite computing resources, of which they can use as much or as little as they need, without having to concern themselves with precisely how those resources are provided or maintained [6].

Cloud computing encompasses a wide range of services that vary according to the degree to which they abstract away the details of the underlying hardware and software from users. At the lowest level of abstraction, often referred to as infrastructure as a service, the provider only virtualizes the hardware and storage while leaving users responsible for maintaining the entire software stack from operating system to applications. Examples of such services include Amazon EC2 [69] and competing offerings from IBM [55], Microsoft [79], and Rackspace [91]. At the opposite end of the spectrum, called software as a service, the provider offers specific applications such as word processing, email, and calendaring directly to end users, usually via the Web, and manages all of the necessary hardware and software [75]. Although this category typically refers to services intended to replace desktop applications such as Google

Apps [45] and Microsoft Office Live [78], it can also cover applications with no desktop analogs such as social networking services like Facebook [36] and Twitter [118].

Regardless of type, users have increasingly adopted cloud deployment to perform functions that they cannot carry out themselves or that cloud providers can execute more efficiently. More specifically, cloud computing offers users the following benefits:

Scalability: To operate their own computing infrastructure, users must make a fixed up-front investment in hardware and software. If the demands on their systems later increase, they must invest in additional resources and bear the burden of integrating them with their existing infrastructure. Furthermore, if load subsequently declines, users are left with unused capacity. With cloud deployment, however, users can purchase only the resources they need in small increments and can easily adjust the resources allocated to them in response to changes in demand.

Availability, reliability, and global accessibility: Because cloud providers are in the business of offering computing resources to many customers they typically have greater expertise in managing systems and benefit from greater economies of scale than their users. As a result, their systems often have higher availability and reliability than systems users could field on their own. Moreover, storing users' data on cloud providers' servers allows users to access the data from anywhere without having to run their own always-on server with a globally-reachable IP address.

Maintainability and convenience: By abstracting away the details of the underlying hardware, and in some cases, the software, cloud providers free users from having to maintain those resources. In particular, software-as-a-service users often can use an application without having to explicitly install any software, simply by navigating to Web site.

Unfortunately, these benefits have come at a significant cost. By delegating critical functions to third-party cloud providers, and, in many cases, transferring data that had previously resided on their own systems to providers' servers, users have been

forced to give up control over their data. They must trust the providers to preserve their data's confidentiality and integrity, and a provider that is malicious or has been subject to attack or legal pressure can compromise them. Providers typically promise to safeguard users' data through privacy policies or service level agreements which are sometimes backed by the force of law. But as we explain below, these measures have been inadequate. As a result, users are currently faced with a dilemma: either forgo the many advantages of cloud deployment or subject their data to a myriad of new threats.

1.1 The Risks of Cloud Deployment

The recent history of cloud computing is rife with data leaks, both accidental and deliberate, and has led to a widespread recognition of the privacy risks of cloud deployment. The first of these risks is unplanned data disclosure that occurs as a result of bugs or design errors in a cloud provider's software. For example, a flaw in Google Docs and Spreadsheets allowed documents to be viewed by unauthorized users [59], while two of the most popular photo hosting sites, Flickr and Facebook, have suffered from flaws that have leaked users' private pictures. [76, 40]. Second, cloud providers' centralization of information makes them attractive targets for attack by malicious insiders and outsiders, and their record of protecting users' data has been disappointing. Indeed, in 2011, Twitter reached a settlement with the United States Federal Trade Commission over its lax security practices that allowed outside attackers to impersonate any user in the system and read their private messages [21]. Moreover, numerous sites have experienced break-ins in which users' email addresses, passwords, and credit card numbers were stolen [98, 80].

Third, cloud providers face pressure from from government agencies worldwide to release users' data on-demand. For example, Google receives thousands of requests

per year to turn over their users' private data, and complies with most of them [49]. In addition, several countries including India, Saudi Arabia, the United Arab Emirates, and Indonesia have threatened to block Research in Motion's email service unless it gave their governments access to users' information. [92]. Furthermore, users' data may be disclosed to authorities without users' knowledge, often without warrants. The U.S. Electronic Communications Privacy Act states that a warrant is not required to access data that has been stored on cloud providers' servers for longer than six months [61]. Moreover, under the "third-party doctrine," U.S. courts have held that information stored on providers' servers is not entitled to the Fourth Amendment's protection against unreasonable searches and seizures because by giving their data to third parties voluntarily, users have given up any expectation of privacy [103].

Finally, cloud providers often have an economic incentive to voluntarily disclose data that their users thought was private. Providers such as Google [97, 119] and Facebook [87, 95] have repeatedly weakened their privacy policies and default privacy settings in order to promote new services, and providers frequently stand to gain by selling users' information to marketers. Furthermore, even if users' data is stored with a provider that keeps its promises, the data is still at risk. If the provider is later acquired by another company or its assets are sold in bankruptcy, the new owners could "repurpose" the data.

These risks to the confidentiality of users' data have received much publicity. Less recognized, however, is the extent to which users trust cloud 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 [105]. This behavior is

an example of provider *equivocation* [74, 67], in which a malicious service presents different clients with divergent views of the system state.

1.2 The Limits of Regulation and Market Incentives

One might wonder whether it is possible to address these threats solely through a combination of market incentives and government regulation. Indeed, over the past two decades both strategies have been employed in attempt to protect the privacy of Web users' data. Web sites post privacy policies that ostensibly describe how they use and protect users' data. In addition, government agencies have written guidelines, and in some cases binding rules, that regulate companies privacy and data protection practices [22, 88].

Despite these efforts, we are skeptical that legal or market-based approaches can fully mitigate the risks to users' data by themselves. Market incentives are unlikely to be sufficient because users of cloud services typically lack enough information about providers' privacy and security practices to be able to make an informed choice of provider. The markets for secure software and privacy-preserving Internet businesses have been described as "lemons markets" [12, 120], in which users' inability to distinguish companies that care about privacy and security from those that are just pretending to do so gives companies little incentive to protect users' data. In addition, providers' privacy policies are often too long and complex for users to understand, and seem to be "... designed more to limit companies' liability than to inform consumers about how their information will be used [22]". Furthermore, as stated above, even when users do understand providers' privacy policies, providers often change their privacy practices over time and put users' data to uses that the users never intended.

Legal approaches, have, and will likely continue to be, insufficient on their own as well. Regulators have struggled to devise rules that mandate good behavior on the part of cloud providers without placing undue limits on their business models. They have also had difficulty quantifying the harm caused by privacy breaches [22]. As a result, unless users have sustained identifiable economic damage from a provider's failure to protect their data, they have no guarantee that they will adequately compensated. Thus, despite attempts to regulate providers' behavior through market incentives and government action, users of cloud services are still ultimately forced to rely on providers' promises to protect their data.

1.3 Our Approach

In this dissertation, we propose a different approach. Rather than forcing users to depend on cloud providers' good behavior, we argue 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 might be actively malicious, and the services' privacy and security guarantees should rooted in cryptographic keys known only to the users rather than in the providers' promises.

We focus on user-facing software as a service applications such word processing, calendaring, email, and social networking, and present two systems, SPORC and Frientegrity, that make it possible to build such applications with untrusted providers. In both systems, the provider observes only encrypted data and cannot deviate from correct execution without detection. Furthermore, 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.

In effect, these systems mitigate many of the risks of cloud deployment by returning control over users' data to the users themselves. Because users' data residing on providers' servers is encrypted under keys that are only stored on users' client devices, it is no more vulnerable to theft by attackers than data residing on users' own machines. Additionally, a provider can only utilize users' information in ways that they have approved and cannot turn it over to authorities without the users' cooperation. Users are also not at the mercy of providers' data retention policies; they can destroy the data stored on the provider's servers at any time by simply discarding the keys that they hold.

One may wonder whether cloud providers would adopt such a design. After all, software as a service providers often support their business by mining users' data for marketing purposes. Although supporting advertising with an untrusted provider remains an open problem [117, 50], we believe that there are at least two reasons for providers to consider our approach. First, there are certain potential customers that simply cannot adopt cloud deployment despite its benefits because of the particular sensitivity of their data (*e.g.*, law firms, hospitals, certain businesses with trade secrets). Second, in some cases, providers might consider holding the plaintext of users' data to be a liability. It may be to their advantage to be able to tell government agencies seeking users' data that their systems' design makes it impossible to do so without their users' consent. Also, court decisions or future government regulations that impose high costs on providers for mishandling users' data or failing to prevent breaches might cause providers to view plaintext data as "toxic waste" to be avoided.

1.4 Contributions and Road Map

The remainder of this dissertation is organized as follows. Chapter 2, introduces a set of concepts and assumptions that is common to both of our systems. It includes a

more detailed description of our threat and deployment models as well as a discussion of *fork* and *fork* consistency*, techniques for defending against server equivocation.

In Chapter 3, we present SPORC, a generic framework for building a wide variety of collaborative software as a service applications with an untrusted provider. As we have explained, in SPORC, the provider only observes encrypted data and cannot misbehave without detection. SPORC allows concurrent, low-latency editing of shared state, permits disconnected operation, and supports dynamic access control even in the presence of concurrency. We demonstrate SPORC's flexibility through two prototype applications: a causally consistent key-value store and a browser-based collaborative text editor.

Conceptually, SPORC illustrates the complementary benefits of operational transformation (OT) and fork* consistency. The former allows SPORC clients to execute concurrent operations without locking and to resolve any resulting conflicts automatically. The latter prevents a misbehaving server from equivocating about the order of operations unless it is willing to fork clients into disjoint sets. Notably, unlike previous systems, SPORC can automatically recover from such malicious forks by leveraging OT's conflict resolution mechanism. This work originally appeared in "SPORC: Group Collaboration using Untrusted Cloud Resources," joint work with William P. Zeller, Michael J. Freedman, and Edward W. Felten, which was presented at the 9th Symposium on Operating Systems Design and Implementation (OSDI '10) [39].

Chapter 4 presents Frientegrity, a framework that extends SPORC's confidentiality and integrity guarantees to social networking. 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. This work originally appeared in "Social Networking with Frientegrity: Privacy and Integrity with an Untrusted Provider," joint work with Aaron Blankstein, Michael J. Freedman, and Edward W. Felten, which will appear at the 21st USENIX Security Symposium (Sec '12),

Finally, in Chapter 5, we summarize our conclusions and highlight some potential directions for future work.

Chapter 2

Background

In traditional software-as-a-service applications, 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. The shared state is stored on the provider's servers either in plaintext form or encrypted under keys that the provider knows. This design allows much of the application's code that manipulates the plaintext of users' data to run on the servers. As a result, users have had to trust that this code handles their data properly.

By contrast, SPORC and Frientegrity assume that the provider is untrusted, and so they do not allow the provider's servers to observe the plaintext of users' data. Users' shared state is encrypted under keys that are known only to authorized users and not to the provider. This requirement leads to a different design. 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 check the servers' output to ensure that they have performed their tasks faithfully.

2.1 Detecting Provider Equivocation

In SPORC and Frientegrity, the order that the provider assigns to clients' operations is crucial because it determines the future state of the system. Yet in principle, the provider could order the operations arbitrarily regardless of when they were submitted. Ideally, however, the provider would choose a *linearizable* [53] order in which all clients observe the same history of operations, and every operation is ordered according to when it was submitted unless it was issued concurrently with other operations [68].

A malicious provider could deviate from this ideal behavior, however, by forging or modifying clients' operations. But clients can prevent this sort of tampering by digitally signing every operation they submit, as they do in SPORC and Frientegrity as well as in many prior systems [67, 68]. Unfortunately, signatures are not sufficient for correctness, because as we describe in previous chapter, a misbehaving provider could still *equivocate* and present different clients with divergent views of the history of operations.

Traditionally, distributed systems have employed Byzantine fault tolerance (BFT) [64] to mitigate the threat of equivocation and other forms of misbehavior. In BFT protocols, every function, such as ordering clients' updates, is executed by multiple servers, each ideally from a different trust domain. A client, in turn, only trusts the output of a function if a quorum of servers produced the same result. Unfortunately, BFT systems are inefficient because the require every function to be

executed multiple times on multiple servers. Furthermore, if a quorum of servers is unavailable, the system cannot make progress.

Beginning with the SUNDR distributed file system [67], more recent systems have defended against equivocation without requiring costly duplication by enforcing variants of a property called *fork consistency*. In fork-consistent systems, clients share their individual views of the order of operations by embedding a compact representation of the history they observe in every operation they send. As a result, if clients to whom the provider has equivocated see each others' operations, they will soon 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.

More specifically, in fork-consistent systems, clients enforce the following properties on the history of operations [68]:

1. Every operation must be signed by a valid client.

2. The order that the system assigns to operations originating from a given client must be consistent with the order that the client assigned to those operations.

3. Every client that observes an operation *op* must have observed the same sequence of operations from the beginning of the history up to *op*.

These properties guarantee that the histories of every client that observes an operation *op* have not been forked from each other up to *op*. Furthermore, they imply that if the provider does fork the clients, it cannot allow the clients in one fork group to observe even one subsequent operation from a client in another fork group or else its misbehavior will be detected.

To make fork consistency possible, whenever a client submits an operation *op*, it must include a compact representation of all of the operations it has seen *up to and including op* with the operation. But because multiple clients can submit operations concurrently, the client cannot know a priori exactly which operations will precede *op* in the provider's order. As a result, submitting a new operation in a concurrent, fork-consistent system requires at least two rounds [68]. The client must first request that the provider reserve a place in its order for the new operation. This step informs the client which operations will precede *op* and commits the provider to an order up to *op*. Only then can the client sign and submit *op* along with the order that the provider committed to in the first round.

Unfortunately, not only does requiring two round trips to the server impose a performance penalty on fork consistent systems, it also makes it possible for a client that crashes or refuses to respond between the first and second rounds to prevent the system from making progress. Moreover, even clients that do not fail can harm the system's overall throughput if they are slow to respond between rounds.

In light of these weaknesses, SPORC and Frientegrity clients defend against equivocation by enforcing a slightly weaker property, *fork* consistency*, that enables a one-round submission protocol. In fork* consistent systems, introduced by BFT2F [68], when a client submits *op*, it does not embed its view of the history up to *op* in the operation. Instead, it only includes its view of the history up to last committed operation that it has seen when it is about to submit *op*. We call this last-known operation the prevOp of *op*. As a result, fork* consistency only guarantees that clients that observe *op* have not been forked from each other up to *op*'s prevOp, rather than up to *op* [1]. Thus, in the event of a fork, *op* could be seen by members of multiple

[1]When they introduce BFT2F, Li and Mazières state fork* consistency's guarantee in slightly weaker terms [68]: if clients observe both op' and *op* from the same client, and op' precedes *op*, then the fork* consistency ensures that the clients have not been forked up to op'. Like SPORC and Frientegrity, however, the implementation of BFT2F that the authors actually enforces the stronger guarantee that we describe.

fork groups if op's prevOp precedes the fork. But if the client that created op submits another operation op', its prevOp will necessarily point to an operation at least as new as op, and the provider will be unable to show op' to multiple fork groups without revealing its misbehavior. Once a malicious provider has forked the history, it can get away with showing exactly one operation from each client to multiple fork groups, but no more. Although fork* consistency's guarantee is weaker than that of fork consistency, it is preferable in practice because it still does not allow cheating to go unnoticed for long, and does not suffer from fork consistency's performance and liveness problems.

In BFT2F [68], clients use *hash chains* to construct the compact representation of their observed history that they embed in every operation. A hash chain is a method of incrementally computing the hash of a list of elements. More precisely, if $op_1, \ldots, op_n$ are the operations in the history, h_0 is a constant initial value, and h_i is the value of the hash chain over the history up to op_i, then $h_i = H(h_{i-1}||H(op_i))$, 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 server has equivocated.

To be able to compare an arbitrary incoming h_k with its own view of the history, a client must compute and store h_i for every op_i in the history (*i.e.*, where $0 \leq i \leq n$). As a result, computing hash chains takes linear time and requires clients to possess the entire history. Hash chains are a good fit for SPORC because, with the notable exception of checkpoints (see Section 3.5.1), SPORC requires clients to download and process the entire history anyway in order to participate in the protocol. In social networking applications, however, histories may contain tens of thousands of operations dating back years, and users are typically only interested in the most recent

updates. Thus, for scalability, Frientegrity uses a different approach that allows clients to only download the most recent operations and yet still verify fork* consistency (see Chapter 4).

