BalanceProofs: Maintainable Vector Commitments with Fast Aggregation

Weijie Wang  Annie Ulichney  Charalampos Papamanthou

USENIX Security ’23
Vector Commitments

- Short commitment to an ordered sequence of values
- VC.Commit, VC.OpenAll, VC.UpdAll, VC.Agg, VC.Verify, ...
- Correctness, soundness (position binding)
- Maintainable (sublinear UpdAll), aggregatable \( \{\pi_i\}_{i \in I} \rightarrow \pi_I \)
- Example: Merkle trees
- Applications in verifiable storage, stateless blockchains, and more
Two Types of Vector Commitments

• Type I: not maintainable, but with fast aggregation
  • aSVC
    • SCN 2020, by Alin Tomescu et al.
  • Pointproofs
    • CCS 2020, by Sergey Gorbunov et al.

• Type II: maintainable, but with slow aggregation
  • Merkle trees
  • Hyperproofs
    • USENIX Security 2022, by Shravan Srinivasan et al.
BalanceProofs

First vector commitment that is maintainable with fast aggregation

<table>
<thead>
<tr>
<th>$n = 2^{20}$</th>
<th>Hyperproofs</th>
<th>BalanceProofs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UpdAll time</strong></td>
<td>1.55 ms</td>
<td>4.60 ms</td>
</tr>
<tr>
<td><strong>Agg time</strong></td>
<td>105 s</td>
<td>0.11 s</td>
</tr>
</tbody>
</table>
BalanceProofs

• A compiler

• Input VC scheme (aggregatable):
  • $O(n \log n)$ time to open all $n$ proofs
  • $O(1)$ time to update each individual proof $\pi_j$ after receiving an update request $Update (i, \delta)$

• Output VC scheme (aggregatable)
  • $O(\sqrt{n} \log n)$ time to update all proofs
  • $O(\sqrt{n})$ time to query any individual proof

We pick aSVC as input scheme
How our compiler works

• At the beginning, we have

\[ v_0 \quad v_1 \quad v_2 \quad \ldots \quad v_{n-1} \]

\[ \pi_0 \quad \pi_1 \quad \pi_2 \quad \ldots \quad \pi_{n-1} \]
How our compiler works

• When we receive an update request \((i_1, \delta_{i_1})\) …
How our compiler works

• When we receive an update request \((i_1, \delta_{i_1})\) ...

\[
\begin{align*}
C' & \quad \text{Update record} \\
\begin{array}{cccc}
\pi_0 & \cdots & \pi_{i_1} & \cdots & \pi_{n-1} \\
\nu_0 & \cdots & \nu_{i_1 + \delta_{i_1}} & \cdots & \nu_{n-1}
\end{array}
\end{align*}
\]
How our compiler works

• When we keep receiving update requests ...

\[ v_0 \ldots v_{i_1} + \delta_{i_1} \ldots v_{i_k} + \delta_{i_k} \ldots v_{n-1} \]

\[ C' \]

Update record

\( (i_1, \delta_{i_1}) \)

\( (i_2, \delta_{i_2}) \)

\( \vdots \)

\( (i_k, \delta_{i_k}) \)
How our compiler works

- When the number of records reaches $\sqrt{n}$,
How our compiler works

• When the number of records reaches $\sqrt{n}$, use $O(n \log n)$ time to open all proofs
How our compiler works

• When the number of records reaches $\sqrt{n}$, use $O(n \log n)$ time to open all proofs, and clear the record list.
How our compiler works

• Anytime if we need to get proof for position $j$,

$$C'$$

\[
\begin{array}{cccccccc}
\pi_0 & \cdots & \pi_j & \cdots & \pi_{i_k} & \cdots & \pi_{n-1} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\nu_0 & \cdots & \nu_j & \cdots & \nu_{i_1 + \delta_{i_k}} & \cdots & \nu_{n-1} \\
\end{array}
\]

Update record

\[
\begin{array}{cccc}
(i_1, \delta_{i_1}) \\
(i_2, \delta_{i_2}) \\
\vdots \\
(i_k, \delta_{i_k}) \\
\end{array}
\]
How our compiler works

• Anytime if we need to get proof for position $j$, apply each update in the record list to $\pi_j$ and get the new proof $\pi_j'$
How our compiler works

