CONIKS (KEY TRANSPARENCY)
The problem of the PKI (Public key infrastructure)

• A long standing problem has been to distribute public keys securely in the presence of attackers

• Use cases:
  • Secure messaging: Alice needs to send an encrypted message to Bob and needs Bob’s public key to encrypt the message
  • Web surfing and https: when Alice’s browser contacts amazon.com over https, we need Amazon’s PK
  • Man in the middle attacker can intercept PK and return incorrect PK
Centralized Key Management

Register [(alice $\rightarrow$ PK$_A$)]

Register [(bob $\rightarrow$ PK$_B$)]

Email + Key Provider

foo.com

User

Alice

User

Bob
Centralized Key Management

1. Register (alice → PK_A)
2. Look up Alice’s public key: PK_A
3. Send message encrypted to PK_A, signed by SK_B
Still, Key Server Compromise

1. Register (alice → PK_A)

2. Look up Alice’s public key: PK'_A

Email + Key Provider foo.com

Equivocates: returns incorrect value

User Alice

User Bob

This isn’t Alice’s real key!
Still, Key Server Compromise

Email + Key Provider \textit{foo.com}

1. User Alice
2. User Bob

3. Send message encrypted to \( \text{PK}'_A \), signed by \( \text{SK}_B \)

4. Read message encrypted to \( \text{PK}'_A \)
KT’s Approach: Consistent Identities

- Users expect consistency of online identities.
- Certificates make no statement about correctness.
Consistency

No unexpected key changes

Alice’s key today = Alice’s key yesterday

Non-Equivocation

Key seen by Alice = Key seen by Bob
Solution: a promising new generation

- **CONIKS**
  - An alternative Public Key Infrastructure (PKI)
  - Embedded in Google’s project called Key Transparency or Trillian
    - The key data structure is the Log-backed Verified Map

- **Certificate Transparency**
  - Similar technology to Key Transparency but aimed at certificates
  - Currently mandated by Chrome, supported by Firefox and others

These are essentially blockchains, with decentralized security but hosted centrally
CONIKS [Melara et al. 2014]

• End-User Key Management Service.

• Consistent name-to-key bindings.

• Verifiable key directories \(\Rightarrow\) Untrusted identity providers.

• Clients verify consistency in-band.
  \(\Rightarrow\) automated key management
CONIKS Overview

Untrusted Identity Provider

Register (alice $\rightarrow$ PK$_A$)

User Alice

Client A

Untrusted Identity Provider foo.com

Register (bob $\rightarrow$ PK$_B$)

Client B

User Bob
Using CONIKS for E2E Encryption

1. Look up the public key for Alice: $PK_A$.
2. Look up the public key for Bob: $PK_B$.
3. Send message encrypted to $PK_A$, signed by $SK_B$.
4. Look up public key for Bob: $PK_B$, verify signature, decrypt using $SK_A$. 

Client A

User Alice

Untrusted Identity Provider
foo.com

Client B

User Bob
Strawman Design

Untrusted Identity Provider
foo.com

Validity Checks
$O(N)$ storage per client

Non-equivocation Checks
$O(N^2)$ downloads per client

Client A

Client B

Client C

Client D

N = 4
CONIKS Design

• Divide time into epochs.

• Providers generate snapshots of directory.
  ➔ Clients do not check individual bindings.

• Providers distribute snapshots to other providers.
  ➔ Build publicly verifiable history

• Non-repudiation: Snapshots are digitally signed.
Efficient Snapshots

foo.com

i

alice:

PK

Alice

H(sub_L)

H(sub_R)

root

i

charlie:

PK

Charlie

i

emily:

PK

Emily

i

george:

PK

George

snapshot = root signed by provider
username -> index; arrange usernames based on index in the leaves
Checking Consistency

• Use snapshots to check for consistency.

• Clients need to ensure bindings are included in snapshots.
  • Lookups: prove inclusion of bindings in directory.

• Clients check validity, non-equivocation.