2.2 Deployment Assumptions

2.2.1 Provider

Scalability: Like traditional cloud providers, a provider in SPORC or Frientegrity would likely run hundreds, if not thousands, of servers to support large numbers of users and many distinct pieces of shared state. 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 [106]. In addition, 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 (see Section 4.2.2).

Reliability: Both SPORC and Frientegrity are compatible with many existing techniques for mitigating nonmalicious failures. In Chapters 3 and 4, we assume that all of the operations within each logically-distinct piece of state 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.

2.2.2 Users and Clients

Multiple clients per user: Each user may connect to the provider from multiple client devices (*e.g.*, a laptop, a tablet, and a mobile phone), perhaps at different

times, and each device has its own separate view of the history of operations. Indeed, a malicious provider might try to fork a given user's clients from each other.

Client local state: Although clients may cache portions of the shared state for efficiency, clients do not necessarily store state for long periods of time. As a result, SPORC and Frientegrity assume that a client may have "forgotten" the whole history each time it connects to the provider, yet still allow it to verify that the provider has not misbehaved.

Keys: Every user has a public/private key pair that she uses to sign every operation that she creates, and all of her clients use the same key pair. The key pair can itself be stored in the cloud encrypted under a strengthened password. Thus, SPORC and Frientegrity can have the same user experience as current Web applications, in which a user only needs her password to use the system.

User and client liveness: Clients are not necessarily online most of the time, and clients that have previously joined the system may leave and never return. (*e.g.*, if a user accesses the system from a public computer that she only uses once). In addition, users who have been granted access to the shared state may never actually access it (*e.g.*, if a user is invited to a collaborative word processing document that she never ends up reading or writing). As a result, our systems do not depend on consensus among users or clients. In fact, neither of our systems' protocols requires multiple clients to be online simultaneously to ensure correctness.

2.3 Threat Model

Provider: As described above, SPORC and Frientegrity assume that the provider may be actively malicious and not just "honest but curious [43]." But although both systems make provider misbehavior detectable, they do not prevent it. A malicious provider could equivocate despite the likelihood of being caught, or it could deny

service by blocking clients' operations or by erasing or refusing to provide users' encrypted state. Fortunately, clients can recover from such misbehavior. As we describe in Section 3.5.3, in the event of equivocation SPORC allows clients to choose a new provider and repair any inconsistencies in the system's state that the fork may have caused. These techniques can be applied to Frientegrity as well (see Section 4.5.4). Furthermore, to mitigate the threat of a provider that erases their data, users could replicate the encrypted operations they observe either locally or on servers run by alternate providers (*e.g.*, Amazon EC2 [69]). Because both systems allow provider misbehavior to be detected quickly, however, we believe that providers will have an incentive to avoid misbehaving out of fear of legal repercussions or damage to their reputations.

In both SPORC and Frientegrity, the provider cannot access the plaintext of users' shared state because it is encrypted under keys known only to the users. But because the provider has access to the size and timing of clients' operations, it may be able to glean information about their contents via traffic analysis. Traffic analysis is made more difficult because operations can be deltas to the shared state that do not reveal which portions of the state they modify. Completely mitigating traffic analysis is beyond the scope of this work, but would likely require padding the length of operations and introducing cover traffic.

SPORC and Frientegrity differ in the extent to which they reveal users' identities to the provider. Whereas in SPORC, the provider identifies users by their real names, in Frientegrity the provider only knows users by their pseudonyms. Even so, as we explain in Section 4.2.3, traffic analysis and social network deanonymization techniques prevent Frientegrity from being able to hide the social graph from the provider.

Users and clients: SPORC and Frientegrity assume that users may also be malicious and may use the clients they control to attempt to read and modify state to which they do not have access. Malicious users may also collude with a malicious

provider or may even be Sybils [34], fake users created by the provider to help it deceive honest users or subvert fork* consistency. As a result, in both systems, users cannot decrypt or modify shared state or participate in the consistency protocol unless they have been invited by another user who already has access.

The two systems differ in the degree to which they trust authorized users, however. In SPORC, authorized users are trusted unless they have been expelled by an administrator user. A malicious authorized user could compromise the confidentiality of the shared state and could collude with a malicious provider to compromise fork* consistency. Nevertheless, the user would still be unable to forge operations on other users' behalf or prevent equivocation from eventually being discovered through honest users' operations. Frientegrity, on the other hand, assumes that some authorized users may be malicious. As Section 4.2.4 explains, Frientegrity guarantees fork* consistency as long as the number of misbehaving users with access to a particular piece of shared state is less than a predetermined constant.

Client code: Both systems' security properties depend on the correctness of the code running on the clients. Unfortunately, in software as a service applications, clients typically download their code from the potentially malicious provider, especially when the client is a Web browser. But by forcing all code that handles plaintext to execute on the clients, SPORC and Frientegrity at least expose it to scrutiny. One could imagine an arrangement under which specific versions of the client code supplied by the provider would be audited and signed by third parties, and in which clients would only run signed code. Although such an arrangement would hardly guarantee that the code is correct, it would prevent a malicious provider from spying on particular clients by sending them compromised code instead of the ordinary software. Furthermore, because both systems have a relatively simple client-server API that mostly involves reading and writing opaque operations, it may be possible

for multiple parties to each write their own version of the client software. Thus, the client code need not come from the provider at all.

2.4 Design Challenges

In sum, assuming an untrusted provider leads us to adopt an architecture for SPORC and Frientegrity that differs substantially from traditional software as a service applications. In both of our systems, the state that users share is comprised of operations that are encrypted under keys known only to the users themselves. The users' clients upload the operations to the providers' servers, the provider orders and stores them, and the clients enforce fork* consistency to allow them to detect provider equivocation.

Putting this design into practice, however, gives rise to several challenges. First, both systems' client-server protocols must be structured so that they give clients enough information to not only verify fork* consistency, but also validate the output of any other tasks the provider performs, such as rejecting invalid operations. Moreover, to be practical, the protocols must allow clients to perform these checks *efficiently*. Second, because clients can submit operations concurrently, conflicts between operations may ensue. Yet the provider, seeing only ciphertext, cannot help resolve them. Thus, both systems must provide a mechanism to enable conflict resolution to be performed entirely by the clients. Third, if operations are to be encrypted under keys only shared with authorized users, both systems must allow only the correct users to efficiently retrieve the necessary keys. In addition, because the set of authorized users can change, and because multiple users may try to modify it simultaneously, the systems must support re-keying that functions even in the face of concurrency. Fourth, because the history of operations can grow long, both systems must enable newly-joined clients and clients that have not connected recently to ef-

ficiently obtain the most recent state of the system. Finally, if the provider does misbehave, the systems should ideally make it possible for users to recover and repair any inconsistencies the provider's misbehavior may caused. The chapters present the design and implementation of SPORC and Frientegrity and describe how they address these and other challenges in greater detail.

Chapter 3

SPORC

3.1 Introduction

In this chapter, we present SPORC, a system that offers managed cloud deployment for group collaboration services, yet does not require users to trust the cloud provider to maintain data privacy or even to operate correctly. SPORC's cloud servers only observe encrypted data, and clients can detect any deviation from correct operation (*e.g.*, adding, modifying, dropping, or reordering operations) and recover from the error. As we have explained previously, SPORC bases its security and privacy guarantees on the security of users' cryptographic keys, and not on the provider's good intentions.

SPORC provides a *generic* collaboration service that supports a wide variety of applications such as word processing, calendaring, and instant messaging. SPORC users can create a document, modify its access control list, edit it concurrently, experience fully automated merging of updates, and even perform these operations while disconnected. Updates to users' shared state are encrypted before being sent to one of the cloud provider's servers. The server assigns a total order to all operations and redistributes the ordered updates to clients. But if a malicious provider drops or

reorders updates, the SPORC clients can detect the provider's misbehavior, switch to a new provider, restore a consistent state, and continue. The same mechanism that allows SPORC to merge correct concurrent operations also enables it to transparently recover from attacks that fork clients' views.

From a conceptual distributed systems perspective, SPORC demonstrates the benefit of combining *operational transformation* [35] and *fork* consistency* protocols [68]. Operational transformation (OT) defines a framework for executing lock-free concurrent operations that both preserves causal consistency and converges to a common shared state. It does so by transforming operations so they can be applied commutatively by different clients, resulting in the same final state. While OT originated with decentralized applications using pairwise reconciliation [35, 58], recent systems like Google Docs [29] have used OT with a trusted central server that orders and transforms clients' operations. Fork* consistency, on the other hand, was introduced as a consistency model for interacting with an untrusted server: If the server causes the views of two clients to diverge, the clients must either never see each others' subsequent updates or else identify the server as faulty.

Recovering from a malicious fork is similar to reconciling concurrent operations in the OT framework. Upon detecting a fork, SPORC clients use OT mechanisms to replay and transform forked operations, restoring a consistent state. Previous applications of fork* consistency [68] could only detect forks, but not resolve them.

3.1.1 Contributions and Road Map

In Section 3.2, we define SPORC's goals, and we introduce operational transformation. In addition, we explain why having a centralized cloud provider is preferable to a fully decentralized design for the collaborative applications that SPORC supports. Section 3.3 presents SPORC's framework and protocols for real-time collaboration. SPORC provides security and privacy against both the potentially malicious provider

and against unauthorized users. In Section 3.4, we explain how SPORC supports dynamic access control, which is challenging because SPORC also supports concurrent operations and offline editing.

Section 3.5 describes several extensions to SPORC that we have designed but not implemented. We show how clients can detect and recover from maliciously instigated forks, and we also present a checkpoint mechanism that reduces saved client state and minimizes the join overhead for new clients. In Section 3.6, we describe our prototype implementation and the prototype applications that we have built using it: a causally consistent key-value store and a browser-based collaborative text editor. These illustrate the extensibility of SPORC's pluggable data model. We have implemented a stand-alone Java client along with a browser-based client that requires no explicit installation. We discuss related work in Section 3.8 and give some final thoughts in Section 3.9.

3.2 System Model

The purpose of SPORC is to allow a group of users who trust each other to collaboratively edit some shared state, which we call the *document*, with the help of an untrusted server. SPORC is comprised of a set of client devices that modify the document on behalf of particular users, and a potentially malicious provider whose main role is to impose a global order on those modifications. The provider receives operations from individual clients, orders them, and then broadcasts them to the other clients. Access to the document is limited to a set of authorized users, but each user may be logged into arbitrarily many clients simultaneously (*e.g.*, her laptop, tablet, and mobile phone). Each client, even if it is controlled by the same user as another client, has its own local view of the document that must be synchronized with all other clients.

3.2.1 Goals

We designed SPORC with the following goals in mind:

Flexible framework for a broad class of collaborative services: Because SPORC uses an untrusted provider which does not see application-level content, the provider's servers are generic and can handle a broad class of applications. On the client side, SPORC provides a library suitable for use by a range of desktop and web-based applications.

Propagate modifications quickly: When a client is connected to the network, its changes to the shared state should propagate quickly to all other clients so that clients' views are nearly identical. This property makes SPORC suitable for building collaborative applications requiring nearly real-time updates, such as collaborative text editing and instant messaging.

Tolerate slow or disconnected networks: To allow clients to edit the document while offline or while experiencing high network latency, clients in SPORC update the document *optimistically*. Every time a client generates a modification, it applies the change to its local state immediately, and only later sends the update to the provider for redistribution. As a result, clients' local views of the document will invariably diverge, and SPORC must be able to resolve these divergences automatically.

Keep data confidential from the provider and unauthorized users: Because the provider is untrusted, operations must be encrypted before being sent to it. For efficiency, the system should use symmetric-key encryption. SPORC must provide a way to distribute this symmetric key to every client of authorized users. When a document's access control list changes, SPORC must ensure that newly added users can decrypt the entire document, and that removed users cannot decrypt any updates subsequent to their expulsion.

Detect a misbehaving provider: Even without access to document plaintext, a malicious provider could still do significant damage by deviating from its assigned role. It could attempt to add, drop, alter, or delay clients' (encrypted) updates, or it could show different clients inconsistent views of the document. SPORC must give clients a means to quickly detect these kinds of misbehavior.

Recover from malicious provider behavior: If clients detect that the provider is misbehaving, clients should be able to fail over to a new provider and resume execution. Because a malicious provider could cause clients to have inconsistent local state, SPORC must provide a mechanism for automatically resolving these inconsistencies.

To achieve these goals, SPORC builds on two conceptual frameworks: *fork* consistency*, which we described in Section 2.1, and *operational transformation*, which we explain below.

3.2.2 Operational Transformation

Operational Transformation (OT) [35] provides a general model for synchronizing shared state, while allowing each client to apply local updates optimistically. In OT, the application defines a 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 state.

Central to OT is an application-specific *transformation function* $T(\cdot)$ that allows two clients whose states have diverged by a single pair of conflicting operations to return to a consistent, reasonable state. $T(op_1, op_2)$ takes two conflicting 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.

To use the example from Nichols *et al.* [85], 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 transformation function, clients that diverge in arbitrarily many operations can return to a consistent state by applying the transformation function repeatedly. For example, suppose that Alice has optimistically applied op_1 and op_2 to her local state, but has yet to send them to other clients. If she receives a new operation op_{new}, Alice must transform it with respect to both op_1 and op_2: she first computes $(op'_{new}, op'_1) \leftarrow T(op_{new}, op_1)$, and then $(op''_{new}, op'_2) \leftarrow T(op'_{new}, op_2)$. This process yields op''_{new}, an operation that Alice has "transformed past" her two local operations and can now apply to her local state.

Throughout this chapter, we use the notation $op' \leftarrow T(op, \langle op_1, \ldots, op_n \rangle)$ to denote transforming op past a sequence of operations $\langle op_1, \ldots, op_n \rangle$ by iteratively

applying the transformation function.[1] Similarly, we define $\langle op'_1, \ldots, op'_n \rangle \leftarrow T(\langle op_1, \ldots, op_n \rangle, op)$ to represent transforming a sequence of operations past a single operation.

Operational transformation can be applied in a wide variety of 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, while for a causally consistent key-value store, operations may contain lists of keys to update or remove. In fact, we have implemented both such systems on top of SPORC, which we describe further in Section 3.6.

For many applications, with a carefully chosen transformation function, OT is able to automatically return divergent clients to a state that is not only consistent, but semantically reasonable as well. But for some applications, such as source-code version control, semantic conflicts must be resolved manually. OT can support such applications through the choice of a transformation function that does not try to resolve the conflict, but instead inserts an explicit conflict marker into the history of operations. A human can later examine the marker and resolve the conflict by issuing new writes. These write operations will supersede the conflicting operations, provided that the system preserves the global order of committed operations and the partial order of each client's operations. Section 3.3 describes how SPORC provides these properties.

While OT was originally proposed for decentralized n-way synchronization between clients, many prominent OT implementations are server-centric, including Jupiter [85] and Google Docs [29]. They rely on the server to resolve conflicts and to maintain consistency, and are architecturally better suited for web services. Un-

[1] Strictly speaking, T always returns a pair of operations. For simplicity, however, we sometimes write T as returning a single operation, especially when the other is unchanged, as in our "delete char" example.

fortunately, a misbehaving server can compromise the confidentiality, integrity, and consistency of the shared state.

In Section 3.3, we describe how SPORC adapts these server-based OT architectures to provide security against a misbehaving server. At a high level, SPORC has each client *simulate* the transformations that would have been applied by a trusted OT server, using the server only for ordering. But we still need to protect against inconsistent orderings, for which we leverage fork* consistency techniques [68].

3.2.3 The Benefits of a Centralized Provider

SPORC uses a centralized untrusted cloud provider, but the provider's main purpose is to order and store client-generated operations. This limited role may lead one to ask whether the server should be removed, leading to a completely peer-to-peer design. Indeed, many group collaboration systems, such as Bayou [115] and Network Text Editor [52], employ decentralized architectures. But these designs are a poor fit for applications in which a user needs a timely notification that her operation has been committed and will not later be overridden by another user's operation. For example, to schedule a meeting room, an online user should be able to quickly determine whether her reservation succeeded without worrying that a request from a client currently offline will later override hers. But in a decentralized system, she could not know the status of her reservation until she had heard from every client that could override her request, or at least from a quorum of all the clients. In an attempt to address this problem, Bayou delegates commits to a statically designated, trusted "primary" peer, which is little different from having a centralized provider.

