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

Bob

1800-ali-ce

100-bob-ph

Need Each Other’s Public Keys

Want to send encrypted messages

Encrypted Messaging Service

+ Identity Provider

Username | Public Key
---------|------------
1800-ali-ce | PK_{alice}
100-bob-ph | PK_{bob}
End-to-End Encrypted Messaging

Encrypted Messaging Service + Identity Provider

Alice

PK_{malicious}

Bob

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.
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 $\mathbf{PK}_{\text{Alice}} \Rightarrow$ the server cannot get away with telling Bob her key is $\mathbf{PK}_{\text{BAD}}$. 
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: \( \text{com}_{\text{state}} \)

Serving commitments: Need a way for users to access \( \text{com}_{\text{state}} \)

<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>

Identity Provider

Identity Provider’s State
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

<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>

Committing to this: \( \text{com}_{\text{state}} \)
## 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): Orders of magnitude better</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): Orders of magnitude better</td>
</tr>
<tr>
<td>Users checking own key</td>
<td>● ZKP</td>
<td>● Impractical for server</td>
<td><strong>Secure compaction:</strong> Find a middle ground between requiring users always online and totally append-only.</td>
</tr>
<tr>
<td></td>
<td>● Always online</td>
<td>● Impractical for client</td>
<td></td>
</tr>
<tr>
<td></td>
<td>● Append-only data structures</td>
<td>● Ever growing storage costs</td>
<td></td>
</tr>
</tbody>
</table>
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: \( \text{com}_{\text{state}} \)

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

Identity Provider

Identity Provider’s State
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, e.g. 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, etc.
- 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.
- Let’s use independent witnesses!
  - Multiple trusted hardware instances.
  - Industry consortiums.
- Witnesses store the latest commitment and check new commitment is ok.
- Server collects signatures and forwards to users.
Consensus-less Strong Consistency

- Consistency, validity and termination are guaranteed similar to byzantine fault tolerant (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.

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

The graph shows latency (s) on the y-axis and throughput (updates/s) on the x-axis. There are three lines indicating different numbers of witnesses and batch sizes:

- 10 witnesses (batch: $2^{10}$)
- 20 witnesses (batch: $2^{10}$)
- 50 witnesses (batch: $2^{10}$)

The latency increases as the throughput increases, especially for the 50 witnesses case, which shows a steeper increase compared to the other two cases.
Summary

- **Goal:** Scale to BILLIONS of users.
- **Two components:**
  - Scaling underlying cryptographic primitive (with easy to standardize cryptographic tools)
  - Scaling distribution of small commitments (for all kinds of users in all kinds of places)
- We explored and addressed both problems while ensuring privacy.
- Other problems show up also (see paper).
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](https://eprint.iacr.org/2023/081.pdf))
- **Open source implementations:** linked in the paper.