• Providers cross-verify each other for non-equivocation.
Proving Inclusion: Authentication Paths

A user can ask the server to prove inclusion of a binding \((i \rightarrow PK)\) against a signed snapshot: this can be the owner of the binding, or a user wanting to fetch the \(PK\) for \(i\).
Proving Inclusion: Authentication Paths

\[ i_{alice} : PK_{Alice} \]

\[ H(sub_L) \]

\[ H(sub_R) \]

\[ root \]

\[ foo.com \]
What we have so far

• A client, say Alice can check that her binding is correct in a signed snapshot
• Another client fetching Alice’s key can check Alice’s binding against the signed snapshot

What is missing?
• Provider can equivocate about the signed snapshot (e.g., give different signed snapshots
• How to address this?
Checking Non-Equivocation: Snapshot History

Trillian calls this: **Log-backed Verified Map**

The provider gives the signed last snapshot to other providers, who **check that it is well-formed** (e.g., it contains a hash of the previous snapshot) and **cross-check** that they all received the same snapshot!
Why do we need to chain the hashes?

• Otherwise, providers need to exchange the whole history to check for consistency
• With a hash chain forks are not possible
• Why?
  • Collusion resistance of the hash
Checking Non-Equivocation: Snapshot History

The server can try a fork attack, but after fork the provider must maintain these forked hash chains for the rest of time, and not allow clients seeing one branch of the hash chain to communicate with anyone seeing the other branch.
CONIKS Protocols

- Registration: New bindings/updated keys.
- Lookups: Clients check bindings are included in directory.
- Monitoring: Clients check for validity of bindings.
- Auditing: Clients & providers check for non-equivocation.
CONIKS Protocols

• Registration: New bindings/updated keys.

• Lookups: Clients check bindings are included in directory.

• Monitoring: Clients check for validity of bindings.

• Auditing: Clients & providers check for non-equivocation.
Registration Protocol

1. User Alice
   Generate key pair \((PK_A, SK_A)\)

2. Client A
   Register \((alice \rightarrow PK_A)\)
   \(2\)

3. Untrusted Identity Provider
   \(*foo.com*
   
   Check "alice" is not taken already

   Temporary binding = \([[alice \rightarrow PK_A] + \text{next epoch}], \text{Sig}(TB)\)

   Why do we need this "promise" signed?
   (provider might discard update forever)
CONIKS Protocols

- Registration: new bindings/updated keys.

- Lookups: Clients check bindings are included in directory.

- Monitoring: Clients check for validity of bindings.

- Auditing: Clients & providers check for non-equivocation.
Lookups without Inclusion Proofs

User Alice

Client A

Untrusted Identity Provider foo.com

Register (alice → PKₐ)

User Bob

Client B

Publish new snapshot including PKₐ

PKₐ

PKₐ

...

Provider regularly generates and publishes snapshots
Lookups without Inclusion Proofs

Untrusted Identity Provider foo.com

This isn't alice's real key!

Look up the public key for alice: PK'_A

Return Fake Key
Lookups **without Inclusion Proofs**

Untrusted Identity Provider  
*foo.com*

User Alice  
Client A  

Client B  

Verify & accept snapshot  

No proof fake key is inconsistent with snapshot
Lookups without Inclusion Proofs

Untrusted Identity Provider

foo.com

User Alice

Client A

Send message encrypted to \( PK'_A \), signed by \( SK_B \)

Read message encrypted to \( PK'_A \)

Untrusted Identity Provider

User Bob

Client B

Provider can read Bob’s message
Lookup Protocol

• Ensures bindings are consistent with snapshots.

• Prevents malicious providers from publishing fake keys and not leaving any evidence of the misbehavior.
Lookup Protocol

1. **Untrusted Identity Provider**
   - *foo.com*
   - Lookup (alice $\rightarrow$ PK$_A$)

2. **Client A**
   - Send proof of inclusion: Authentication path for (alice $\rightarrow$ PK$_A$)

3. **Client B**
   - Verify auth. path

- User Alice
- Client A
- Untrusted Identity Provider
- User Bob
CONIKS Protocols

- Registration: new bindings/updated keys.
- Lookups: Clients check bindings are included in directory.
- Monitoring: Clients check for validity of bindings.
- Auditing: Clients & providers check for non-equivocation.

