Parakeet: Practical Key Transparency for End-to-End Encrypted Messaging

Harjasleen Malvai
(UIUC, IC3)

joint work with
Lefteris Kokoris-Kogias (IST Austria, Mysten Labs), Alberto Sonnino (Mysten Labs), Esha Ghosh (MSR), Ercan Ozturk, Kevin Lewi, and Sean Lawlor (Meta)
End-to-End Encrypted Messaging

Alice
1800-ali-ce

encrypted Messaging Service

+ Identity Provider

Bob
100-bob-ph

Want to send encrypted messages

Need Each Other’s Public Keys

<table>
<thead>
<tr>
<th>Username</th>
<th>Public Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>1800-ali-ce</td>
<td>PK_{alice}</td>
</tr>
<tr>
<td>100-bob-ph</td>
<td>PK_{bob}</td>
</tr>
</tbody>
</table>
End-to-End Encrypted Messaging

Alice

Bob

Encrypted Messaging Service + Identity Provider

Server mounts meddler-in-the-middle

Undetected!
Identity provider holds a dictionary with username-PK pairs, such that

- **New Users**: New users may join.
- **Changing state**: Users “own” their usernames and can update PKs.
- **Privacy**: Users want the server to restrict queries to their usernames.

**Identity provider could cheat**: Same username queried at the same time → diverging value.

<table>
<thead>
<tr>
<th>Uname</th>
<th>PK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>PK(_{\text{alice}})</td>
</tr>
<tr>
<td>Bob</td>
<td>PK(_{\text{bob}})</td>
</tr>
</tbody>
</table>
More General

- Privacy-preserving centralized financial ledger:
  - Central server trusted for privacy + ordering.✅
  - Not trusted for serving correct state, e.g. FTX.✅

- Certificate transparency, tamper evident logging.
  - No privacy requirement, ❌
  - Larger computation at clients. ❌
  - Usually only for “logging” not mapping. ❌
Key Transparency Security Guarantee

● Threat model:
  ○ IdP holds dictionary,
  ○ May “cheat”: show diverging views to different parties.
● Ideal: want to **prevent** the IdP from cheating at all.
● Assumptions on clients:
  ○ Need them to store secrets, etc.
● Security = **non-equivocation**:
  ○ Cannot show different keys to different clients w/o getting caught.
  ○ At any given time:
    ■ Alice thinks her key is $PK_{Alice}$ ⇒ the server cannot get away with telling Bob her key is $PK_{BAD}$. 
Model: Parties

Identity Provider

- Trusted for privacy and authentication.
- **Want to eliminate trust for serving correct public keys.**
- Updates are batched and take effect in discrete time steps called *epochs*.

Users

- May update their public key.
- Lookup each other’s public keys if permitted e.g. not blocked.
- Check their own keys’ history up till the present epoch.
- **Want no changes to their keys without their finding out.**
- Their friends should receive matching keys for them.

Auditors

- Share some computational burden.
- **Check global predicates.**
- Audit(start_epoch, end_epoch): Check that the server’s state changes between these epochs are valid. E.g., the server doesn’t destroy records.
- Could be users, smart contracts, designated machines, unrelated third-parties, etc.
- **Should not learn data about particular users!**
Key Transparency Components + Desiderata

Components

- Mechanism for server committing to mutating state.
- Mechanism to allow users to monitor their own keys.
- Need some ground truth, i.e. way to share a small commitment.

Also would like to be able to support

- Billions of users.
- Users with computationally limited devices.
Model

Need a way to commit to this: 
com

Serving commitments: Need a way for users to access com

<table>
<thead>
<tr>
<th>Username</th>
<th>Public Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>User1</td>
<td>PK_{user1}</td>
</tr>
<tr>
<td>User2</td>
<td>PK_{user2}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Problem Breakdown

Key Transparency

Commitment & Verification for Mutating DB

Disseminating Small Commitments
Committing to Server State

Key Transparency

Commitment & Verification for Mutating DB

Disseminating Small Commitments
Committing to Server State

Committing to this: \( \text{com} \)_{\text{state}}

<table>
<thead>
<tr>
<th>Username</th>
<th>Public Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>User1</td>
<td>( \text{PK}_{\text{user1}} )</td>
</tr>
<tr>
<td>User2</td>
<td>( \text{PK}_{\text{user2}} )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Identity Provider

Identity Provider’s State
Committing to Server State: Previous Work

- **Committing to state**: Solutions based on Merkle Trees or other kinds of vector commitments.
- **Reducing user load**: Reduce user computation using third-party auditors or SNARKs etc.
- **Users monitoring key history**:
  - Append-only authenticated data structures to prevent the server from cheating and then erasing evidence.
    - NO STATE CHANGES EVER DELETED.
  - OR users must do **linear work in server epochs** to monitor their own keys...
  - OR server must generate expensive proofs.
## Committing to State: This Work

