VeRSA
Verifiable Registries with Efficient Client Audits from RSA Authenticated Dictionaries

Nirvan Tyagi       Ben Fisch       Andrew Zitek
Joseph Bonneau    Stefano Tessaro

ACM CCS 2022
Setting: Verifiable registries

Applications: certificate transparency, key transparency, binary transparency, etc.
Setting: Verifiable registries

Key-value Mapping

Server

<table>
<thead>
<tr>
<th>Name</th>
<th>Public Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>ae7b</td>
</tr>
<tr>
<td>Bob</td>
<td>422a</td>
</tr>
<tr>
<td>Cindy</td>
<td>87bd</td>
</tr>
</tbody>
</table>

e.g. public key identities

software binary checksums

domain name routing info
Setting: Verifiable registries

Key-value Mapping

<table>
<thead>
<tr>
<th>Name</th>
<th>Digest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>ae7b</td>
</tr>
<tr>
<td>Bob</td>
<td>422a</td>
</tr>
<tr>
<td>Cindy</td>
<td>87bd</td>
</tr>
</tbody>
</table>

Digest \( d \)

Server

e.g. public key identities
     software binary checksums
     domain name routing info
Setting: Verifiable registries

Key-value Mapping

Alice: ae7b
Bob: 422a
Cindy: 87bd
...

Digest

Goal 1
Users can \textit{lookup} values and verify they are consistent with what other users receive.

e.g.
- public key identities
- software binary checksums
- domain name routing info

Server

Bob

Carol

Lookup Alice

ae7b, d, π\textsubscript{Alice}

Lookup Alice

ae7b, d, π\textsubscript{Alice}
Users can monitor key-value mappings to detect unexpected modifications.

**Goal 2** Users can *monitor* key-value mappings to detect unexpected modifications.

**e.g.** public key identities, software binary checksums, domain name routing info.
Users can monitor key-value mappings to detect unexpected modifications.

**Key-value Mapping**

- Alice: ae7b
- Bob: 422a
- Cindy: 87bd

**Digest**

- $d_0$
- $d_1$
- $d_2$
- $d_3$
- $\cdots$

**New digests published over time**

**Goal 2**

Users can monitor key-value mappings to detect unexpected modifications.

**Problem**

User must monitor every published digest.

**Setting: Verifiable registries**

- Public key identities
- Software binary checksums
- Domain name routing info
Previous approaches: Trusted third-party auditors

[CONIKS’15, SEEMless’19, Mog’20]

Trusted third-party auditors verify version-only invariant is preserved between digests. Invariant allows efficient detection of unexpected changes by user.
Previous approaches: Trusted third-party auditors

[CONIKS’15, SEEMless’19, Mog’20]

New digests published over time

Alice can detect misbehavior without verifying all digests!

Trusted third-party auditors verify **version-only** invariant is preserved between digests. Invariant allows efficient detection of unexpected changes by user.

[CONIKS’15, SEEMless’19, Mog’20]
This work: Enabling efficient client auditability
This work: Enabling efficient client auditability

Contribution 1: New RSA key-value commitment with succinct proofs that invariant is preserved over ranges of digests.
Contribution 1
New RSA key-value commitment with succinct proofs that invariant is preserved over ranges of digests

Contribution 2
Checkpointing technique to ensure user views remain eventually consistent even when auditing distinct ranges of digests
Prior work: Invariant proofs for Merkle trees

$$n_s = H(n_{s||0} \ || \ n_{s||1})$$

Digest $d_i$

$H(k_1)$

$H(k_2)$

$k_2 \ || \ \text{ver}_2 \ || \ \text{val}_2$

[CONIKS ‘15, SEEMless ‘19, Mog ‘20, Verdict ‘21]
Prior work: Invariant proofs for Merkle trees

[CONIKS '15, SEEMless '19, Mog '20, Verdict '21]
Prior work: Invariant proofs for Merkle trees

\[ n_s = H(n_{s||0} \| n_{s||1}) \]

\[ d_i \rightarrow d_{i+1} \]

[CONIKS ‘15, SEEMless ‘19, Mog ‘20, Verdict ‘21]
Prior work: Invariant proofs for Merkle trees

[CONIKS '15, SEEMless '19, Mog '20, Verdict '21]
Prior work: Invariant proofs for Merkle trees

To obtain succinct invariant proofs over a range of digests, we compress the Merkle paths proof into a generic-circuit SNARK, which enables SNARK recursion/aggregation.
Prior work: Invariant proofs for Merkle trees

To obtain succinct invariant proofs over a range of digests, we compress the Merkle paths proof into a generic-circuit SNARK, which enables SNARK recursion/aggregation.

