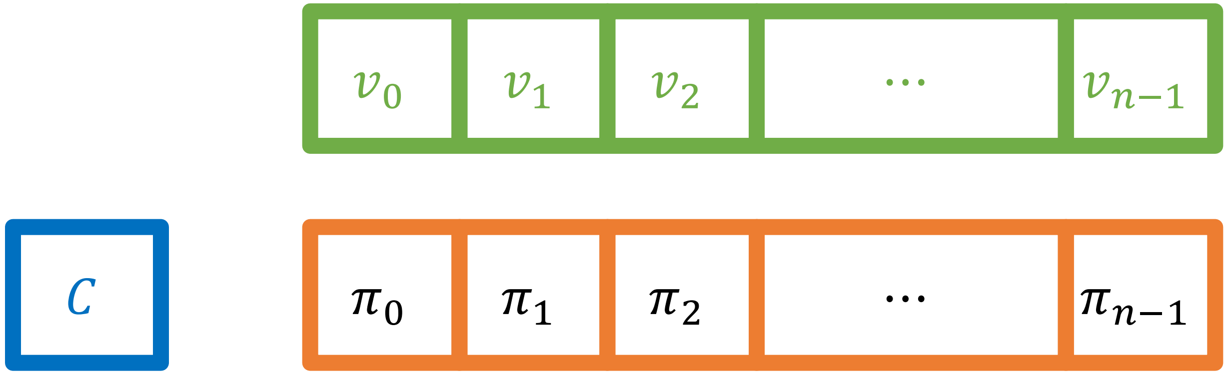
BalanceProofs: Maintainable Vector Commitments with Fast Aggregation

Weijie Wang   Annie Ulichney   Charalampos Papamanthou

Yale University

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

$n = 2^{20}$	**Hyperproofs**	**BalanceProofs**
UpdAll time	1.55 ms	4.60 ms
Agg time	105 s	0.11 s

BalanceProofs

- A compiler

We pick aSVC as input scheme

- Input VC scheme (aggregatable):
  - $O(n \log n)$ time to open all n proofs
  - $O(1)$ time to update each individual proof π_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