<table>
<thead>
<tr>
<th>Problem</th>
<th>Previous Work</th>
<th>Pitfall</th>
<th>Our Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>State machine replication, storage layer separation.</td>
<td>AbSTRACTED away, experiments used single machines.</td>
<td>Entire large data structures cannot be read locally</td>
<td>Flexible storage layer API: modularly plugs in existing DBs.</td>
</tr>
<tr>
<td>Data structure for state commitments</td>
<td>Grow to \sim100s of TB over years</td>
<td>Storage cost blowup</td>
<td>Replace underlying data structure (oZKS)</td>
</tr>
</tbody>
</table>
Comparing Storage Costs

- Numbers for 10M updates a day
<table>
<thead>
<tr>
<th>Problem</th>
<th>Previous Work</th>
<th>Pitfall</th>
<th>Our Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>State machine replication, storage layer separation.</td>
<td>Abtracted away, experiments used single machines.</td>
<td>Entire large data structures cannot be read locally</td>
<td>Flexible storage layer API: modularly plugs in existing DBs.</td>
</tr>
<tr>
<td>Data structure for state commitments</td>
<td>Grow to ~100s of TB over years</td>
<td>Storage cost blowup</td>
<td>Replace underlying data structure (oZKS)</td>
</tr>
</tbody>
</table>
| Users checking own key | ● ZKP  
● Always online  
● Append-only data structures | ● Impractical for server  
● Impractical for client  
● Ever growing storage costs | Secure compaction: Find a middle ground between requiring users always online and totally append-only. |
**Committing to State: This Work**

<table>
<thead>
<tr>
<th>Problem</th>
<th>Previous Work</th>
<th>Pitfall</th>
<th>Our Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>State machine replication, storage layer separation</td>
<td>Abstracted away, experiments used single machines.</td>
<td>Entire large data structures cannot be read locally</td>
<td>Flexible storage layer API: modularly plugs in existing DBs.</td>
</tr>
<tr>
<td>Data structure for state commitments</td>
<td>Grow to ~100s of TB over years</td>
<td>Storage cost blowup</td>
<td>Replace underlying data structure (oZKS)</td>
</tr>
</tbody>
</table>
| Users checking own key | - ZKP  
- Always online  
- Append-only data structures | - Impractical for server  
- Impractical for client  
- Ever growing storage costs | Secure compaction: Find a middle ground between requiring users always online and totally append-only. |
Secure Compaction: Attempt 1

Delete *obsolete* entries here

- At any given epoch, auditors check that only old enough entries are deleted.
- **Problem:** Some old data may still be relevant.
Secure Compaction: Attempt 2

Delete *obsolete* entries here

Server Epochs

- At any given epoch, auditors check that only old enough entries are deleted.
- Show that for any deleted entry, there exists some update superseding it.
- **Problem:** Leaks metadata for users!

Bob's latest key: DO NOT DELETE!
Secure Compaction

- Involve both individual users and auditors.
- Well-defined (and sparse) epochs for deletion-related ops.
- Provide a grace period or checking period where values are tombstoned.
- Auditors check everything tombstoned is old enough.
- Users come online to check that only expired values for them are marked for deletion.
- Garbage collection after grace period is over.
Committing to Server State

Key Transparency

Commitment & Verification for Mutating DB

Disseminating Small Commitments
Serving Commitments

Key Transparency

Commitment & Verification for Mutating DB

Disseminating Small Commitments
Serving Commitments

Need a way to commit to this: 
\[ \text{com} \] 

Identity Provider

Identity Provider's State

<table>
<thead>
<tr>
<th>Username</th>
<th>Public Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>User1</td>
<td>PK_{user1}</td>
</tr>
<tr>
<td>User2</td>
<td>PK_{user2}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Serving commitments: Need a way for users to access \( \text{com} \)
Sharing a small commitment: Gossip?

- If users have an out-of-band communication mechanism, they could gossip the commitment they get, i.e. share their views.

- Problem for a global scale system because:
  - Might end up with partitions in the network of users, for example geographically.
  - Users may come online intermittently.
  - Users may not have the bandwidth (or enough battery) to gossip!
  - Dissemination might be too slow.
Sharing a small commitment: Blockchains?

- Could post the commitment in a smart contract or OP_RETURN on bitcoin.
- Must trust the blockchain and its code.
- Even “light” clients could be too heavyweight.
- If billions of users query, could end up flooding the network with queries!
Custom Consensus for Strong Consistency?

- Blockchain → Consensus.
- Is consensus really needed?
- Consensus pitfalls:
  - $N^2$ communication cost
  - Delays
  - Complex to implement and analyse
Consensus-less Strong Consistency

- Server is trusted for compiling the updates and finalizing the underlying database.
- Server also has infrastructure for compiling and serving messages from some sort of “independent witnesses”.
- Let’s use **witnesses**!
  - Multiple trusted hardware instances.
  - Industry consortiums.
- Witnesses store the latest commitment and check new commitment is ok.
Consensus-less Strong Consistency

Communication:
- IdP ↔ Witnesses
- IdP ↔ Users
- Witness ↔ Witness
- Users ↔ Witnesses
- Clients accept if a threshold number of witnesses sign.