SPORC, on the other hand, only requires an *untrusted* provider for globally ordering operations. Thus, it can leverage the benefits of a cloud deployment—high availability and global accessibility—to achieve timely commits. We show in Sec-

tion 3.4.2 how SPORC's centralized architecture also helps support dynamic access control and key rotation, even in the face of concurrent membership changes.

3.3 System Design

This section describes SPORC's design in greater detail, including its synchronization mechanisms and the measures that clients implement to detect a malicious provider that may modify, reorder, duplicate, or drop operations. In our description, we focus on the behavior of a single server (among many belonging to the provider) that orders and stores the operations in a single document. For this section, we assume that the set of users and clients editing a given document is fixed; we consider dynamic membership in Section 3.4.

3.3.1 System Overview

The parties and stages involved with SPORC operations are shown in Figure 3.1. At a high level, the local state of a SPORC application is synchronized between multiple clients, using a server to collect updates from clients, order them, then redistribute the client updates to others. There are four types of state in the system:

1. The *local state* is a compact representation of the client's current view of the document (*e.g.*, the most recent version of a collaboratively edited text).

2. The *encrypted history* is the set of operations stored at and ordered by the server. The payloads of operations that change the contents of the document are encrypted to preserve confidentiality. The server orders the operations oblivious to their payloads but aware of the previous operations on which they causally depend.

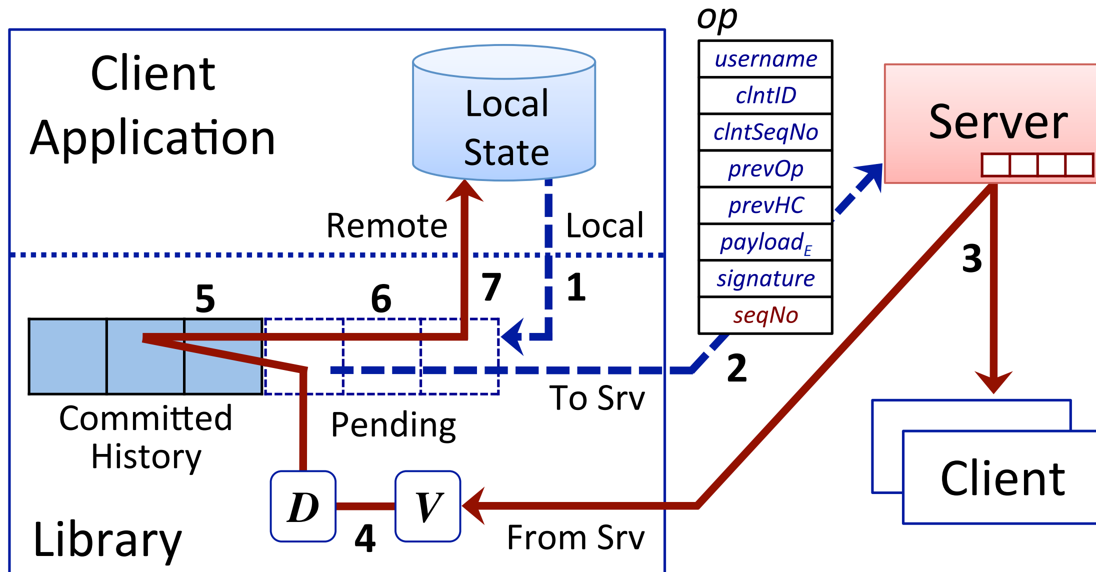


Figure 3.1: SPORC architecture and synchronization steps

3. The *committed history* is the official set of (plaintext) operations shared among all clients, as ordered by the server. Clients derive this committed history from the server's encrypted history by transforming operations' payloads to reflect any changes that the server's ordering might have caused.

4. A client's *pending queue* is an ordered list of the client's local operations that have already been applied to its local state, but that have yet to be committed (*i.e.*, assigned a sequence number by the server and added to the client's committed history).

SPORC synchronizes clients' local state for a particular document using the following steps, also shown in Figure 3.1. This section restricts its consideration to interactions with a static membership and well behaved server; we relax these restrictions in the next two sections, respectively. The flow of local operations to the server is illustrated by dashed blue arrows; the flow of operations received from the server is shown by solid red arrows.

1. A client application generates an operation, applies it to its local state immediately, and then places it at the end of the client's pending queue.

2. If the client does not currently have any operations under submission, it takes its oldest queued operation yet to be sent, *op*, assigns it a client sequence number (clntSeqNo), embeds in it the global sequence number of the last committed operation (prevOp) along with the corresponding hash chain value (prevHC), encrypts its payload, digitally signs it, and transmits it to the server. (As an optimization, if the client has multiple operations in its pending queue, it can submit them as a single batched operation.)

3. The server adds the client-submitted *op* to its encrypted history, assigning it the next available global sequence number (seqNo). The server forwards *op* with this seqNo to all the clients participating in the document.

4. Upon receiving an encrypted operation *op*, the client verifies its signature (V) and checks that its clntSeqNo, seqNo, and prevHC fields have the expected values. If these checks succeed, the client decrypts the payload (D) for further processing. If they fail, the client concludes that the server is malicious.

5. Before adding *op* to its committed history, the client must transform it past any other operations that had been committed since *op* was generated (*i.e.*, all those with global sequence numbers greater than *op*'s prevOp). Once *op* has been transformed, the client appends *op* to the end of the committed history.

6. If the incoming operation *op* was one that the client had initially sent, the client dequeues the oldest element in the pending queue (which will be the uncommitted version of *op*) and prepares to send its next operation. Otherwise, the client transforms *op* past all its pending operations and, conversely, transforms those operations with respect to *op*.

7. The client returns the transformed version of the incoming operation *op* to the application. The application then applies *op* to its local state.

SPORC maintains the following invariants with respect to the system's state:

Local coherence: A client's local state is equivalent to the state it would be in if, starting with an initial empty document, it applied all of the operations in its committed history followed by all of the operations in its pending queue.

Fork* consistency: If the server is well behaved, all clients' committed histories are linearizable (*i.e.*, for every pair of clients, one client's committed history is equal to or a prefix of the other client's committed history). If the server is faulty, however, clients' committed histories may be forked [68].

Client order preservation: As we explain in Section 2.1, fork* consistency implies that the order that a server assigns to operations originating from a given client must be consistent with the order that the client assigned to those operations.

3.3.2 Operations

SPORC clients exchange two types of operations: *document operations*, which represent changes to the content of the document, and *meta-operations*, which represent changes to document metadata such as the document's access control list. Meta-operations are sent to the server in the clear, but the payloads of document operations are encrypted under a symmetric key that is shared among all of the clients but is unknown to the server. (See Section 3.4.1 for a description of how this key is chosen and distributed.) In addition, every operation is labeled with the name of the user that created it and is digitally signed by that user's private key. Each operation also contains a unique client ID (clntID) that identifies the client device from which it came.

3.3.3 The Server's Limited Role

Because the SPORC server is untrusted, its role is limited to ordering and storing the operations that clients submit, most of which are encrypted. The server stores the operations in its encrypted history so that new clients joining the document or existing clients that have been disconnected can request from the server the operations that they are missing. This storage function is not essential, however, and in principle it could be handled by a different party.

Notably, because the server does not have access to the plaintext of document operations, the same generic server implementation can be used for any application that uses our protocol regardless of the kind of document being synchronized.

3.3.4 Sequence Numbers and Hash Chains

SPORC clients use sequence numbers and a hash chain to ensure that operations are properly serialized and that the server is well behaved. Every operation has two sequence numbers: a client sequence number (clntSeqNo) which is assigned by the client that submitted the operation, and a global sequence number (seqNo) which is assigned by the server. On receiving an operation, a client verifies that the operation's clntSeqNo is one greater than the last clntSeqNo seen from the submitting client, and that the operation's seqNo is one greater than the last seqNo that the receiving client saw. These sequence number checks enforce the "client order preservation" invariant and ensure that there are no gaps in the sequence of operations.

When a client uploads an operation op_{new} to the server, the client sets op_{new}'s prevOp field to the global sequence number of the last committed operation, op_n, that the client knows about. The client also sets op_{new}'s prevHC field to the value of the client's hash chain over the committed history up to op_n. A client that receives op_{new} compares its prevHC with the client's own hash chain computation up to op_n. If they

match, the recipient knows that its committed history is identical to the sender's committed history up to op_n, thereby guaranteeing fork* consistency.

A misbehaving server cannot modify the prevOp or prevHC fields, because they are covered by the submitting client's signature on the operation. The server can try to tell two clients different global sequence numbers for the same operation, but this will cause the two clients' histories—and hence their future hash chain values—to diverge, and it will eventually be detected.

To simplify the design, each SPORC client has at most one operation "in flight" at any time: only the operation at the head of a client's pending queue can be sent to the server. Among other benefits, this rule ensures that operations' prevOp and prevHC values will always refer to operations that are in the committed history, and not to other operations that are pending. This restriction could be relaxed, but only at considerable cost in complexity, and so other OT-based systems such as Google Wave adopt it as well [121].

Prohibiting more than one in-flight operation per client is less restrictive than it might seem, as operations can be combined or batched. Like Wave, SPORC includes an application-specific composition function that consolidates a pair of operations into a single logical operation that can be submitted as a unit. This composition function can be applied iteratively to combine an arbitrarily long sequence of operations. As a result, a client can empty its pending queue every time it gets an opportunity to submit an operation to the server.

3.3.5 Resolving Conflicts with OT

Once a client has validated an operation received from the server, the client must use OT to resolve the conflicts that may exist between the new operation and other operations in the committed history and pending queue. These conflicts might have arisen for two reasons. First, the server may have committed additional operations

since the new operation was generated. Second, the receiving client's local state might reflect uncommitted operations that reside on the client's pending queue but that other clients do not yet know about.

Before a client appends an incoming operation op_{new} to its committed history, it compares op_{new}'s prevOp value with the global sequence number of the last committed operation. The prevOp field indicates the last committed operation that the submitting client knew about when it uploaded op_{new}. Thus, if the values match, the client knows that no additional operations have been added to its committed history since op_{new} was generated, and the new operation can be appended directly to the committed history. But if they do not match, then other operations were committed since op_{new} was sent, and op_{new} needs to be transformed past each of them. For example, if op_{new} has a prevOp of 10, but was assigned global sequence number 14 by the server, then the client must compute $op'_{new} \leftarrow T(op_{new}, \langle op_{11}, op_{12}, op_{13} \rangle)$ where $\langle op_{11}, op_{12}, op_{13} \rangle$ are the intervening committed operations. Only then can the resulting transformed operation op'_{new} be appended to the committed history. After appending the operation, the client updates the hash chain computed over the committed history so that future incoming operations can be validated.

At this point, if op'_{new} is one of the receiving client's own operations that it had previously uploaded to the server (or a transformed version of it), it will necessarily match the operation at the head of the pending queue. Because op'_{new} has now been committed, its uncommitted version can be retired from the pending queue, and the next pending operation can be submitted to the server. Furthermore, because the client has already optimistically applied the operation to its local state even before sending it to the server, the client does not need to apply op'_{new} again, and nothing more needs to be done.

If op'_{new} is not one of the client's own operations, however, the client must perform additional transformations in order to reestablish the "local coherence" invariant,

which states that the client's local state is equal to the in-order application of its committed history followed by its pending queue. First, to obtain a version of op'_{new} that it can apply to its local state, the client must transform op'_{new} past all of the operations in its pending queue. This step is necessary because the pending queue contains operations that the client has already applied locally, but have not yet been committed and, therefore, were unknown to the sender of op'_{new}.

Second, the client must transform the entire pending queue with respect to op'_{new} to account for the fact that op'_{new} was appended to the committed history. More specifically, the client computes $\langle \overline{op'_1}, \ldots, \overline{op'_m} \rangle \leftarrow T(\langle \overline{op_1}, \ldots, \overline{op_m} \rangle, op'_{new})$ where $\langle \overline{op_1}, \ldots, \overline{op_m} \rangle$ is the pending queue. This transformation has the effect of pushing the pending queue forward by one operation to make room for the newly extended committed history. The operations on the pending queue need to stay ahead of the committed history because they will receive higher global sequence numbers than any of the currently committed operations. Furthermore, by transforming its unsent operations in response to updates to the document, the client reduces the amount of transformation that other clients will need to perform when they eventually receive its operations.

3.4 Membership Management

Document membership in SPORC is controlled at the level of users, each of which is associated with a public-private key pair. When a document is first created, only the user that created it has access. Subsequently, privileged users can change the document's access control list (ACL) by submitting ModifyUserOp meta-operations, which get added to the document's history (covered by its hash chain), just like normal operations.

A user can be given one of three privilege levels: *reader*, which entitles the user to decrypt the document but not to submit new operations; *editor*, which entitles the user to read the document and to submit new operations (except those that change the ACL); and *administrator*, which grants the user full access, including the ability to invite new users and remove existing users. Because ModifyUserOps are not encrypted, a well-behaved server will immediately reject operations from users with insufficient privileges. But because the server is untrusted, every client maintains its own copy of the ACL, based on the history's ModifyUserOps, and refuses to apply operations that came from unauthorized users.

3.4.1 Encrypting Document Operations

To prevent eavesdropping by the server or unapproved users, the payloads of document operations are encrypted under a symmetric key known only to the document's current members. More specifically, to create a new document, the creator generates a random AES key, encrypts it under her own public key, and then writes the encrypted key to the document's initial create meta-operation. To add new users, an administrator submits a ModifyUserOp that includes the document's AES key encrypted under each of the new users' public keys.

If users are removed, the AES key must be changed so that the removed users will not be able to decrypt subsequent operations. To do so, an administrator picks a new random AES key, encrypts it under the public keys of all the remaining participants, and then submits the encrypted keys as part of the ModifyUserOp.[2] The ModifyUserOp also includes an encryption of the old AES key under the new AES key so that later users can learn earlier keys and thus decrypt old operations without requiring the operations to be re-encrypted.

[2] In our current implementation, the size of a ModifyUserOp may be linear in the number of users participating in the document, because the operation may contain the current AES key encrypted under each of the users' RSA public keys. An optimization to achieve constant-sized ModifyUserOps could instead use a space-efficient broadcast encryption scheme [13].

By tracking prevOps, SPORC preserves causal dependencies between the ModifyUserOps and the other operations in the history. For example, if a ModifyUserOp removes a user, other operations submitted concurrently may be ordered before it and remain accessible to the user. But once a client sees the removal meta-operation in its committed history, any subsequent operation the client submits will be inaccessible to the removed user.

3.4.2 Barrier Operations

Implementing dynamic access control correctly in the face of concurrency is a challenge. Suppose two clients concurrently issue ModifyUserOps, op_1 and op_2, that both attempt to change the current AES key. If the server naively scheduled one after the other, then the continuous chain of old keys encrypted under new ones would be broken.

To address these situations, we introduce a primitive called a *barrier operation*. When the server receives an operation that is marked "barrier" and assigns it global sequence number b, the server requires that every subsequent operation have a prevOp $\geq b$. Subsequent operations that do not are rejected and must be revised and resubmitted with a later prevOp. In this way, the server can force all future operations to depend on the barrier operation.[3]

Returning to the example of the concurrent ModifyUserOps, suppose that op_1 and op_2 were both marked "barrier" and suppose that the server received op_1 first and assigned it sequence number b. Because the operations were submitted concurrently, op_2's prevOp would necessarily be less than b. But because op_1 is a barrier, the server would reject op_2 and would not allow it to be resubmitted until the client that created it updated it to reflect knowledge of op_1 (*i.e.*, by setting op_2's prevOp $\geq b$

[3]To prevent a malicious server from violating the rules governing barrier operations, an operation's "barrier" flag is covered by the operation's signature, and all clients verify that the server is handling barrier operations correctly.

and encrypting op_1's key under op_2's new key). Thus, the chain of old keys encrypted under new ones would be preserved.

Barrier operations have uses beyond membership management. For example, as we describe in the next section, they are useful in implementing checkpoints as well.