Problem

SNARK provers are concretely expensive, and every Merkle path must be included.
Our work: Invariant proofs for RSA KV commitments

[AR Asiacrypt ‘20]
Our work: Invariant proofs for RSA KV commitments

$\pi_{\text{RSA}}$
Our work: Invariant proofs for RSA KV commitments

\[ d_i \xrightarrow{\text{upd } k_2} \xrightarrow{\text{upd } k_1} \xrightarrow{\text{upd } k_3} d_{i+1} \]

\[ \pi_{\text{RSA}} \]

Constant-size and constant-verif invariant proof!
Using variant of proof of knowledge of integer exponentiation [Wesolowski ‘19][BBF ‘19]
Our work: Invariant proofs for RSA KV commitments

\[ d_i \xrightarrow{\text{upd } k_2} d_{i+1} \xrightarrow{\text{upd } k_1} \text{upd } k_3 \]

\[ \pi_{\text{RSA}} \]

\[ \pi_{\text{SNARK}} \]
Our work: Invariant proofs for RSA KV commitments

Circuit size linearly dependent on number of key updates.

Constant circuit size independent of number of key updates.
Our work: Invariant proofs for RSA KV commitments

\[ d_i \xrightarrow{\text{upd } k_2} \xrightarrow{\text{upd } k_1} \xrightarrow{\text{upd } k_3} d_{i+1} \]

- Small circuit translates to high update throughput for invariant proofs.
- Lookup proofs for RSA key-value commitment are expensive to compute on demand.

Constant circuit size independent of number of key updates.

[AR Asiacrypt '20]
This work: Enabling efficient client auditability

Contribution 1: New RSA key-value commitment with succinct proofs that invariant is preserved over ranges of digests

Contribution 2: Checkpointing technique to ensure user views remain eventually consistent even when auditing distinct ranges of digests

- When auditing a range, users additionally audit logarithmic checkpoints within range
- Two users are guaranteed to eventually share checkpoints and will be able to detect inconsistencies if they exist
Inconsistent user views: Oscillation attacks

Since users are not guaranteed to see the same digests, a malicious platform may “oscillate”, publishing digests for two different valid data structures at different time steps.
An invariant proof is verified for a sequence of “checkpoints”. The number of checkpoints between two digests is logarithmic in the size of the range.
An invariant proof is verified for a sequence of “checkpoints”. The number of checkpoints between two digests is logarithmic in the size of the range.
An invariant proof is verified for a sequence of “checkpoints”. The number of checkpoints between two digests is logarithmic in the size of the range.
An invariant proof is verified for a sequence of "checkpoints". The number of checkpoints between two digests is logarithmic in the size of the range.

- Overlapping ranges are guaranteed to share at least one checkpoint.
Eventual inconsistency detection via checkpointing

- An invariant proof is verified for a sequence of “checkpoints”. The number of checkpoints between two digests is logarithmic in the size of the range.
- Overlapping ranges are guaranteed to share at least one checkpoint.
Eventual inconsistency detection via checkpointing
Eventual inconsistency detection via checkpointing
Eventual inconsistency detection via checkpointing

Checkpoints determined by compact subtree representation of range. The number of checkpoints will be logarithmic in the size of the range.
Checkpoints determined by compact subtree representation of range. The number of checkpoints will be logarithmic in the size of the range.
Eventual inconsistency detection via checkpointing

Checkpoints determined by compact subtree representation of range. The number of checkpoints will be logarithmic in the size of the range.