1. Notify about new commitment
2. Check and sign
3. Collect signatures
**Consensus-less Strong Consistency**

- Consistency, validity and termination are guaranteed similar to BFT consensus protocols.
- No liveness but much faster!
- Can be used in addition to other mechanisms.
- Simple protocol, easier to implement and fewer bugs.
- Uses existing server infrastructure.
- Users get certified commitments together with proofs and query responses.

![Diagram of update protocol](image)

**Fig. 4:** Illustration of the key update protocol.
Consensus-less Strong Consistency Performance

![Graph showing latency vs. throughput for different numbers of witnesses and batch sizes.](image-url)
Thank You
Thank you!

- Thanks to my awesome collaborators!
- Find the paper on the NDSS website! (more details in full version: https://eprint.iacr.org/2023/081.pdf)
- **Open source implementations:** linked in the paper.
Appendix
(useful for several FAQs)
**SEEMless Data Structure: aZKS**

### Epoch 0

\[ h_0 = H(\epsilon) \]

### Epoch 1

- **h_{000} = H(v_{000})**
- **h_{110} = H(v_{110})**

- **\( h_\epsilon = H(h_{000}, 000, h_{110}, 110) \)**

After inserting label-value pairs \((000, v_{000})\) and \((110, v_{110})\).

### Epoch 2

- **h_{0} = H(h_{000}, 000, h_{011}, 011)**
- **h_{110} = H(v_{110})**

- **\( h_{110} = H(v_{110}) \)**

- **h_{000} = H(v_{000})**
- **h_{011} = H(v_{011})**

After inserting label-value pair \((011, v_{011})\).
Parakeet Data Structure: oZKS

Epoch 0

\[ \epsilon \quad h_\epsilon = H(\epsilon) \]

Epoch 1: After inserting label-value pairs \((000, v_{000})\) and \((110, v_{110})\).

\[ h_\epsilon = H(h_{000}, 000, h_{110}, 110) \]

\[ h_{000} = H(v_{000}, 1) \]

\[ h_{110} = H(v_{110}, 1) \]

Epoch 2: After inserting label-value pair \((011, v_{011})\).

\[ h_\epsilon = H(h_0, 0, h_{110}, 110) \]

\[ h_0 = H(h_{000}, 000, h_{011}, 011) \]

\[ h_{000} = H(v_{000}, 1) \]

\[ h_{011} = H(v_{011}, 2) \]

\[ h_{110} = H(v_{110}, 1) \]
Update Time Breakdown

![Graph showing update time breakdown]

- **Total Time to Insert 100K Users**
- **Database Write Time**
- **Database Read Time**

The graph illustrates the time (in minutes) taken to insert 100K users as the number of keys increases from 0 to 5.
Comparing Storage costs

![Graph comparing storage costs between SEEMless and Parakeet. The x-axis represents the number of epochs per day, and the y-axis represents storage in TB. SEEMless has a higher storage requirement than Parakeet.]

- SEEMless
- Parakeet
Storage Breakdown

![Storage Breakdown Graph]
Entries for a User

Fig. 3: An example of labels corresponding to user Alice's key updates. Since Alice has changed their key enough times since their initial entry into the system and epochs before epoch 200 are considered old enough, Alice1, Alice1|stale can be deleted.
Sharing a small commitment: Blockchains?

- Could post the commitment in a smart contract or OP_RETURN on bitcoin.
- Must trust the blockchain and its code.
- Even “light” clients could be too heavyweight.
- If billions of users query, could end up flooding the network with queries!
- Proposed mitigation: header relay network.
  - Similar to having a few nodes serving the commitment (i.e. blockchain view).
  - Might as well, just have the nodes agree on and serve the commitment...
  - Custom blockchain?
Commitments for One User

<table>
<thead>
<tr>
<th>Server Epochs</th>
<th>PK_1</th>
<th>PK_2</th>
<th>PK_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Alice</td>
<td>1</td>
<td>stale</td>
</tr>
<tr>
<td>500</td>
<td>Alice</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>Alice</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td></td>
<td></td>
<td>Alice</td>
</tr>
</tbody>
</table>
Commitments for One User Over Time

Server Epochs

100

Alice | 1

PK₁

500

Alice | 2

PK₂

Alice | 1 | stale

100k

Alice | 10

PK₁₀

Alice | 9 | stale

... 200k
(In)Valid Deletions

Don’t care about *obsolete* entries here

Server

<table>
<thead>
<tr>
<th>Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>100k ... 200k</td>
</tr>
</tbody>
</table>

Alice | 1
---|---
PK₁

Alice | 2
---|---
PK₂

Bob | 10
---|---
PK₁₀

Alice | 10
---|---
PK₁₀

Alice | 1 | stale
---|---|---

Alice | 9 | stale
---|---|---

Don’t care about obsolete entries here

Bob’s latest key: DO NOT DELETE!