3.5 Extensions

This section describes extensions to the basic SPORC protocols: supporting checkpoints to reduce the size requirements for storing the committed history (Section 3.5.1), detecting forks through out-of-band communication (Section 3.5.2), and recovering from forks by replaying and possibly transforming forked operations (Section 3.5.3). Our current prototype does not yet implement these extensions, however.

3.5.1 Checkpoints

To reach a document's latest state, a new client in our current implementation must download and apply the entire history of committed operations. It would be more efficient for a new client to instead download a *checkpoint*—a compact representation of the document's state, akin to each client's local state—and then only apply individual committed operations since the last checkpoint. Much as SPORC servers cannot transform operations, they similarly cannot perform checkpoints; SPORC once again has individual clients play this role.

To support checkpoints, each client maintains a compacted version of the committed history up to the most recent barrier operation. When a client is ready to upload a checkpoint to the server, it encrypts this compacted history under the current document key. It then creates a new CheckpointOp meta-operation containing the hash of the encrypted checkpoint data and submits it into the history. Requiring

the checkpoint data to end in a barrier operation ensures that clients that later use the checkpoint will be able to ignore the history before the barrier without having to worry that they will need to perform OT transformations involving that old history. After all, no operation after a barrier can depend on an operation before it. If the most recent barrier is too old, the client can submit a new null barrier operation before creating the checkpoint.[4]

Checkpoints raise new security challenges, however. A client that lacks the full history cannot verify the hash chain all the way back to the document's creation. It can verify that the operations it has chain together correctly, but the first operation in its history (i.e., the barrier operation) is "dangling," and its prevHC value cannot be verified. This is not a problem if the client knows in advance that the CheckpointOp is part of the valid history, but this is difficult to verify. The CheckpointOp will be signed by a user, and users who have access to the document are assumed to be trusted, but there must be a way to verify that the signing user had permission to access the document at the time the checkpoint was created. Unfortunately, without access to a verifiable history of individual ModifyUserOps going back the beginning of the document, a client deciding whether to accept a checkpoint has no way to be certain of which users were actually members of the document at any given time.

To address these issues, we propose that the server and clients maintain a *meta-history*, alongside the committed history, that is comprised solely of meta-operations. Meta-operations are included in the committed history as before, but each one also has a prevMetaOp pointer to a prior element of the meta-history along with a corresponding prevMetaHC field. Each client maintains a separate hash chain over the

[4]The checkpoint data always ends with an earlier barrier operation rather than with the CheckpointOp itself. This design is preferable because if a checkpoint ended with the CheckpointOp, then the client making the checkpoint would have to "lock" the history to prevent new operations from being admitted before the CheckpointOp was uploaded. Alternatively, the system would have to reject checkpoints that did not reflect the latest state, potentially leading to livelock.

meta-history and performs the same consistency checks on the meta-history that it performs on the committed history.

When a client joins, before it downloads a checkpoint, it requests the entire meta-history from the server. The meta-history provides the client with a fork* consistent view of the sequence of ModifyUserOps and CheckpointOps that indicates whether the checkpoint's creator was an authorized user when the checkpoint was created. Moreover, the cost of downloading the entire meta-history is likely to be low because meta-operations are rare relative to document operations.

Although checkpoints allow new clients or clients that have been offline for a long period of time to synchronize with the current state of the document more efficiently, they do not allow the clients or the server to discard old portions of the history. As we explain below, the individual operations that make up the history might be needed in order to recover from a malicious fork. One could imagine a design in which old history could be discarded after a majority, or even all, of the users participating in a document agreed on a particular checkpoint and signed it. But as we explain in Section 2.2, because we cannot assume that a majority of users or clients will ever access the document, we eschew consensus protocols like this one in SPORC.

3.5.2 Checking for Forks Out-of-Band

Fork* consistency does not prevent a server from forking clients' state, as long as the server never tells any member of one fork about any operation done by a member of another fork. To detect such forks, clients can exchange state information out-of-band, for example, by direct socket connections, email, instant messaging, or posting on a shared server or DHT service.

Clients can exchange messages of the form $\langle c, d, s, h_s \rangle$, asserting that in client c's view of document d, the hash chain value as of sequence number s is equal to h_s. On receiving such a message, a client compares its own hash chain value at

sequence number s with h_s, and if the values differ, it knows a fork has occurred. If the recipient does not yet have operations up to sequence number s, it requests them from the server; a well behaved server will always be able to supply the missing operations.

These out-of-band messages should be digitally signed to prevent forgery. To prevent out-of-band messages from leaking information about which clients are collaborating on a document, and to prevent a client from falsely claiming that it was invited into the document by a forked client, the out-of-band messages should be encrypted and MACed with a separate set of symmetric keys that are known only to nodes that have been part of the document.[5] These keys might be conveyed in the first operation of the document's history.

3.5.3 Recovering from a Fork

A benefit of combining OT and fork* consistency is that we can use OT to recover from forks. OT is well suited to this task because, in normal operation, OT clients are essentially creating small forks whenever they optimistically apply operations locally, and resolving these forks when they transform operations to restore consistency. In this section, we sketch an algorithm that a pair of forked clients can use to merge their divergent histories into a consistent whole. This pairwise algorithm can be repeated as necessary to resolve forks involving multiple clients or multi-way forks.

At a high level, the algorithm proceeds as follows. When a pair of clients detect that they have been forked from each other, they first abandon the malicious provider and agree on a new server operated a different provider. Second, both clients roll back their histories to the last common point before the fork, and then one of them

[5]A client falsely claiming to have been invited into the document in another fork will eventually be detected when the other clients try to recover from the (false) fork. This process is expensive, however, and so we would prefer to avoid it. By protecting the out-of-band messages with symmetric keys known only to clients who have been in the document at some point, we reduce the set of potential liars substantially.

uploads the common history up to the fork point to the new server. Finally, each client resubmits the operations that it saw after the fork as if they were newly created. Upon receiving these operations, the clients transform them as they would any other incoming operations. OT ensures that these resubmitted operations are merged safely so that both nodes end up in the same state.

The situation becomes more complicated if the same operation appears in both histories. We cannot just remove the duplicate because later operations in the sequence may depend on it. Instead, we must cancel it out. To make cancellation possible, we require that all operations be invertible: clients be able to construct an inverse operation op^{-1} such that applying op followed by op^{-1} results in a no-op. Invertible operations must store enough information about the prior state so that a client can determine what the inverse should be. For example, a delete operation can store the information that was deleted, enabling the creation of an insert operation as the inverse.

To cancel each duplicate, we cannot simply splice its inverse into the history right after it for the same reason that we cannot just remove the duplicate. Instead, we compute the inverse operation and then transform it past all of the operations following the duplicate. This process results in an operation that has the effect of canceling out the duplicate when appended to the end of the sequence.

We expect that in the event of a fork, users would typically decide on a new provider manually. But, one could imagine an arrangement in which clients were configured in advance with a preference list of servers and migrated automatically. It is worth noting, however, that switching providers and repairing an inconsistent history are not necessarily possible unless each client backed-up the encrypted operations that it has observed, because a malicious provider might simply fail to supply the complete history. Clients could replicate the operations locally or with a different cloud storage provider.

3.6 Implementation

SPORC provides a framework for building collaborative applications that need to synchronize different kinds of state between clients. It consists of a generic server implementation and a client-side library that implements the SPORC protocol, including the sending, receiving, encryption, and transformation of operations, as well as the necessary consistency checks and document membership management. To build applications within the SPORC framework, a developer only needs to implement client-side functionality that (i) defines a data type for SPORC operations, (ii) defines how to transform a pair of operations, and (iii) defines how to combine multiple document operations into a single one. The server need not be modified, as it always deals with opaque, encrypted operations.

3.6.1 Variants

We implemented two variants of SPORC: a command-line version in which both client and server are stand-alone applications, and a web-based version with a browser-based client and a Java servlet. The command-line version, which we use for later microbenchmarks, is written in approximately 5500 lines of Java code (per SLOCCount [123]) and uses the socket-based RPC library in the open-source release of Google Wave [47]. Because the server's role is limited to ordering and storing client-supplied operations, our prototype implementation is simple and only requires approximately 300 lines of code.

The web-based version shares the majority of its code with the command-line variant. The server just encapsulates the command-line server functionality in a Java servlet, whereas the client consists almost entirely of JavaScript code that was automatically generated using the Google Web Toolkit (GWT) [48]'s Java-to-JavaScript compiler. Network communication uses a combination of the GWT RPC framework,

which wraps browser XmlHttpRequests, and the GWTEventService [107], which allows the server to push messages to the browser asynchronously through a long-lived HTTP connection (the so-called "Comet" style of web programming). To support disconnected operation, our prototype could be extended to use the Web Storage API [54].

The only part of the client that could not be translated directly to JavaScript was the cryptography module. Even with the recent improvements in browsers' JavaScript runtime environments, JavaScript remains too slow to implement public key cryptography efficiently. To work around this limitation, we encapsulate our cryptography module in a Java applet and implement JavaScript-to-Java communication using the LiveConnect API [81] (a strategy employed by Wu [125] and Adida [2]). Relying on LiveConnect is not ideal, however, because different browsers' implementations are often buggy and inconsistent. Our experience suggests it would be beneficial for browsers to provide a JavaScript API that supported basic cryptographic primitives.

3.6.2 Building SPORC Applications

To demonstrate the usefulness of our framework, we built two prototype applications: a causally-consistent key-value store and a web-based collaborative text editor. The key-value store keeps a simple dictionary—mapping strings to strings—synchronized across a set of participating clients. To implement it, we defined a data type that represents a list of keys to update or remove. We wrote a simple transformation function that implements a "last writer wins" policy, as well as a composition function that merges two lists of key updates in a straightforward manner. Overall, the application-specific portion of the key-value store only required 280 lines of code.

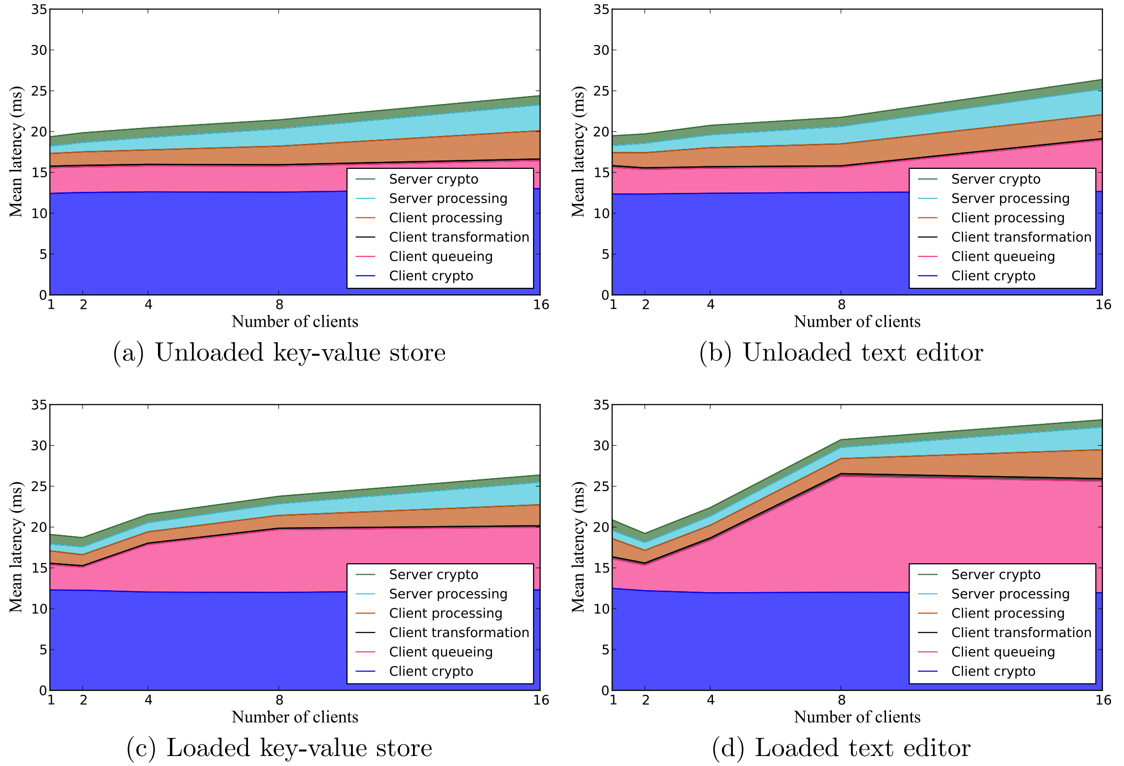
The collaborative editor allows multiple users to modify a text document simultaneously via their web browsers and see each other's changes in near real-time. It provides a user experience similar to Google Docs [46] and EtherPad [44], but, un-

like those services, it does not require a trusted server. To build it, we were able to reuse the data types and the transformation and composition functions from the open-source release of Google Wave. Although Wave is a server-centric OT system without SPORC's level of security and privacy, we were able to adapt its components for our framework with only 550 lines of wrapper code.

3.7 Experimental Evaluation

The user-facing collaborative applications for which SPORC was designed—*e.g.*, word processing, calendaring, and instant messaging—require latency that is low enough for human users to see each others' updates in real-time. But unlike file or storage systems, their primary goal is not high throughput. In this section, we present the results of several microbenchmarks of our Java-based command-line version to demonstrate SPORC's usefulness for this class of applications.

We performed our experiments on a cluster of five commodity machines, each with eight 2.3 GHz AMD Opteron cores and 8 GB of RAM, that were connected by gigabit switched Ethernet. In each of our experiments, we ran a single server instance on its own machine along with varying numbers of client instances. To scale our system to moderate numbers of clients, in many of our experiments, we ran multiple client instances on each machine. We ran all the experiments under the OpenJDK Java VM (version IcedTea6 1.6). For RSA signatures, however, we used the Network Security Services for Java (JSS) library from the Mozilla Project [82] because, unlike Java's default cryptography library, it is implemented in native code and offers considerably better performance.



(a) Unloaded key-value store

(b) Unloaded text editor

(c) Loaded key-value store

(d) Loaded text editor

Figure 3.2: Latency of SPORC while unloaded, with only a single client writer, and loaded, with all clients issuing writes.

3.7.1 Latency

To measure SPORC's latency, we conducted three minute runs with between one and sixteen clients for both key-value and text editor operations. We tested our system under both low-load conditions, where only one of the clients submitted new operations (once every 200 ms), and high-load conditions, where all of the clients were writers. We measured latency by computing the mean time that an operation was "in flight": from the time that it was generated by the sender's application-level code, until the time it was delivered to the recipient's application.

Under low-load conditions with only one writer, we would expect the load on each client to remain constant as the number of clients increases because each additional client does not add to the total number of operations in flight. We would, however,

expect to see server latency increase modestly, as the server has to send operations to increasing numbers of clients. Indeed, as shown in Figure 3.2, the latency due to server processing increased from under 1 ms with one client to over 3 ms with sixteen clients, while overall latency increased modestly from approximately 19 ms to approximately 25 ms.[6]

On the other hand, when every client is a writer, we would expect the load on each client to increase with the number of clients. As expected, Figure 3.2 shows that with sixteen clients under loaded conditions, overall latency is higher: approximately 26 ms for key-value operations and 33 ms for the more expensive text-editor operations. The biggest contributor to this increase is client queueing, which is primarily the time that a client's received operations spend in its incoming queue before being processed. Queueing delay begins at around 3 ms for one client and then increases steadily until it levels off at approximately 8 ms for the key-value application and 14 ms for the text editor. Despite this increase,the figure demonstrates that SPORC successfully supports real-time collaboration for moderately-sized groups, even under load. As these experiments were performed on a local-area network, a wide-area deployment of SPORC would see an increase in latency that reflects the correspondingly higher network round-trip-time.

Figures 3.2 also shows that client-side cryptographic operations account for a large share of overall latency. This occurs because SPORC performs a 2048-bit RSA signature on every outgoing operation and because Mozilla JSS, while better than Java's cryptography built-in library, still requires about 10 ms to compute a single signature. Using an optimized implementation of a more efficient signature scheme, such as ESIGN, could improve the latency of signatures by nearly two orders of magnitude [67].