What can happen without monitoring?
Communication without Monitoring

Untrusted Identity Provider

foo.com

User Alice

Client A

Register (alice \(\rightarrow PK_A\))

Client B

Look up the public key for alice: \(PK_A\)

User Bob

Epoch 1: Key has not Changed
Communication without Monitoring

User Alice

User Bob

Client A

Client B

Untrusted Identity Provider

foo.com

Send message encrypted to \( \text{PK}_A \), signed by \( \text{SK}_B \)

Epoch 1: Provider cannot read Bob’s message
Communication without Monitoring

Untrusted Identity Provider foo.com

This isn’t Alice’s real key!

Look up the public key for alice: \( PK'_A \)

Epoch 2: Key has been Changed
Communication without Monitoring

Untrusted Identity Provider

foo.com

Read message encrypted to PK'_{A}

Send message encrypted to PK'_{A}, signed by SK_{B}

User Alice

Client A

Client B

User Bob

Epoch 2: Provider can read Bob’s message
Monitoring Protocol

- Ensures public keys do not change unexpectedly.

- Prevents malicious providers from getting away with replacing existing keys.
Monitoring Protocol

Every user is responsible for monitoring their binding

1. Lookup (alice → PK_A)
2. Send (alice → PK_A) + authentication path
3. Verify validity of binding & auth. path

How often does each user need to check his/her binding?
CONIKS Protocols

• Registration: new bindings/updated keys.

• Lookups: Clients check bindings are included in directory.

• Monitoring: Clients check for validity of bindings.

• Auditing: Clients & providers check for non-equivocation.

What can happen without auditing?
Communication without Auditing

Clients register legitimate Keys
Communication without Auditing

Untrusted Identity Provider
foo.com

Provider creates two versions of its Directory

client A sees PK'\textsubscript{B} as bob’s public key

client B sees PK'\textsubscript{A} as alice’s public key

User Alice

Client A

User Bob

Client B
Communication without Auditing

The attacker can provide different but well-formed snapshots

Provider presents two different but valid snapshots
Auditing Protocol

• Ensures clients have consistent views of directories.

• Clients query random other providers for snapshots observed from given provider.

• Prevents malicious providers from getting away with publishing inconsistent versions of key directory.
Auditing Protocol

1. Verify foo.com’s snapshot history

Untrusted Identity Provider:
- rando.com

Distribute signed snapshot

Untrusted Identity Provider:
- foo.com

Verify foo.com’s snapshot history

Untrusted Identity Provider:
- bar.com

Client A
User Alice

Client B
Auditing Protocol

1. Verify foo.com’s snapshot history
2. Request most recent snapshot
3. Send most recent snapshot
4. Verify foo.com’s snapshot history
Checking Snapshot History

- Check signature on snapshot
- Compare $H(\text{root}_{\text{prev}})$ of cached and new snapshot
- Check passed
Checking Snapshot History

Check signature on snapshot

Valid

Compare $H(root_{prev})$ of cached and new snapshot

$root_{prev}$

Not matching

Fail
Auditing Protocol

Untrusted Identity Provider
*rando.com*

Untrusted Identity Provider
*foo.com*

Untrusted Identity Provider
*bar.com*

User Alice

Client A

Request snapshot observed for foo.com

Client B

Request snapshot observed for foo.com

Compare observed snapshots for foo.com

Is it clear why foo cannot equivocate?
Privacy problems with this protocol?

- Auditors and the provider see contents of directory
- Authentication paths reveal other entries in the directory
Prototype Implementation

- CONIKS Chat: Secure chat service.
- Stand-alone provider alongside XMPP server.
- Supports registration, lookups and basic auditing.
Server performance

The graph shows the time for 1k updates (s) for a range of number of existing users. The x-axis represents the number of existing users, starting from $1 \times 10^5$ to $1 \times 10^7$. The y-axis represents the time for 1k updates in seconds, ranging from 0 to 3.5. Error bars indicate the standard deviation. The graph indicates an increasing trend in time as the number of users increases.
### Client Overheads

- $N \approx 4$ bill. users, $n \approx 2$ mill. updates/epoch, $d = 24$ epochs/day.