Shared checkpoints between overlapping ranges guaranteed to exist – see paper!
This work: Enabling efficient client auditability

Contribution 1
New RSA key-value commitment with succinct proofs that invariant is preserved over ranges of digests

Contribution 2
Checkpointing technique to ensure user views remain eventually consistent even when auditing distinct ranges of digests
- When auditing a range, users additionally audit logarithmic checkpoints within range
- Two users are guaranteed to eventually share checkpoints and will be able to detect inconsistencies if they exist
Implementation and performance evaluation

- RSA key-value commitment and invariant proofs
- R1CS constraints for RSA algorithms in arkworks ecosystem for zkSNARKs
- Open source: github.com/nirvanyagi/versa
Implementation and performance evaluation

- RSA key-value commitment and invariant proofs
- R1CS constraints for RSA algorithms in arkworks ecosystem for zkSNARKs
- Open source: github.com/nirvantyagi/versa

Comparison to Merkle Tree baseline: Server with 32 CPU cores + 512 GB memory

- **Client verification costs**: similar
  - Proofs < 20kB, verify in < 100ms

- **Update proof throughput**: 10x-400x higher
  - Prototype achieves 60-90 updates/second on a single server

- **Lookup proof costs**: substantially worse
  - VeRSA limited to registries of ~millions of entries due to $O(n^2)$ costs
  - Millions of entries can be handled with $O(n \log n)$ batch computation costs
Potential application: *binary transparency*

**Characteristics:**
- Medium overall registry size
- Relatively high update frequency
- Moderate latency is acceptable (~30 minutes)

**Examples:**
- Ubuntu package repo: 106k packages, mean 3.4 versions/year
- Apple iOS app store: 2.1M apps, mean 52.5 versions/year
Conclusion

- VeRSA: New design for verifiable registry enabling efficient client-auditing
  - New RSA key-value commitments and constant-size invariant proofs
  - New client auditing approach that maintains eventual consistency
- Suitable for binary transparency applications with medium-size registries
  - Bottleneck: RSA lookup proof computation
- Open source: github.com/nirvantyagi/versa

eprint.iacr.org/2021/627
Backup slides
Trusted third-party auditors verify \textit{append-only} invariant is preserved between digests. Invariant allows efficient detection of unexpected changes by user.
Invariant proofs: RSA key-value commitments

Digest \( d_i = (d_{i,1}, d_{i,2}) = \left( g^{(\prod_j H(k_j)^{ver_j})} \cdot (\sum_j val_j / H(k_j)), g^{\prod_j H(k_j)^{ver_j}} \right) \)
Invariant proofs: RSA key-value commitments

[AR’20]

Digest $d_i = (d_{i,1}, d_{i,2}) = \left( g^{(\prod_j H(k_j)^{\text{ver}_j})} \cdot (\sum_j \text{val}_j / H(k_j)), g^{\prod_j H(k_j)^{\text{ver}_j}} \right)$

$d_i \xrightarrow{\text{upd } k_2} d_{i+1} : d_{i+1} = (d_{i+1,1}, d_{i+1,2}) = (d_{i,1}^{H(k_2)}, d_{i,2}^{\delta}, d_{i,2}^{H(k_2)})$

where $\delta = \text{val’}_2 - \text{val}_2$
Invariant proofs: RSA key-value commitments

Digest \( d_i = (d_{i,1}, d_{i,2}) = (g^{(\prod_j H(k_j)^{\text{ver}_j})} \cdot (\sum_j \text{val}_j / H(k_j)), g^{\prod_j H(k_j)^{\text{ver}_j}}) \)

\[
d_i \xrightarrow{\text{upd } k_2} d_{i+1} : \quad d_{i+1} = (d_{i+1,1}, d_{i+1,2}) = (d_{i,1}^H(k_2), d_{i,2}^\delta, d_{i+1}^H(k_2))
\]

where \( \delta = \text{val}'_2 - \text{val}_2 \)

\[
d_i \xrightarrow{\text{upd } k_2} \xrightarrow{\text{upd } k_1} \xrightarrow{\text{upd } k_3} d_{i+1} : \quad d_{i+1} = (d_{i+1,1}, d_{i+1,2}) = (d_{i,1}^Z d_{i,2}^\Delta, d_{i,2}^Z)
\]

where \( \Delta = (\prod_j H(k_j)) \cdot (\sum_j \delta_j / H(k_j)) \)