[6] The figure also shows small increases in the latency of client processing and queuing when the number of clients was greater than four. These increases are most likely due to the fact that, when we conducted experiments with more than four clients, we ran multiple client instances per machine.

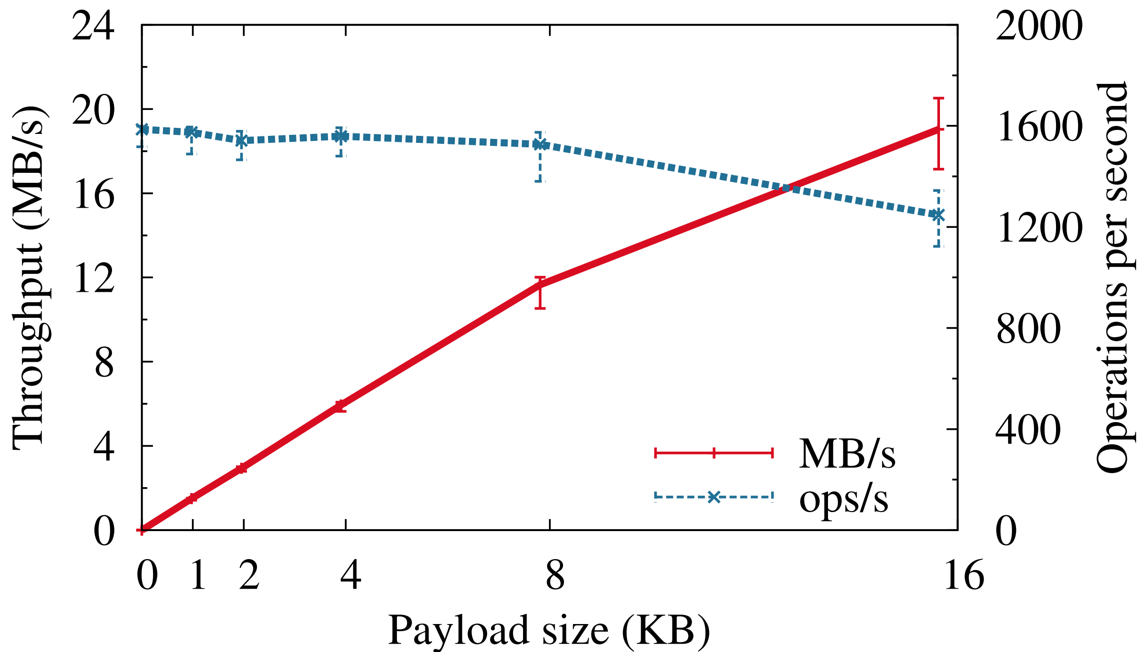


Figure 3.3: SPORC server throughput as a function of payload size

3.7.2 Server throughput

We measured the server's maximum throughput by saturating the server with operations using 100 clients. These particular clients were modified to allow them to have more than one operation in flight at a time. Figure 3.3 shows server throughput as a function of payload size, measured in terms of both operations per second and MB per second. Each data point was computed by performing a three minute run of the system and then taking the median of the mean throughput of each one second interval. The error bars represent the 5th and 95th percentiles. The figure shows that, as expected, when payload size increases, the number of operations per second decreases, because each operation requires more time to process. But, at the same time, data throughput (MB/s) increases because the processing overhead per byte decreases.

3.7.3 Client time-to-join

Because our current implementation lacks the checkpoints of Section 3.5.1, when a client joins the document, it must first download each individual operation in the committed history. To evaluate the cost of joining an existing document, we first filled the history with varying numbers of operations. Then, we measured the time it took for a new client to receive the shared decryption key and download and process all of the committed operations. We performed two kinds of experiments: one where the client started with an empty local state, and a second in which the client had 2000 pending operations that had yet to be submitted to the server. The purpose of the second test was to measure how long it would take for a client that had been working offline for some length of time to synchronize with the current state of the document. Pending operations increase the cost of synchronization because they must be transformed past the committed operations that the client has not seen when the client reconnects to the server. Notably, because the fork recovery algorithm sketched in Section 3.5.3 relies on the same mechanism that is used to synchronize clients that have been offline—it treats operations after the fork as if they were pending uncommitted operations—this test also sheds light on the cost of recovering from a fork.

Figure 3.4 shows time-to-join as a function of history size. Each data point represents the median of ten runs, and the error bars correspond to the 10th and 90th percentiles. We find that time-to-join is linear in the number of committed operations. It takes a client with an empty local state approximately one additional second to join a document for every additional 1000 committed operations.

In addition, the figure shows that the time-to-join with a significant number of pending operations varies greatly by application. In the key-value application, the transformation function is cheap, because it is effectively a no-op if the given operations do not affect the same keys. As a result, the cost of transforming 2000

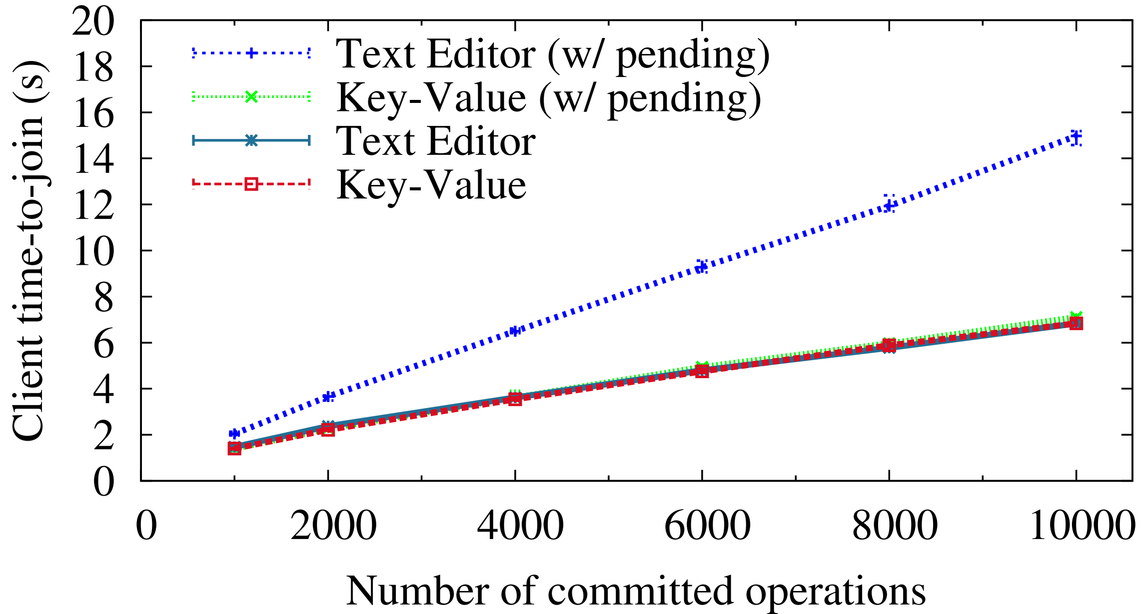


Figure 3.4: SPORC client time-to-join given a variable length history

operations adds little to the time-to-join. By contrast, the text editor's more complex transformation function adds a nontrivial, although still acceptable, amount of overhead.

3.8 Related Work

Real-time "groupware" collaboration systems have adapted classic distributed systems techniques for timestamping and ordering (*e.g.*, [62, 11, 96]), but have also introduced novel techniques to automatically resolve conflicts between concurrent operations in an intention-preserving manner (*e.g.*, [35, 58, 93, 111, 109, 112, 110, 113]). These techniques form the basis of SPORC's client synchronization mechanism and allow it to support slow or disconnected networks. Several systems also use OT to implement undo functionality (*e.g.*, [90, 93]), and SPORC's fork recovery algorithm draws upon these approaches. Furthermore, as an alternative to OT, Bayou [115] allows applications to specify conflict detection and merge protocols to reconcile concurrent operations. Most of these protocols focus on decentralized settings and use

n-way reconciliation, but several well-known systems use a central server to simplify synchronization between clients (including Jupiter [85] and Google Wave [121]). SPORC also uses a central server for ordering and storage, but allows the server to be untrusted. Secure Spread [3] presents several efficient message encryption and key distribution architectures for such client-server group collaboration settings. But unlike SPORC, it relies on trusted servers that can generate keys and re-encrypt messages as needed.

Traditionally, distributed systems have defended against potentially malicious servers by replicating functionality and storage over multiple servers. Protocols, such as Byzantine fault tolerant (BFT) replicated state machines [64, 19, 126] or quorum systems [73, 1], can then guarantee safety and liveness, provided that some fraction of these servers remain non-faulty. Modern approaches optimize performance by, for example, concurrently executing independent operations [60], permitting client-side speculation [122], or supporting eventual consistency [102]. BFT protocols face criticism, however, because when the number of correct servers falls below a certain threshold (typically two-thirds), they cannot make progress.

Subsequently, variants of fork consistency protocols (*e.g.*, [74, 17, 86]) have addressed the question of how much safety one can achieve with a single untrusted server. These works demonstrate that server equivocation can always be detected unless the server permanently forks the clients into groups that cannot communicate with each other. SUNDR [67] and FAUST [16] use these fork consistency techniques to implement storage protocols on top of untrusted servers. Other systems, such as A2M [20] and TrInc [66], rely on trusted hardware to detect server equivocation. BFT2F [68] combines techniques from BFT replication and SUNDR to achieve fork* consistency with higher fractions of faulty nodes than BFT can resist. SPORC borrows from the design of BFT2F in its use of hash chains to limit equivocation, but

unlike BFT2F or any of these other systems, SPORC allows disconnected operation and enables clients to recover from server equivocation, not just detect it.

Like SPORC, two very recent systems, Venus [101] and Depot [72], allow clients to use a cloud resource without having to trust it, and they also support some degree of disconnected operation. Venus provides strong consistency in the face of a potentially malicious server, but does not support applications other than key-value storage. Furthermore, unlike SPORC, it requires the majority of a “core set” of clients to be online in order to achieve most of its consistency guarantees. In addition, although members may be added dynamically to the group editing the shared state, it does not allow access to be revoked, nor does it provide a mechanism for distributing encryption keys. Depot, on the other hand, does not rely on the availability of a “core set” of clients and supports varied applications. Moreover, similar to SPORC, it allows clients to recover from malicious forks using the same mechanism that it uses to keep clients synchronized. But rather than providing a means for reconciling conflicting operations as SPORC does with OT, Depot relies on the application for conflict resolution. Because Depot treats clients and servers identically, it can also tolerate faulty clients, in addition to faulty servers. Unlike SPORC, however, Depot does not consider dynamic access control or confidentiality.

SpiderOak [56] is a commercial cloud file storage provider in which users’ data residing on the company’s servers are encrypted under keys that the company does not know. File encryption keys are encrypted under users’ strengthened passwords, and SpiderOak’s servers only observe encrypted data unless users access their data via the Web or a mobile device. SpiderOak supports data de-duplication, retention of previous file versions, and file sharing between users. The extent to which it protects the integrity of users’ data is unclear, however. It most likely employs digital signatures or MACs on individual files. But to our knowledge, it does not address the

threat of provider equivocation. Moreover, it does not support automatic resolution of the conflicts that arise when multiple clients edit the same data concurrently.

3.9 Conclusion

Our original goal for SPORC was to design a general framework for web-based group collaboration that could leverage cloud resources, but not be beholden to them for privacy guarantees. This goal leads to a design in which servers only store encrypted data, and each client maintains its own local copy of the shared state. But when each client has its own copy of the state, the system must keep them synchronized, and operational transformation provides a way do to so. OT enables optimistic updates and automatically reconciles clients' conflicting states.

Supporting applications that need timely commits requires a centralized provider. But if we do not trust the provider to preserve data privacy, we should not trust it to commit operations correctly either. This requirement led us to employ fork* consistency techniques to allow clients to detect server equivocation about the order of committed operations. But beyond the benefits that each provides independently, this work shows that OT and fork* consistency complement each other well. Whereas prior systems that enforced fork* consistency alone were only able to detect malicious forks, by combining fork* consistency with OT, SPORC can recover from them using the same mechanism that keeps clients synchronized. In addition to these conceptual contributions, we present a membership management architecture that provides dynamic access control and key distribution with an untrusted provider, even in the face of concurrency.

Chapter 4

Frientegrity

4.1 Introduction

Popular social networking sites have hundreds of millions of active users [37]. 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" [127]. But as we have discussed in Chapter 1, it now 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. Less recognized, however, is the extent to which users trust social networking sites with the integrity of their data. Indeed, prior work on secure social networking has focused primarily on privacy and largely neglected integrity. At most, these systems have employed digital signatures on users' individual messages without dealing with the threat of provider equivocation [9, 108, 116, 114].

To address the security concerns surrounding social networking, numerous prior works (*e.g.*, [9, 116, 32]) 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 as in SPORC, 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. Frientegrity remains highly scalable while providing these properties by spreading system state across many shared-nothing servers [106].

Like SPORC and other prior systems [67, 68, 16, 101, 72], Frientegrity employs fork* consistency to enable clients to detect provider equivocation. These earlier systems assumed, however, 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) [38]. Yet, among prior social networking systems that employ encryption for access control (*e.g.*, [9, 71, 23]), 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* [25] and *key graphs* [124].

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.

4.1.1 Road Map

In Section 4.2, we introduce Frientegrity's goals and describe the ways in which Frientegrity's consistency and threat models differ from those of SPORC. Section 4.3 presents an overview of Frientegrity's architecture using the task of fetching a "news feed" as an example. Section 4.4 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. Section 4.5 discusses additional issues for untrusted social networks such as friend discovery and

group administration. We describe our prototype implementation in Section 4.6 and then evaluate its performance and scalability in Section 4.7. We discuss related work in Section 4.8 and then conclude.

4.2 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. As in SPORC, 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.

4.2.1 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 it can perform the necessary verification. Furthermore, because social networking users routinely have hundreds of friends and tens of thousands of friends-of-friends [38], access-control-list changes must be performed in time that is better than linear in the number of users.

4.2.2 Scaling Fork* Consistency

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 [63]) 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 [4, 42] or through hashing [57, 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 Section 4.3.4).

4.2.3 The Limits of User Anonymity

Unlike SPORC, Frientegrity does not directly reveal users' identities to the provider. Users' names are encrypted, and the provider can only identify users by pseudonyms, such as the hash of the public keys they use within the system. Nevertheless, Frientegrity does not attempt 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 [8, 83]).

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.

4.2.4 Defending Against Malicious Users

Frientegrity differs from SPORC in another important respect. Both systems assume that users who have not been invited to access shared state are untrusted and prevent them from compromising the data's confidentiality and integrity. Unless an uninvited user steals an authorized user's private key, she cannot decrypt the shared state or submit operations that other clients will accept, even with the aid of a malicious provider. But whereas SPORC assumes that all *authorized* users are trusted, Frientegrity assumes that some will be malicious (Byzantine faulty). These misbehaving users could attempt to deny service by flooding the shared state with new operations that other users' clients would have to verify. But an honest provider could mitigate this attack via rate-limiting.[1] More significantly, however, malicious authorized users could collude with a malicious provider to subvert fork* consistency by signing multiple inconsistent views of the history of operations.

In Frientegrity, we assume that the number of malicious users with access to a given object is no more than some constant f. As we describe in Section 4.4.1, this assumption allows clients to collaborate to detect provider equivocation. 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. 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). In addition, no matter how many clients a given user operates, she is still considered a single user. Thus, creating enough Sybil users to subvert fork* consistency is expensive because making each Sybil requires tricking

[1] A dishonest provider could deny service without the help of malicious users.

an existing administrator into authorizing it. Furthermore, because the provider establishes the order of operations and creates signed commitments to it, no amount of Sybils is enough to create multiple inconsistent histories without the provider's cooperation.

4.3 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 by issuing a `read`, 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*.

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.

4.3.1 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 4.1.