<table>
<thead>
<tr>
<th></th>
<th>Download Requirements</th>
<th>Storage Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup (per binding)</td>
<td>&lt; 1.4KB</td>
<td>0B</td>
</tr>
<tr>
<td>Monitoring (epoch)</td>
<td>&lt; 800B</td>
<td>~ 300B</td>
</tr>
<tr>
<td>Monitoring (day)</td>
<td>&lt; 20KB</td>
<td>~ 300B</td>
</tr>
<tr>
<td>Auditing (epoch, per snapshot)</td>
<td>~ 100B</td>
<td>~ 100B</td>
</tr>
<tr>
<td>Auditing (day, per snapshot)</td>
<td>~ 2.5KB</td>
<td>~ 100B</td>
</tr>
</tbody>
</table>
Client Overheads

- $N \approx 4 \text{ bill. users}$, $n \approx 2 \text{mill. updates/epoch}$, $d = 24$ epochs/day.

<table>
<thead>
<tr>
<th></th>
<th>Download Requirements</th>
<th>Storage Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup (per binding)</td>
<td>&lt; 1.4KB</td>
<td>0B</td>
</tr>
<tr>
<td>Monitoring (epoch)</td>
<td>&lt; 800B</td>
<td>~ 300B</td>
</tr>
<tr>
<td>Monitoring (day)</td>
<td>&lt; 20KB</td>
<td>~ 300B</td>
</tr>
<tr>
<td>Auditing (epoch, per snapshot)</td>
<td>~ 100B</td>
<td>~ 100B</td>
</tr>
<tr>
<td>Auditing (day, per snapshot)</td>
<td>~ 2.5KB</td>
<td>~ 100B</td>
</tr>
</tbody>
</table>
## Client Overheads

- $N \approx 4 \text{ bill. users}, \ n \approx 2 \text{ mill. updates/epoch}, \ d = 24 \text{ epochs/day.}$

<table>
<thead>
<tr>
<th></th>
<th>Download Requirements</th>
<th>Storage Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup (per binding)</td>
<td>&lt; 1.4KB</td>
<td>~ 300B</td>
</tr>
<tr>
<td>Monitoring (epoch)</td>
<td>&lt; 800B</td>
<td>~ 300B</td>
</tr>
<tr>
<td>Monitoring (day)</td>
<td>&lt; 20KB</td>
<td>~ 300B</td>
</tr>
<tr>
<td>Auditing (epoch, per snapshot)</td>
<td>~ 100B</td>
<td>~ 100B</td>
</tr>
<tr>
<td>Auditing (day, per snapshot)</td>
<td>~ 2.5KB</td>
<td>~ 100B</td>
</tr>
</tbody>
</table>
Client Overheads

<table>
<thead>
<tr>
<th></th>
<th>Download Requirements</th>
<th>Storage Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup (per binding)</td>
<td>&lt; 1.4KB</td>
<td>~ 300B</td>
</tr>
<tr>
<td>Monitoring (epoch)</td>
<td>&lt; 800B</td>
<td>~ 300B</td>
</tr>
<tr>
<td>Monitoring (day)</td>
<td>&lt; 20KB</td>
<td>~ 300B</td>
</tr>
<tr>
<td>Auditing (epoch, per snapshot)</td>
<td>~ 100B</td>
<td>~ 100B</td>
</tr>
<tr>
<td>Auditing (day, per snapshot)</td>
<td>~ 2.5KB</td>
<td>~ 100B</td>
</tr>
</tbody>
</table>

- $N \approx 4$ bill. users, $n \approx 2$ mill. updates/epoch, $d = 24$ epochs/day.
CONIKS summary

• CONIKS is an efficiently verifiable, privacy-preserving key management service for end-user public keys.

• End-to-end security can be practical for end-users.

• Some industry adoption: e.g. Google’s Key Transparency
Open questions/projects

Progress on any of:

• Scalability:
  • Users should not have to check their bindings every epoch
  • Provider should be able to scale to the whole world of people and IoT devices, wants to generate a snapshot every second

• Privacy:
  • Provider sees who talks to who
  • Auditors sees many bindings and changes