Privacy and Integrity in the Untrusted Cloud

Edward W. Felten
felten@cs.princeton.edu

Joint work with:
Ariel J. Feldman, William P. Zeller,
Aaron Blankstein, Michael J. Freedman
Cloud deployment: Pro & Con

For user-facing apps:

**Pro:** Availability, reliability, global accessibility, convenience

**Con:** Users give up control over their data

Must trust provider for confidentiality & integrity
Threats to confidentiality

- Theft by attackers
- Accidental leaks
- Privacy policy changes
- Government pressure
Threats to integrity

Simple: Corrupting messages

Complex: Server equivocation

Does this happen? Yes!

(e.g. to disguise censorship)

http://songshinan.blog.caixin.com/archives/22322 (translated by Google)
Legal or market-based solution?

We’re skeptical…

Users’ limited information

• May not know what third party is doing (i.e. security is a “lemons market”)
• May not find out until it’s too late
• Third party could change its behavior over time

Not enough to wait until damage is done

• Harm could be irreparable
• Quantifying harm is often hard
Our approach

Privacy & integrity by design:

• Benefit from cloud deployment
• Assume untrusted provider

Contributions:

• Practical cloud apps
• Preventing confidentiality violations
• Detecting and recovering from misbehavior
Outline

1. Introduction

2. **SPORC:**
   Cloud-based group collaboration [FZFF10]

3. **Frientegrity:**
   Privacy & integrity for online social networks [FBFF12]

4. Conclusion
SPORC goals

Collaborative editing of shared state

- Flexible framework
- Real-time
- Work offline

Untrusted servers

- Can’t read user data
- Can’t tamper with user data without risking detection
- Clients can recover from tampering
Making servers untrusted

SPORC: Server’s limited role:
- Storage
- Ordering msgs

Client 1
- App logic
- Copy of state

Client 2
- App logic
- Copy of state

Server
- App logic
- State

Privacy and Integrity in the Untrusted Cloud / Edward W. Felten / TRUST WISE Berkeley / June 2012
Problem #1: How do you keep clients’ local copies consistent?

(esp. with offline access)
Problem #2: How do you deal with a malicious server?

Client 1

Client 2

Server

Encrypted state

How do you deal with a malicious server?
Keeping clients in sync

Operational transformation (OT) [EG89]
(Used in Google Docs, EtherPad, etc.)

OT can sync arbitrarily divergent clients
Dealing with a malicious server

Digital signatures aren’t enough

Server can equivocate

fork* consistency [LM07]
• Honest server: linearizability
• Malicious server: Alice and Bob detect equivocation after exchanging 2 messages
• Embed history hash in every message

Server can still fork the clients, but can’t unfork
System design

Client app

Local state

SPORC lib
System design

Client app

Local state

Committed Pending

SPORC lib

fork* consistent

causally consistent
System design

Client app

Local state

SPORC lib

Committed Pending

Encrypt & sign

Server

Encrypted state

Encrypt
& sign
System design

Client app

Local state

SPORC lib

Committed Pending

Compare history hashes

Verify & decrypt

Server

Encrypted state

Client
System design

Client app

Local state

Committed
Pending

SPORC lib

Verify & decrypt

T

Server

Encrypted state

Client
System design

Client app

Local state

Committed

Pending

SPORC lib

Server

Encrypted state

Client

T
Access control

Challenges
• Server can’t enforce — it’s untrusted!
• Preserving causality
• Concurrency makes it harder

Solutions
• Ops encrypted with symmetric key shared by clients
• ACL changes are ops too
• Concurrent ACL changes handled with barriers
Adding a user

Group members:

- Alice
- Bob
- Charlie

ModifyUserOp

Add “Charlie”

$E_{Charlie\_pk}(k)$
Removing a user

Server

ModifyUserOp

Rm “Charlie”

E_{alice_{pk}}(k')

E_{bob_{pk}}(k')

E_{k'}(k)

Group members:

Alice

Bob

Charlie

E_{alice_{pk}}(k') = E_{bob_{pk}}(k') = E_{k'}(k)
Barriers: dealing with concurrency

Clients check on the server

Group members:
- Alice
- Bob
- Charlie
- Eve

ModifyUserOp
Rm “Charlie”
$E_{k1}(k)$

ModifyUserOp
Rm “Eve”
$E_{k2}(k1)$
Recovering from a fork

Can use OT to resolve malicious forks too
Implementation

Client lib + generic server

App devs only need to define ops and provide a transformation function

Java CLI version + browser-based version (GWT)

Demo apps: key value store, browser-based collaborative text editor
Evaluation

Setup

- Tested Java CLI version
- 8-core 2.3 GHz AMD machines
  - 1 for server
  - 4 for clients (often >1 instance per machine)
- Gigabit LAN

Microbenchmarks

- Latency
- Server throughput
- Time-to-join (in paper)
Latency

Low load
(1 client writer)

High load
(all clients are writers)
Server throughput

![Graph showing server throughput](image)

- Throughput (MB/s) vs. Payload size (KB)
- Red line with error bars: MB/s
- Blue dashed line: MB/s
- X-axis: Payload size (KB)
- Y-axis: Throughput (MB/s)
- Data points and error bars indicate variability in throughput.
Summary

Practical cloud apps + untrusted servers

Dynamic access control and key distribution prevents confidentiality violations

OT + fork* consistency enables detection of and recovery from misbehavior
Outline

1. Introduction

2. SPORC: Cloud-based group collaboration [FZFF10]

3. Frientegrity: Privacy & integrity for online social networks [FBFF12]

4. Conclusion
Social network privacy & integrity

Particularly problematic:
Switching is difficult, provider tempted to repurpose data

Prior work:
1. Cryptographic (e.g. Persona, flyByNight, NOYB, Lockr, [BMP11])
   Don’t protect integrity

2. Decentralized (e.g. Diaspora, Safebook, eXO, PeerSON, PrPl)
   Run your own server (sacrifice availability, convenience, etc.)
   OR
   Trust a provider (who you probably don’t know)
Q: Why not SPORC?