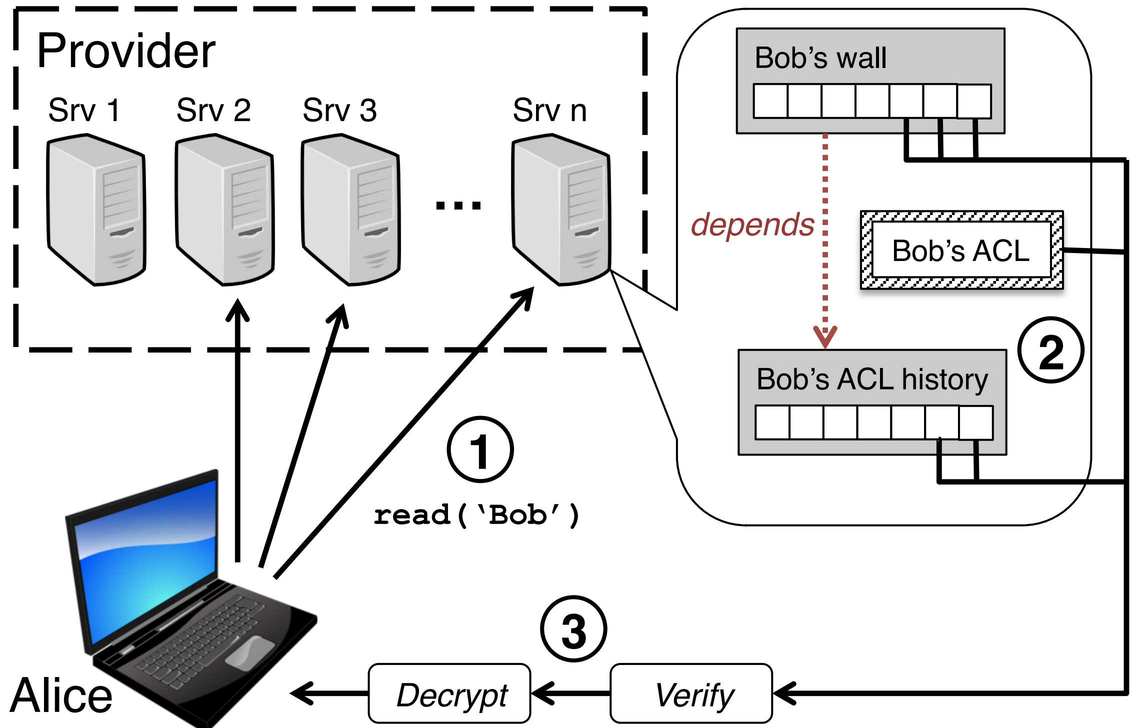


Figure 4.1: A client fetches a news feed in Frientegrity by reading the latest posts from her friends' walls, as well as information to verify, authenticate, and decrypt the posts.

1. For each of Alice's friends, Alice's sends a `read` RPC to the server containing the friend's wall object.

2. In response to a `read` 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 `read`, 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.

4.3.2 Enforcing Fork* Consistency

As we have explained in Section 2.1, 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, including SPORC, 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, 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 Section 4.7.4). We present Frientegrity's collaborative verification algorithm in Section 4.4.1.

4.3.3 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 stores in its ACLs 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 users in social networks may have hundreds of friends and potentially tens of thousands of friends-of-friends [38], adding and removing users 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* [25] and *key graphs* [124]. 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 Section 4.4.3.

4.3.4 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 4.1). 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 Section 4.4.2, we explain the implementation of dependencies and describe additional situations where they can be used.

4.4 System Design

This section describes Frientegrity's design in greater detail. It discusses the algorithms and data structures underlying object verification (Section 4.4.1), dependencies between objects (Section 4.4.2), and verifiable access control Section 4.4.3).

4.4.1 Making Objects Verifiable

Object Representation

To make it possible for clients to verify an object collaboratively in the manner described above, Frientegrity's object representation must allow clients to compare their views of the object's history without requiring either 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.*[24] for tamper-evident logging, is essentially a versioned Merkle tree [77]. 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 then recomputing the hashes of the internal nodes.

Frientegrity uses history trees as follows. Upon receiving a new operation, 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, the server responds with a pruned copy of the object's history tree containing only a subset of the operations, 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. Like the prevOp field in SPORC operations, Frientegrity's prevCommitment field indicates the last committed operation that Alice observed when she submitted her operation. If Bob later reads the object, which by then has version $j > i$, 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.

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, each client executes the following algorithm to verify an object that it has fetched. Every operation that the client checks must be included in the pruned object history tree that the provider sends to the client, along with any extra operations that the client specifically requests. 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 4.2. 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}. 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.)

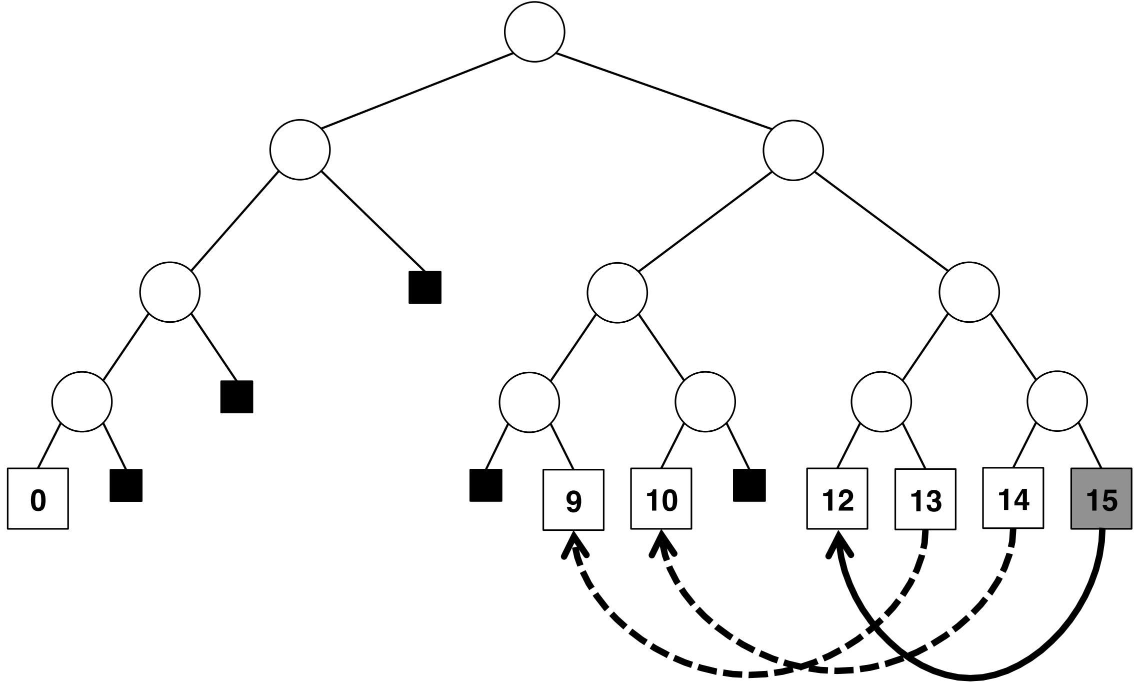


Figure 4.2: A pruned object history that a provider might send to a client. Numbered leaves represent operations and filled boxes represent stubs of omitted subtrees. The solid arrow represents the last operation's prevCommitments. Dashed arrows represent other prevCommitments.

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.[2] Thus, to make sure that the provider did not give her the wrong object, Alice checks that op_0 has the value she expects.

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

[2]Specifically, an object's ID 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.

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 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 specify 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 Section 4.7.4.

4.4.2 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 Section 4.3.4, 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.[3] 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

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

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* [24] 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.

4.4.3 Verifiable Access Control

Supporting Membership Proofs

As explained earlier, because the provider is untrusted, 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. A dictionary data structure that supports both membership proofs and queries on previous versions is called a *persistent authenticated dictionary (PAD)*. Thus, in Frientegrity, ACLs are PADs.

To support PAD functionality, 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.[4] To prove that an entry u exists, it suffices for the ACL to return a pruned tree

[4]Our ACL construction expands on a PAD design from Crosby *et al.* [25] that is based on a *treap* [5]. 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

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 a node needs to be updated, rather than modifying it, the ACL copies the node, 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.

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 Section 4.4.1.

keys and according to the heap property. If nodes' priorities are chosen pseudorandomly, the tree will be balanced in expectation.

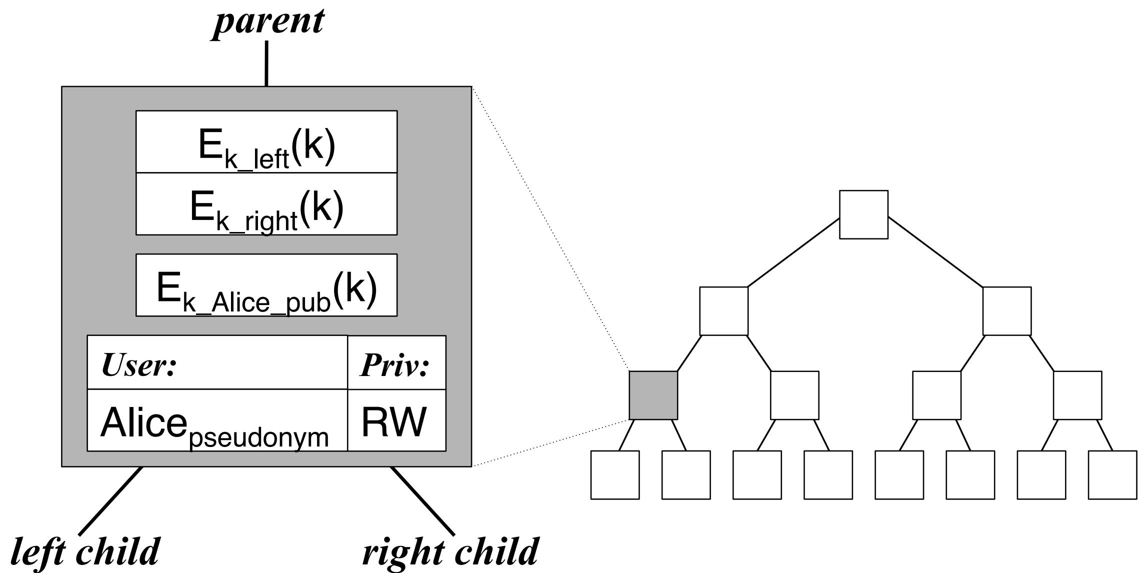


Figure 4.3: ACLs are organized as trees for logarithmic access time. This figure illustrates Alice's entry in Bob's ACL.

Efficient Key Management and Revocation

Like many prior systems designed for untrusted servers (*e.g.*, [9, 71, 51]), Frientegrity protects the confidentiality of users' data by encrypting it under a key that is only shared 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* [124]. But rather than maintaining a separate data structure for keys, Frientegrity exploits the existing structure of ACL PADs. As shown in Figure 4.3, 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}.[5] This structure allows any user in Bob's ACL to follow a chain of decryptions up

[5]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}.

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.

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 $k_{Alice_FoF_pub}$.

4.5 Extensions

4.5.1 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.[6] 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.

[6]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 [95].

4.5.2 Multiple Group Administrators

As we describe in Section 4.4.3, 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 [68] and SPORC. 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.

4.5.3 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 (as we employ in SPORC), or custom merge procedures [115]. 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.

4.5.4 The Possibility of Fork Recovery

If clients detect that a malicious provider has forked the histories of some of the objects to which they have access, then they could, in principle, recover using the same method as SPORC (see Section 3.5.3). To do so, they would migrate each forked object to a new provider, and then apply SPORC's fork recovery algorithm to each object individually. For each object, clients would first identify the fork point and then have one client upload all of the operations before the fork to the new provider. Then, each client would treat every operation that it observed after the fork as if it were new and submit it the replacement provider so that it could be assigned a new place in the order. Finally, the clients would delete any duplicate operations in the new history either by simply removing them, or if operational transformation is used, uploading inverse operations. Each operation would have to be uploaded to the new provider individually so that the new provider would have an opportunity to generate a new commitment to each version of the history, and the clients would have an opportunity to replace the operations' old prevCommitments with ones from the new provider.

Dependencies between objects are the main challenge in applying SPORC-style for recovery to Frientegrity, however. Every object that depends on an object that

was forked must also be migrated to a new provider and repaired. Every individual dependency from that object to the forked object must be recomputed to account for the changes in the forked object's history. More specifically, a dependency that points to $\langle$ dstObj, dstVers, dstCommitment $\rangle$ must be updated to $\langle$ dstObj, dstVers $'$, dstCommitment $'$ $\rangle$ because the target operation in dstObj may have been moved to dstVers $'$ with a new commitment dstCommitment $'$ in the process of resolving the fork. Furthermore, not only must the dependant object be repaired, but every other object that depends on *it* as well. Thus, a fork in a single object may cause multiple objects to need repair.

At the very least, because many of the objects that make up a user's profile depend on each other, fork recovery would likely entail moving a user's entire profile to a new provider, something the user would probably do anyway. But if the application calls for dependencies between objects controlled by different users, such as when a user "retweets" another user's post, this process may require a migrating and repairing a large number of objects. Indeed, if the network of dependencies is extensive, it could require fixing a significant fraction of the objects in the system. Unfortunately, this consequence may be unavoidable because the more interconnected the social network, the farther inconsistency is likely to spread, and the harder it is likely be to repair. As a result, a straightforward application of SPORC's fork recovery algorithm is probably only practical in applications with relatively few dependencies between objects. Hence, we leave recovery of inconsistency that propagates to a large percentage of the social graph to future work.

4.6 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 [123]) and uses the protobuf-socket-rpc [31] 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.* [27].

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 [82] 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* [26, 27]. 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.

4.7 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.

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.

4.7.1 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 4.4 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. As shown in Figure 4.5, write latency was approximately 11 ms (as it includes both a server and client sig-

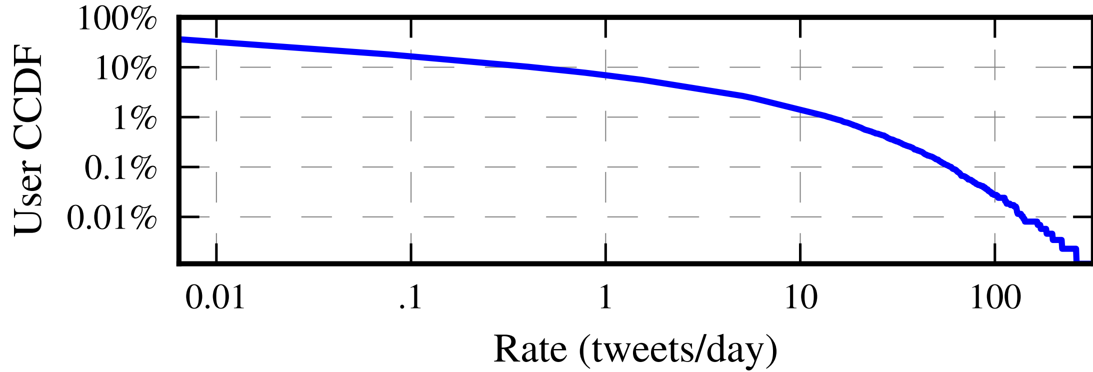
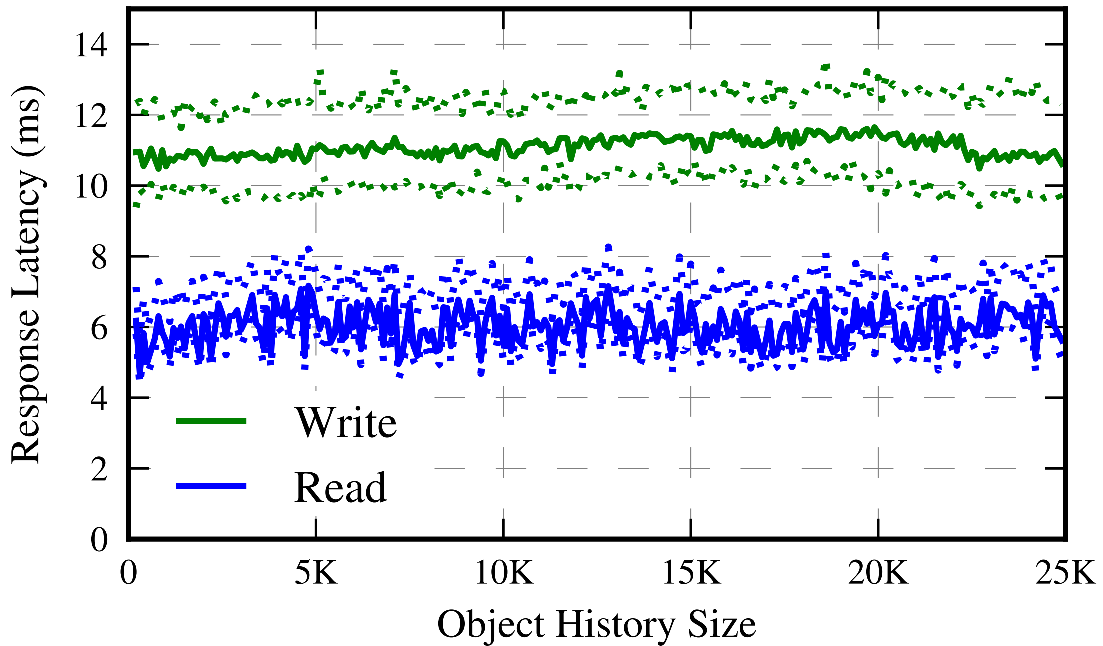


Figure 4.4: Distribution of post rates for Twitter users. 1% of users post at least 14 times a day, while 0.1% post at least 56 times a day.

nature 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 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 4.6. Given this linear growth in latency, verifying an object with history size of 25,000 operations would take approximately 11.6 s in the implementation based on a hash chain compared to Frientegrity's 11 ms.