• Above all,
  • We need amortized $O\left(\frac{n \log n}{\sqrt{n}}\right) = O(\sqrt{n} \log n)$ time to do the update part
  • We need at most $O(\sqrt{n})$ time to get any individual proof
  • For any index set $I$, we need $O(|I|\sqrt{n})$ time to get each individual proof (and then we can do aggregation)
  • If $|I|\sqrt{n} > n \log n$, we can choose to open all proofs instead to get each proof
How our compiler works

• Above all, We can use amortization technique to improve the worst case
  • We need amortized \( O\left(\frac{n \log n}{\sqrt{n}}\right) = O(\sqrt{n} \log n) \) time to do the update part

• Extend the size of update list to \( 2\sqrt{n} \)

• When we have \( \sqrt{n} \) records, separate the \( O(n \log n) \) time computation in next \( \sqrt{n} \) updates

• When we have \( 2\sqrt{n} \) records, clear the first \( \sqrt{n} \) records in the list and start another \( O(n \log n) \) time computation
Bucketing BalanceProofs

- Cut the vector into buckets
- Reduce the time of $UpdAll$
- Ensure that digest is still $O(1)$ size
Basic Bucketing
Space-efficient Bucketing

\[ \phi(x, y) \]

\[ \phi_0(y) \]
\[ \phi_1(y) \]
\[ \phi_2(y) \]
\[ \phi_3(y) \]

\[ v_{0,0} \]
\[ v_{0,1} \]
\[ v_{0,2} \]
\[ v_{1,0} \]
\[ v_{1,1} \]
\[ v_{1,2} \]
\[ v_{2,0} \]
\[ v_{2,1} \]
\[ v_{2,2} \]
\[ v_{3,0} \]
\[ v_{3,1} \]
\[ v_{3,2} \]
Two-layer bucketing

• Introduce three variables
• First layer: $p$ buckets; second layer: $p \cdot t$ buckets (subvectors)
• Each subvector has size $\frac{n}{pt}$
• Pick $p = t = n^{1/4}$, $O(n^{1/4} \log n)$ UpdAll time, $O(n^{1/2})$ proof size
## Performance and Comparison

The table below compares the performance of different systems in terms of update all time, batch proof size, aggregation time, and verification aggregation time. The table is divided into two rows, each representing a different set of parameters.

### $n = 2^{20}$

<table>
<thead>
<tr>
<th></th>
<th>Hyperproofs</th>
<th>aSVC</th>
<th>Basic compiler</th>
<th>Two-layer bucketing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UpdAll time</strong></td>
<td>1.55 ms</td>
<td>98 s</td>
<td>3.03 s</td>
<td>4.60 ms</td>
</tr>
<tr>
<td><strong>Batch proof size</strong></td>
<td>51.6 KB</td>
<td>48 bytes</td>
<td>30~60 KB</td>
<td></td>
</tr>
<tr>
<td><strong>Agg time</strong></td>
<td>105 s</td>
<td>0.39 s</td>
<td>0.11 s</td>
<td></td>
</tr>
<tr>
<td><strong>VrfyAgg time</strong></td>
<td>12.9 s</td>
<td>0.43 s</td>
<td>0.20 s</td>
<td></td>
</tr>
</tbody>
</table>

### $n = 2^{30}$

<table>
<thead>
<tr>
<th></th>
<th>Hyperproofs</th>
<th>aSVC</th>
<th>Basic compiler</th>
<th>Two-layer bucketing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UpdAll time</strong></td>
<td>2.58 ms</td>
<td>&gt;20 hrs</td>
<td>136 s</td>
<td>19.0 ms</td>
</tr>
<tr>
<td><strong>Batch proof size</strong></td>
<td>51.6 KB</td>
<td>48 bytes</td>
<td>50~100 KB</td>
<td></td>
</tr>
<tr>
<td><strong>Agg time</strong></td>
<td>123 s</td>
<td>0.41 s</td>
<td>0.008 s</td>
<td></td>
</tr>
<tr>
<td><strong>VrfyAgg time</strong></td>
<td>17.4 s</td>
<td>0.44 s</td>
<td>0.11 s</td>
<td></td>
</tr>
</tbody>
</table>
Summary - BalanceProofs

• Both maintainable and aggregatable

• Compiler: balance UpdAll time and Query time by auxiliary lists

• Bucketing: balance UpdAll time and proof size
  • Basic bucketing, space-efficient bucketing, two-layer bucketing

Thanks!