A: Scalability

<table>
<thead>
<tr>
<th>SPORC provides</th>
<th>Social networks need</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each document is independent (Has its own ACL)</td>
<td>Multiple related objects (e.g. on a user's profile) (Under a single friend list)</td>
</tr>
<tr>
<td>Enforcing fork* consistency is $O(n)$ (Downloads entire document)</td>
<td>Objects are large (e.g. Facebook wall)</td>
</tr>
<tr>
<td>• ACL changes rare</td>
<td>• Enforcing correctness must be fast</td>
</tr>
<tr>
<td>• Revoking access is $O(n)$</td>
<td>• Only want latest changes</td>
</tr>
<tr>
<td>Few participants</td>
<td>Many friends</td>
</tr>
<tr>
<td>• “Friending” &amp; “unfriending” common</td>
<td>• “Friending” &amp; “unfriending” common</td>
</tr>
<tr>
<td></td>
<td>• Revoking access must be fast</td>
</tr>
</tbody>
</table>
## Frientegrity

<table>
<thead>
<tr>
<th>Social networks need</th>
<th>Frientegrity provides</th>
</tr>
</thead>
</table>
| **Multiple related objects** (e.g. on a user’s profile)  
(Under a single friend list) | **Multiple related objects**  
• Spread across servers  
• Share an ACL |
| **Objects are large** (e.g. Facebook wall)  
• Enforcing correctness must be fast  
• Only want latest changes | **Clients enforce fork* consistency collaboratively**  
Each client only downloads & verifies a small part of an object |
| **Many friends**  
• “Friending” & “unfriending” common  
• Revoking access must be fast | **ACL operations** $O(\log n)$ |
Frientegrity overview

Only returns latest ops
Yet, provides enough info to:
- Verify fork* consistency
- Obtain decryption keys

Server 1
- Alice’s profile
  - Alice’s photo album
  - Alice’s ACL
  - Comment thread
  - Alice’s wall

Server 2
- Bob’s profile
- Charlie’s profile

Server n

Client
- read(‘Alice’)
- Verify & decrypt

Fork* consistency
Enforcing fork* consistency in SPORC

SPORC’s hash chains are $O(n)$
(Also, must download entire history)

Prior systems were linear in history size or number of clients
(e.g. SUNDR, Depot)
Objects in Frientegrity

h_{root} commits to entire history

Let C_{15} be a server-signed commitment to h_{root} up to op_{15}

h_i = H(h_{leftChild(i)} || h_{rightChild(i)})

History tree [CW09]
Objects (cont.)

Is $C_8$ consistent with $C_{15}$?
Verifying an object

Clients collaborate to verify the history

Is $C_{11}$ consistent with $C_{15}$?
Tolerating malicious users

Tolerate up to $f$ malicious users

Alice’s ops
Bob’s ops
Charlie’s ops

op_0  op_1  op_2  op_3  op_4  op_5  op_6  op_7  op_8  op_9  op_10  op_11  op_12  op_13  op_14  op_15

C_1

C_9
Scalable access control

SPORC membership ops are expensive

Instead, use a key graph [WGL98]
Adding a friend

ModifyUserOp
Add “Sarah”

\[
\text{ModifyUserOp} \rightarrow k_0 \rightarrow k_1 \rightarrow k_3 \rightarrow k_7 \rightarrow k_14 \rightarrow E_{k_{14}}(k_6) || E_{k_{13}}(k_6) \rightarrow E_{sarah\_pk}(k_{14})
\]
Removing a friend

ModifyUserOp
Rm "David"

\[ k_0' = k_{alice\_friend}' \]
Read & write latency

Frientegrity
(collaborative verification)

SPORC
(hash chain)

Constant cost of signatures dominates
Latency of ACL changes

![Graph showing latency of ACL changes with response latency (ms) on the y-axis and ACL size on the x-axis. Two lines are shown: blue for Add User and green for Revoke User.](image-url)
Effect of increasing $f$

- 50 writers
- 5000 operations

![Graph showing response latency vs. $f + 1$ for Power and Uniform distributions.](image)
Summary

Online social networking + untrusted provider

Clients collaborate to defend against equivocation
(i.e. to enforce fork* consistency)

Tolerates up to f malicious users
(SPORC assumed trusted clients)

Scalable access control: key distribution & revocation are $O(\log n)$
Outline

1. Introduction

2. SPORC: Cloud-based group collaboration [FZFF10]

3. Frientegrity: Privacy & integrity for online social networks [FBFF12]

4. Conclusion
Conclusion

Practical apps + untrusted provider are possible

- Assume actively malicious (Byzantine faulty) provider
- Privacy & integrity guaranteed by users’ keys

Contributions:

- Frameworks for group collaboration & online social networking
- Detect and recover from equivocation
- Dynamic access control & key distribution that supports concurrency
- Protocols that scale to needs of real-time collaboration & large online social networks
Thank you
Questions?

felten@cs.princeton.edu

References