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 opera-



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 4.5: 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.

tions. Even if clients were stateful, however, Figure 4.4 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.

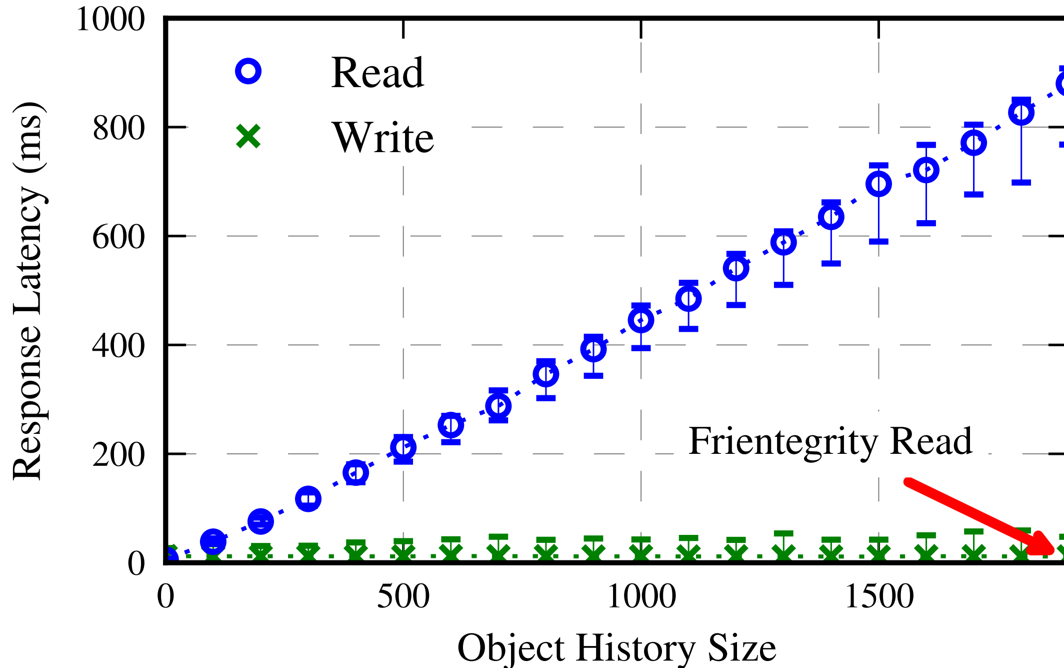


Figure 4.6: Latency for requests in a naive implementation using hash chains. The red arrow indicates the response time for Frientegrity read requests at an object size of 2000. Each data point is the median of 100 requests. The error bars indicate the 90th and 10th percentiles.

4.7.2 Latency of Fetching a News Feed

To present a user with a news feed, the client must perform one read request 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 3.765 ms per additional friend (with a correlation coefficient of 0.99997). As expected, the value is very close to the cost of client signature verification and decryption from Figure 4.5.

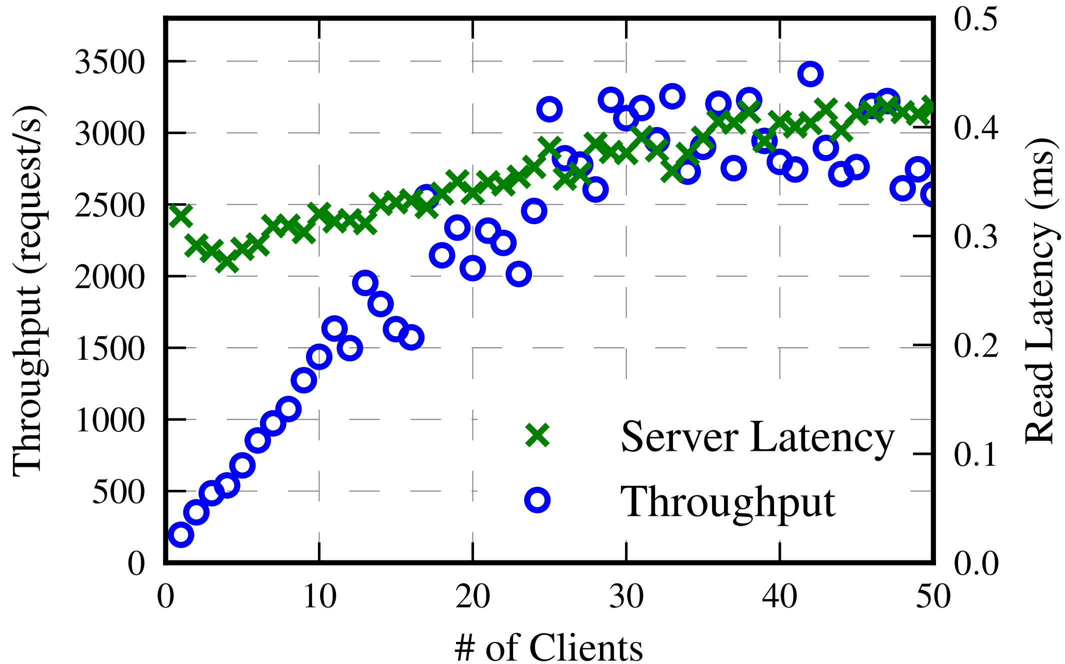


Figure 4.7: Frientegrity server performance under increasing client load

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 [38]. With Frientegrity's measured per-object overhead, fetching wall posts from all 500 friends would require approximately 1.9s. 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.

4.7.3 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

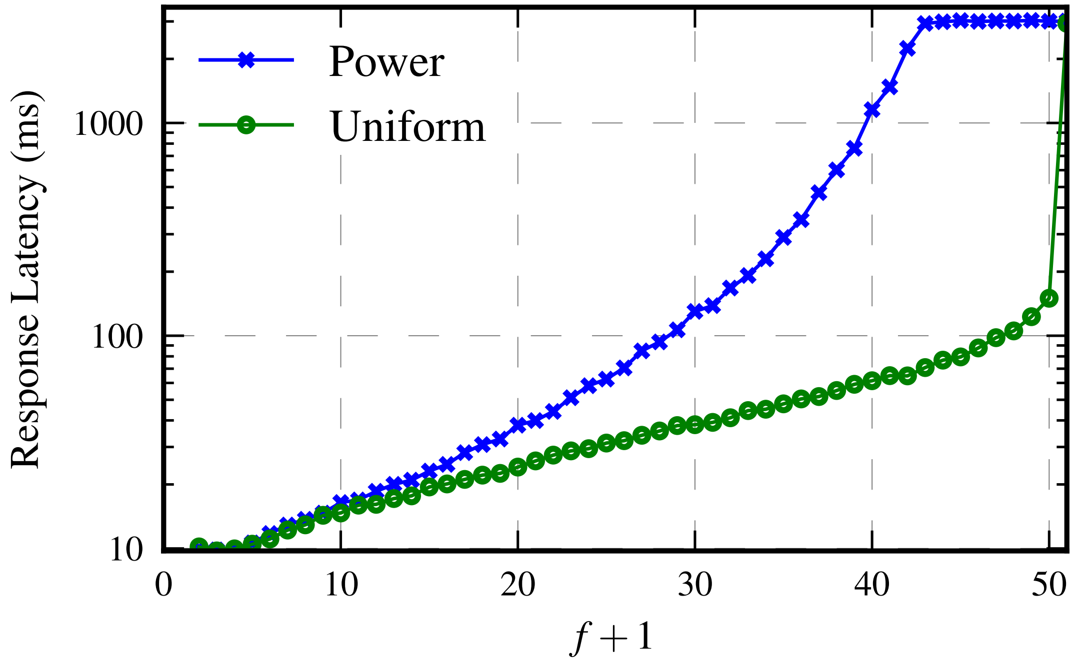


Figure 4.8: Performance implications of varying the minimum set of trusted writers for collaborative verification.

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 probability of 0.01 and a read otherwise. Read responses measured approximately 32 KB in size, mostly due to the signatures they contained.

Figure 4.7 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 3000 requests per second, while median latency remained below 0.5 ms.

4.7.4 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.

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 (0.02) 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 [65]). A client then issued a read using increasing values of f. Read latencies plotted in Figure 4.8 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.

4.7.5 Latency of ACL Modifications

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

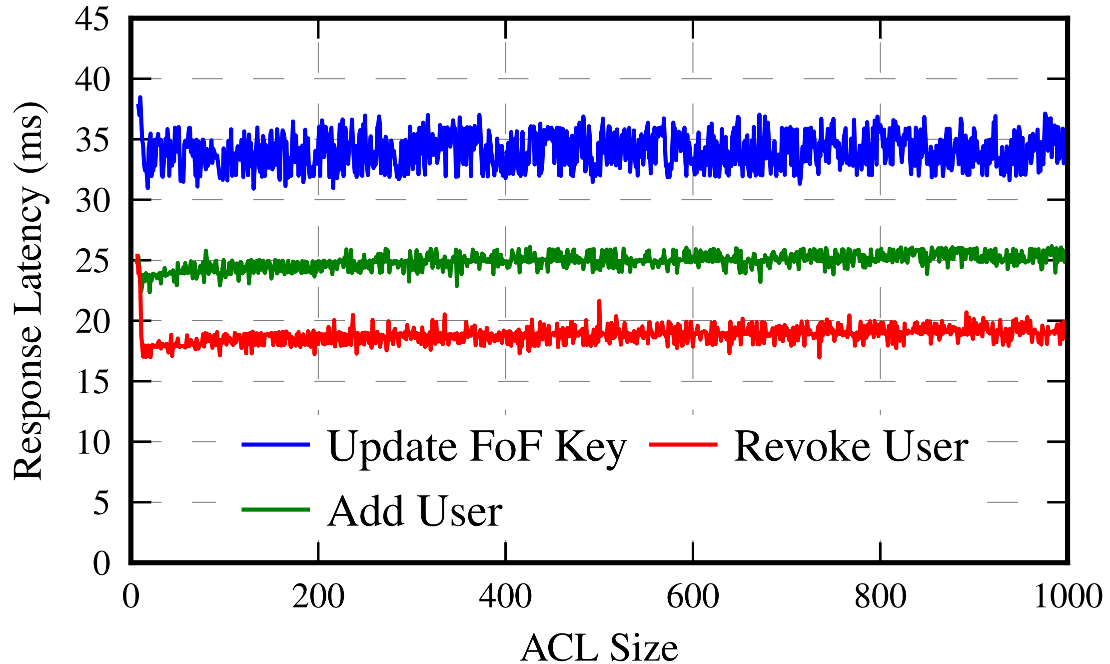


Figure 4.9: Latency of various ACL operations as a function of number of friends. Friend of Friend updates are measured as time to change a single user's FoF key. Each data point is the median of 100 runs.

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 4.9.

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.

4.8 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 [32], 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 [28], eXO [70], PeerSoN [15], and porkut [84], 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 [99] and Vis-à-Vis [100], 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 [9], flyByNight [71], NOYB [51], and Contrail [108] store users' data with untrusted providers but protect its contents with encryption. Others, such as Hummingbird [23], Lockr [116], and systems from Backes *et al.* [7], Domingo-Ferrer *et al.*[33] and Carminati *et al.* [18] 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 traffic analysis other than decentralization. Unlike Frientegrity, in many of these systems (*e.g.*, [9, 71, 23]), "un-friending" requires work that is linear in the number of a user's friends. The scheme of Sun *et al.* [114] is an exception, but it does not support FoFs. 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.*, [71, 23]), 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 [67, 68] and key-value stores [16, 101, 72] 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 [94] 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.

4.9 Conclusion

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.

Chapter 5

Conclusion

Currently, adopting cloud deployment requires users to cede control of their data to cloud providers and to rely on providers' promises that they will handle users' data properly. In this dissertation we have shown, however, that many user-facing cloud services can be redesigned so that users can benefit from cloud deployment without having to trust providers with the confidentiality and integrity of their data. We have presented two systems, SPORC and Frientegrity, that make it possible to build practical software as a service applications that maintain their privacy and security guarantees even in the face of an actively malicious provider.

To accomplish this goal, our systems employ a two-pronged strategy. First, to protect the confidentiality of users' information, both systems ensure that providers' servers only observe encrypted data. In so doing, not only do they stop the provider from misusing or disclosing users' data itself, they also prevent users' data from being stolen by an attacker who compromises the providers' servers. Second, to protect the integrity of users' data, both systems' client-server protocols provide clients enough information to verify that the provider is performing its role faithfully. As a result, they enable clients to quickly detect any deviation from correct execution, ranging

from simple tampering with users' individual updates to complex equivocation about the system state.

Our first system, SPORC, is a generic framework for building a wide variety of applications from collaborative word processors to group calendars to email and instant messaging systems 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. Beyond demonstrating that practical collaborative applications are possible in our threat model, SPORC makes two other conceptual contributions. First, it illustrates the complementary benefits of operational transformation (OT) and fork* consistency, two concepts from disparate research communities. It shows that the same mechanism that provides lock-free concurrency and automatic conflict resolution under normal operation can also enable clients to repair any inconsistencies that provider equivocation may have caused.

Second, SPORC demonstrates that many practical cloud applications can be cleanly separated into application-specific client software and generic servers that are oblivious to the contents of users' data and even to the applications that users are running. In so doing, it shows that all these applications really need from a cloud provider is a highly available, low-latency ordering and storage service (or even a separate service for each of these tasks). Thus, it potentially offers an alternative architecture in which applications run on a user's client, but each application is augmented by a generic cloud ordering and storage service of the users' choice. Users could treat such services as interchangeable, metered utilities. But regardless of which provider they chose, they would not have to trust it for confidentiality or integrity.

Frientegrity, our second system, shows that our vision of untrusted cloud deployment is applicable beyond group collaboration by extending it to online social networking. SPORC and other prior systems that are robust to equivocation do not scale to the needs of social networking applications in which users may have hun-

dreds of friends, and in which users are mainly interested the latest updates, not in the thousands that may have come before. But Frientegrity offers a novel method of enforcing fork* consistency in which clients collaborate so that each client only needs to do a small portion of the work required to verify correctness. Not only is this consistency mechanism more efficient than SPORC's, it is also robust to collusion between a misbehaving provider and as many as f malicious users. Furthermore, Frientegrity introduces an access control mechanism that offers efficient revocation and scales logarithmically with the number of friends.

5.1 Future Work

SPORC and Frientegrity demonstrate that practical cloud services with untrusted providers are possible. Yet due to the diversity of cloud applications, their use cases, and their threat models, our work can only be considered a small part of a comprehensive mitigation of the risks of cloud deployment. Much work remains, and we discuss three directions for future work here.

Other applications of untrusted cloud deployment: Frientegrity shows that, even within our threat model of a potentially Byzantine faulty 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. We believe that there are likely other classes of software as a service applications that could be implemented with untrusted providers. A framework to support a new class of applications would share our two-pronged strategy: only showing the provider encrypted data and enabling clients to verify that the provider is behaving correctly. But the algorithms and data structures that the framework would use might differ from those that we have described. Attempting to support new applications with untrusted providers is a worthwhile goal. If the attempt succeeds, not only will

it benefit users, it may also provide insights into general techniques for developing efficient systems with untrusted parties. If it fails, it may at least shed light on the classes of applications that cannot be supported in our threat model.