\( Z = \prod_j H(k_j) \) for \( j \in \{1, 2, 3\} \)
Invariant proofs: RSA key-value commitments

[AR’20]

Digest $d_i = (d_{i,1}, d_{i,2}) = \left(g^{(\prod_j H(k_j)^{\text{ver}_j}) \cdot (\sum_j \text{val}_j / H(k_j))}, g^{\prod_j H(k_j)^{\text{ver}_j}}\right)$

$d_i \xrightarrow{\text{upd } k_2} d_{i+1} : d_{i+1} = (d_{i+1,1}, d_{i+1,2}) = \left(d_{i,1}^{H(k_2)}, d_{i,2}^\delta, d_{i,2}^{H(k_2)}\right)$

where $\delta = \text{val'}_2 - \text{val}_2$

$d_i \xrightarrow{\text{upd } k_2 \text{ upd } k_1 \text{ upd } k_3} d_{i+1} : d_{i+1} = (d_{i+1,1}, d_{i+1,2}) = \left(d_{i,1}^Z d_{i,2}^\Delta, d_{i,2}^Z\right)$

where $\Delta = (\prod_j H(k_j)) \cdot (\sum_j \delta_j / H(k_j))$

$Z = \prod_j H(k_j)$ for $j \in \{1, 2, 3\}$
Invariant proofs: RSA key-value commitments

Digest $d_i = (d_{i,1}, d_{i,2}) = \left( g^{(\prod_j H(k_j)_{\text{ver},j}) \cdot (\sum_j \text{val}_j / H(k_j))}, g^{\prod_j H(k_j)_{\text{ver},j}} \right)$

$d_i \xrightarrow{\text{upd } k_2} d_{i+1} \quad \quad \quad \quad \quad d_{i+1} = (d_{i+1,1}, d_{i+1,2}) = \left( H(k_2) d_{i,1}^{\delta}, d_{i,2}^{H(k_2)} \right)$

where $\delta = \text{val}_2' - \text{val}_2$

$d_i \xrightarrow{\text{upd } k_2} d_{i+1} \xrightarrow{\text{upd } k_1} d_{i+1} \xrightarrow{\text{upd } k_3} d_{i+1} \quad \quad \quad \quad \quad d_{i+1} = (d_{i+1,1}, d_{i+1,2}) = \left( d_{i,1}^{\Delta}, d_{i,2}^{Z} \right)$

where $\Delta = \left( \prod_j H(k_j) \right) \cdot \left( \sum_j \delta_j / H(k_j) \right)$, $Z = \prod_j H(k_j)$ for $j \in \{1, 2, 3\}$

Algebraic invariant proof (constant-size!)

Statement $\{ (\alpha, \beta) : d_{i+1,1} = d_{i,1}^{\alpha}d_{i,2}^{\beta} \land d_{i+1,2} = d_{i,2}^{\alpha} \} \rightarrow \pi_{\text{RSA}}$
Eventual inconsistency detection via checkpointing

Committed joint view

Alice

Bob
Eventual inconsistency detection via checkpointing

Committed joint view
Eventual inconsistency detection via checkpointing

Committed joint view

Alice

Bob
Eventual inconsistency detection via checkpointing

Committed joint view
Eventual inconsistency detection via checkpointing

Committed joint view
Eventual inconsistency detection via checkpointing

Committed joint view
Eventual inconsistency detection via checkpointing

Committed joint view
Eventual inconsistency detection via checkpointing
Eventual inconsistency detection via checkpointing

Checkpointing allows users to implicitly create an ordered consistent view that trails the current time step.
Eventual inconsistency detection via checkpointing

Checkpoints are determined by the minimum number of subtrees that span the range in the superimposed binary tree -- guaranteed to be logarithmic in range size!