Supporting additional features: SPORC and Frientegrity support a variety of practical group collaboration and social networking applications. Yet because they prevent server code from manipulating plaintext, there are still certain common features that they cannot support in a straightforward manner such as cross-document or cross-object search, automatic language translation, and delivery of advertising. Finding practical ways to at least partially support these features would go a long way in spurring the adoption of systems like SPORC and Frientegrity.

One approach to supporting these features is to delegate these tasks to a trusted party or to a subset of clients themselves. For example, in both of our systems, one could imagine "robot" clients who are authorized to decrypt other clients' updates and whose sole purpose is construct search indices or perform language translation. Similarly, privacy-preserving advertising systems such as Adnostic [117] and Privad [50] runs the ad selection algorithm on clients rather than ad networks' servers so that users' do not have to share their browsing habits with marketers.

Another approach is to perform these functions on untrusted servers, but to use algorithms that operate on encrypted data. Fully homomorphic encryption, which allows arbitrary functions to be performed on encrypted data, remains orders of magnitude too inefficient for practical use [41]. But efficient algorithms for performing limited classes of operations on encrypted data, such as keyword search [104, 14, 10], are available. Preventing the provider from learning the contents of the data upon which it is operating is not enough, however. Because we assume that the provider may be actively malicious, clients must be given a means to verify the result of any operation that the provider performs on users' data.

Alternative threat and deployment models: Both of SPORC and Frientegrity assume a strong adversary: a provider that may be actively malicious. They also assume weak clients that cannot be counted on to remember state between sessions and that may be online very infrequently. We made these assumptions because we believe that they reflect the real-world balance of power between cloud providers and their users. Yet, our threat model and deployment assumptions represent one end of the design spectrum. For certain use cases, it may be possible to relax some of our assumptions to produce systems that are more efficient and can delegate more work to the provider. For example, assuming the existence of a trusted third party or trusted hardware [20, 66] could yield a more efficient system by reducing the amount of verification clients needs to perform. In addition, assuming that clients are regularly online could enable consensus protocols between clients that allow clients to discard old state that they have all seen or identify and exclude misbehaving clients. Finally and not surprisingly, if one is willing to assume an "honest but curious" provider, then many more tasks can be left to the provider. In these cases, the provider can play a more traditional role in the system, such as running a database server [89].

5.2 Final Remarks

Cloud computing has become increasingly popular despite its risks and despite the apparent inability of regulation or market incentives to sufficiently mitigate them. This outcome might have led one to conclude that users' loss of control over their data is an unavoidable consequence of this new model of organizing computing resources. This dissertation has shown, however, that this result need not be inevitable. It demonstrates that if we are willing to redesign cloud services, we can allow users to obtain most of the benefits of cloud deployment while effectively returning control users' data to the users themselves. Although the solutions we have presented here

are incomplete, they provide reason for optimism that many of cloud computing's risks can be mitigated by technical means.

Bibliography

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2005.

[2] Ben Adida. Helios: Web-based open-audit voting. In *Proceedings of the USENIX Security Conference*, August 2008.

[3] Yair Amir, Cristina Nita-Rotaru, Jonathan Stanton, and Gene Tsudik. Secure spread: An integrated architecture for secure group communication. *IEEE Transactions on Dependable and Secure Computing*, 2:248–261, 2005.

[4] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems (TOCS)*, 14(1), 1996.

[5] Cecilia R. Aragon and Raimund G. Seidel. Randomized search trees. In *Proceedings of the Symposium on the Foundations of Computer Science (FOCS)*, October 1989.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Dept. of Electrical Engineering and Computer Sciences, University of California at Berkeley, February 2009.

[7] Michael Backes, Matteo Maffei, and Kim Pecina. A security API for distributed social networks. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, February 2011.

[8] Lars Backstrom, Cynthia Dwork, and Jon Kleinberg. Wherefore Art Thou R3579X? Anonymized social networks, hidden patterns, and structural steganography. In *Proceedings of the International World Wide Web Conference (WWW)*, May 2007.

[9] Randy Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee, and Daniel Starin. Persona: an online social network with user-defined privacy. In *Proceedings of the Conference of the Special Interest Group on Data Communication (SIGCOMM)*, August 2009.

[10] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. *Advances in Cryptology – CRYPTO*, pages 535–552, August 2007.

[11] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[12] Rainier Böhme. A comparison of market approaches to software vulnerability disclosure. In *Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS)*, March 2006.

[13] Dan Boneh, Craig Gentry, and Brent Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. *Advances in Cryptology – CRYPTO*, Volume 3621/2005:258–275, August 2005.

[14] Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. In *Proceedings of the Theory of Cryptography Conference (TCC)*, March 2006.

[15] Sonja Buchegger, Doris Schiöberg, Le hung Vu, and Anwitaman Datta. PeerSoN: P2P social networking early experiences and insights. In *Proceedings of the ACM Workshop on Social Network Systems (SNS)*, March 2009.

[16] Christian Cachin, Idit Keidar, and Alexander Shraer. Fail-aware untrusted storage. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2009.

[17] Christian Cachin, Abhi Shelat, and Alexander Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proceedings of the Symposium on the Principles of Distributed Computing (PODC)*, August 2007.

[18] Barbara Carminati and Elena Ferrari. Privacy-aware collaborative access control in web-based social networks. In *Proceedings of the IFIP WG 11.3 Conference on Data and Applications Security (DBSec)*, July 2008.

[19] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[20] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2007.

[21] 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`, March 2011.

[22] U.S. Federal Trade Commission. Protecting consumer privacy in an era of rapid change: Recommendations for businesses and policymakers. http://www.ftc.gov/os/2012/03/120326privacyreport.pdf, March 2012.

[23] Emiliano De Cristofaro, Claudio Soriente, Gene Tsudik, and Andrew Williams. Hummingbird: Privacy at the time of twitter. Cryptology ePrint Archive, Report 2011/640, 2011. http://eprint.iacr.org/.

[24] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the USENIX Security Conference*, August 2009.

[25] Scott A. Crosby and Dan S. Wallach. Super-efficient aggregating history-independent persistent authenticated dictionaries. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, September 2009.

[26] Scott A. Crosby and Dan S. Wallach. High throughput asynchronous algorithms for message authentication. Technical Report CS TR10-15, Rice University, December 2010.

[27] Scott A. Crosby and Dan S. Wallach. Reference implementation of history trees and spliced signatures. https://github.com/scrosby/fastsig, December 2010.

[28] Leucio Antonio Cutillo, Refik Molva, Thorsten Strufe, and Tu Darmstadt. Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine*, 47(12):94–101, December 2009.

[29] John Day-Richter. What's different about the new google docs: Conflict resolution. http://googledocs.blogspot.com/2010/09/whats-different-about-new-google-docs_22.html, September 2010.

[30] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2007.

[31] Shardul 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.

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

[33] Josep Domingo-Ferrer, Alexandre Viejo, Francesc Sebé, and írsula González-Nicolás. Privacy homomorphisms for social networks with private relationships. *Computer Networks*, 52:3007–3016, October 2008.

[34] John R. Douceur. The Sybil attack. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

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

[36] Facebook, Inc. Facebook. `http://www.facebook.com/`. Retrieved April 23, 2012.

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

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

[39] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

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

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

[42] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2003.

[43] Oded Goldreich. The semi-honest model. In *Foundations of Cryptography*, volume 2, chapter 7.2.2, pages 619–626. Cambridge University Press, 2004.

[44] Google, Inc. EtherPad. `http://etherpad.com/`. Retrieved April 23, 2012.

[45] Google, Inc. Google Apps. `http://www.google.com/apps/index1.html`. Retrieved April 23, 2012.

[46] Google, Inc. Google Docs. `http://docs.google.com/`. Retrieved April 23, 2012.

[47] Google, Inc. Google Wave federation protocol. `http://www.waveprotocol.org/federation`. Retrieved April 23, 2012.

[48] Google, Inc. Google Web Toolkit (GWT). `https://developers.google.com/web-toolkit/`. Retrieved April 23, 2012.

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

[50] Saikat Guha, Bin Cheng, and Paul Francis. Privad: Practical privacy in online advertising. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, March 2011.

[51] Saikat Guha, Kevin Tang, and Paul Francis. NOYB: Privacy in online social networks. In *Proceedings of the Workshop on Online Social Networks (WOSN)*, August 2008.

[52] Mark Handley and Jon Crowcroft. Network text editor (NTE): A scalable shared text editor for MBone. In *Proceedings of the Conference of the Special Interest Group on Data Communication (SIGCOMM)*, October 1997.

[53] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990.

[54] Ian Hickson. Web storage - W3C candidate recommendation. http://www.w3.org/TR/webstorage/, December 2011.

[55] IBM. IBM cloud computing: Overview. http://www.ibm.com/cloud-computing/us/en/. Retrieved April 23, 2012.

[56] SpiderOak Inc. SpiderOak: Nuts and bolts. https://spideroak.com/engineering_matters. Retrieved April 23, 2012.

[57] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Symposium on the Theory of Computing (STOC)*, May 1997.

[58] Alain Karsenty and Michel Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, May 1993.

[59] Jason Kincaid. Google privacy blunder shares your docs without permission. *TechCrunch*, March 2009.

[60] Ramakrishna Kotla and Mike Dahlin. High-throughput Byzantine fault tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2004.

[61] David Kravets. Aging 'privacy' law leaves cloud e-mail open to cops. *Wired Threat Level Blog*, October 2011.

[62] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[63] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[64] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3), 1982.

[65] Jure Leskovec and Eric Horvitz. Planetary-scale views on a large instant-messaging network. In *Proceedings of the International World Wide Web Conference (WWW)*, April 2008.

[66] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, April 2009.

[67] Jinyuan Li, Maxwell N. Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[68] Jinyuan Li and David Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, April 2007.

[69] Amazon Web Services LLC. Amazon elastic compute cloud (Amazon EC2). `http://aws.amazon.com/ec2/`. Retrieved April 23, 2012.

[70] Andreas Loupasakis, Nikos Ntarmos, and Peter Triantafillou. eXO: Decentralized autonomous scalable social networking. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, January 2011.

[71] Matthew M. Lucas and Nikita Borisov. flyByNight: mitigating the privacy risks of social networking. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES)*, October 2008.

[72] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[73] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the Symposium on the Theory of Computing (STOC)*, May 1997.

[74] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the Symposium on the Principles of Distributed Computing (PODC)*, July 2002.

[75] Peter Mell and Timothy Grance. NIST special publication 800-145: The NIST definition of cloud computing. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf, September 2011.

[76] John P. Mello. Facebook scrambles to fix security hole exposing private pictures. *PC World*, December 2011.

[77] Ralph C. Merkle. A digital signature based on a conventional encryption function. *Advances in Cryptology – CRYPTO*, pages 369–378, 1987.

[78] Microsoft. Microsoft office live. http://www.officelive.com/. Retrieved April 23, 2012.

[79] Microsoft. Windows azure virtual machine role. http://www.windowsazure.com/en-us/home/features/virtual-machines/. Retrieved April 23, 2012.

[80] Elinor Mills. Hackers release credit card, other data from stratfor breach. *CNET News*, December 2011.

[81] Mozilla Project. LiveConnect. https://developer.mozilla.org/en/LiveConnect. Retrieved April 23, 2012.

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

[83] Arvind Narayanan and Vitaly Shmatikov. De-anonymizing social networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2009.

[84] Rammohan Narendula, Thanasis G. Papaioannou, and Karl Aberer. Privacy-aware and highly-available OSN profiles. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE)*, June 2010.

[85] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proceedings of the Symposium on User Interface Software and Technology (UIST)*, November 1995.

[86] Alina Oprea and Michael K. Reiter. On consistency of encrypted files. In *Proceedings of the Symposium on Distributed Computing (DISC)*, September 2006.

[87] Kurt Opsahl. Facebook's eroding privacy policy: A timeline. *EFF Deeplinks Blog*, April 2010.

[88] European Parliament and European Council. Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:1995:281:0031:0050:EN:PDF, October 1995.

[89] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2011.

[90] Atul Prakash and Michael J. Knister. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction*, 4(1):295–330, December 1994.

[91] Rackspace US, Inc. Rackspace. `http://www.rackspace.com/`. Retrieved April 23, 2012.

[92] Marguerite Reardon. India threatens to shut down blackberry service. *CNET News*, August 2010.

[93] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*, November 1996.

[94] Daniel R. Sandler and Dan S. Wallach. Birds of a FETHR: Open, decentralized micropublishing. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, April 2009.

[95] Ruchi Sanghvi. Facebook blog: New tools to control your experience. `https://blog.facebook.com/blog.php?post=196629387130`, December 2009.

[96] André Schiper, Kenneth Birman, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, August 1991.

[97] Erick Schonfeld. Watch out who you reply to on google buzz, you might be exposing their email address. *TechCrunch*, February 2010.

[98] Jason Schreier. Playstation network hack leaves credit card info at risk. *Wired Game—Life Blog*, April 2011.

[99] Seok-Won Seong, Jiwon Seo, Matthew Nasielski, Debangsu Sengupta, Sudheendra Hangal, Seng Keat Teh, Ruven Chu, Ben Dodson, and Monica S. Lam. PrPl: A decentralized social networking infrastructure. In *Proceedings of the ACM Workshop on Mobile Cloud Computing & Services (MCS)*, June 2010.

[100] Amre Shakimov, Harold Lim, Ramón Caceres, Landon P. Cox, Kevin Li, Dongtao Liu, and Alexander Varshavsky. Vis-à-Vis: Privacy-preserving online social networking via virtual individual servers. In *Proceedings of the International Conference on Communication Systems and Networks (COMSNETS)*, January 2011.

[101] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: Verification for untrusted cloud storage. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, October 2010.

[102] Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, and Petros Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, April 2009.

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

[104] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2000.

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

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

[107] Sven Strohschein. GWTEventService. `https://code.google.com/p/gwteventservice/`. Retrieved April 23, 2012.

[108] Patrick Stuedi, Iqbal Mohomed, Mahesh Balakrishnan, Zhuoqing Morley Mao, Venugopalan Ramasubramanian, Doug Terry, and Ted Wobber. Contrail: Enabling decentralized social networks on smartphones. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, December 2011.

[109] Maher Suleiman, Michèle Cart, and Jean Ferrié. Serialization of concurrent operations in distributed collaborative environment. In *Proceedings of the International Conference on Supporting Group Work (GROUP)*, November 1997.

[110] Chengzheng Sun and Clarence (Skip) Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*, November 1998.

[111] Chengzheng Sun, Xiahua Jia, Yanchun Zhang, and Yun Yang. A generic operation transformation schema for consistency maintenance in real-time cooperative editing systems. In *Proceedings of the International Conference on Supporting Group Work (GROUP)*, November 1997.

[112] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):64–108, 1998.

[113] David Sun, Steven Xia, Chengzheng Sun, and David Chen. Operational transformation for collaborative word processing. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*, November 2004.

[114] Jinyuan Sun, Xiaoyan Zhu, and Yuguang Fang. A privacy-preserving scheme for online social networks with efficient revocation. In *Proceedings of the International Conference on Computer Communications (INFOCOM)*, March 2010.

[115] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, December 1995.

[116] Amin Tootoonchian, Stefan Saroiu, Yashar Ganjali, and Alec Wolman. Lockr: Better privacy for social networks. In *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, December 2009.

[117] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, February 2010.

[118] Twitter, Inc. Twitter. `http://www.twitter.com/`. Retrieved April 23, 2012.

[119] Jaikumar Vijayan. 36 state ags blast google's privacy policy change. *Computerworld*, February 2012.

[120] Tony Vila, Rachel Greenstadt, and David Molnar. Why we can't be bothered to read privacy policies: Privacy as a lemons market. In *Proceedings of the International Conference on Electronic Commerce (ICEC)*, October 2003.

[121] David Wang and Alex Mah. Google wave operational transformation. `http://www.waveprotocol.org/whitepapers/operational-transform`. Retrieved April 23, 2012.

[122] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, April 2009.

[123] David Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`. Retrieved April 23, 2012.

[124] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30, 1998.

[125] Thomas D. Wu. The secure remote password protocol. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, March 1998.

[126] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2003.

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