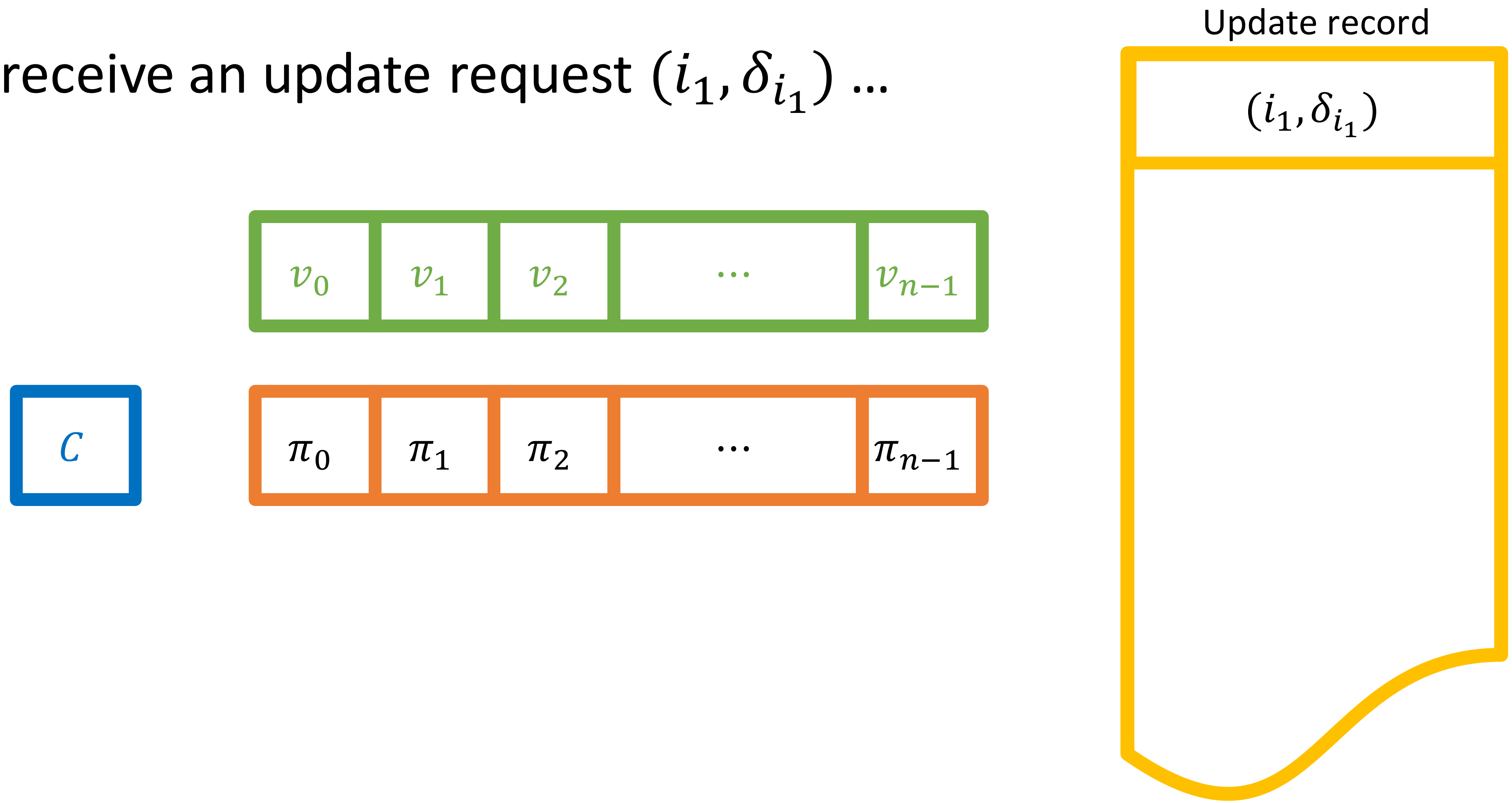
How our compiler works

- At the beginning, we have

	v_0	v_1	v_2	$\cdots$	v_{n-1}
C	π_0	π_1	π_2	$\cdots$	π_{n-1}

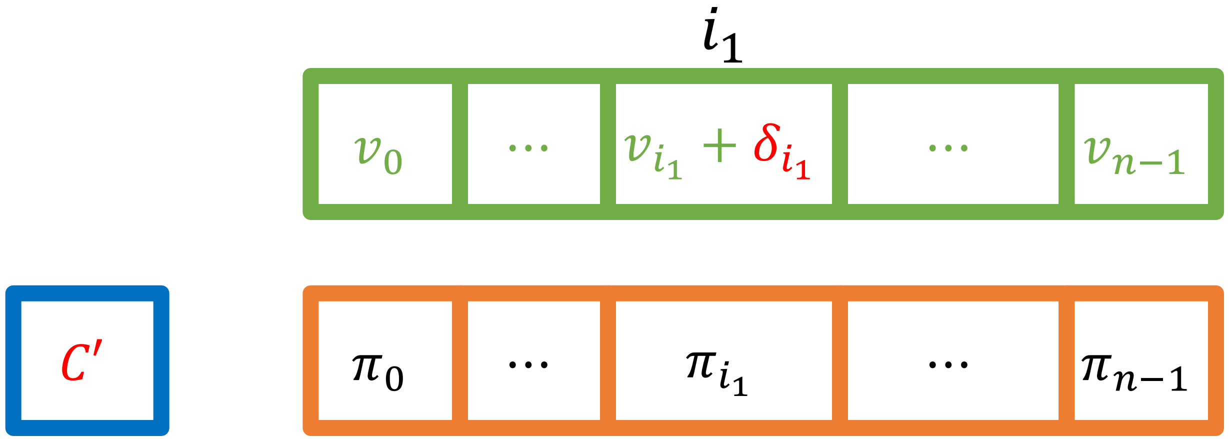
How our compiler works

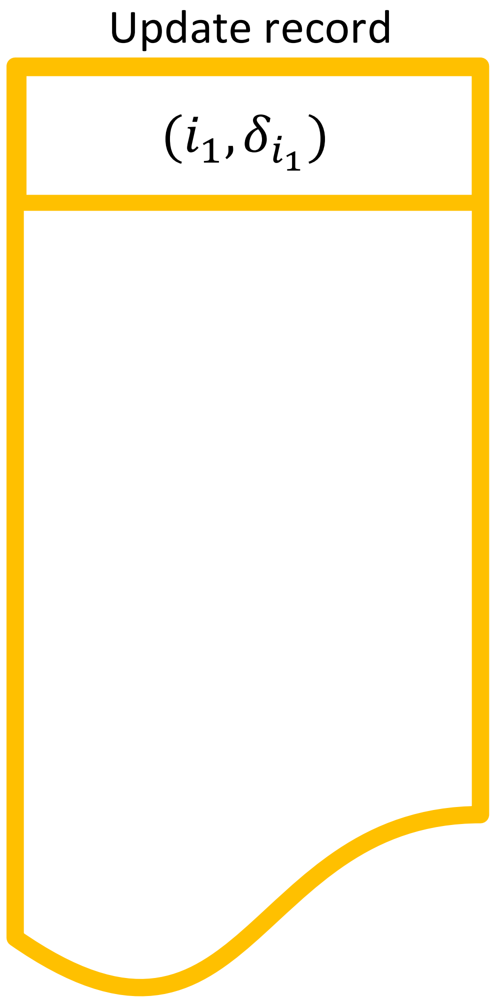
- When we receive an update request (i_1, δ_{i_1}) …



How our compiler works

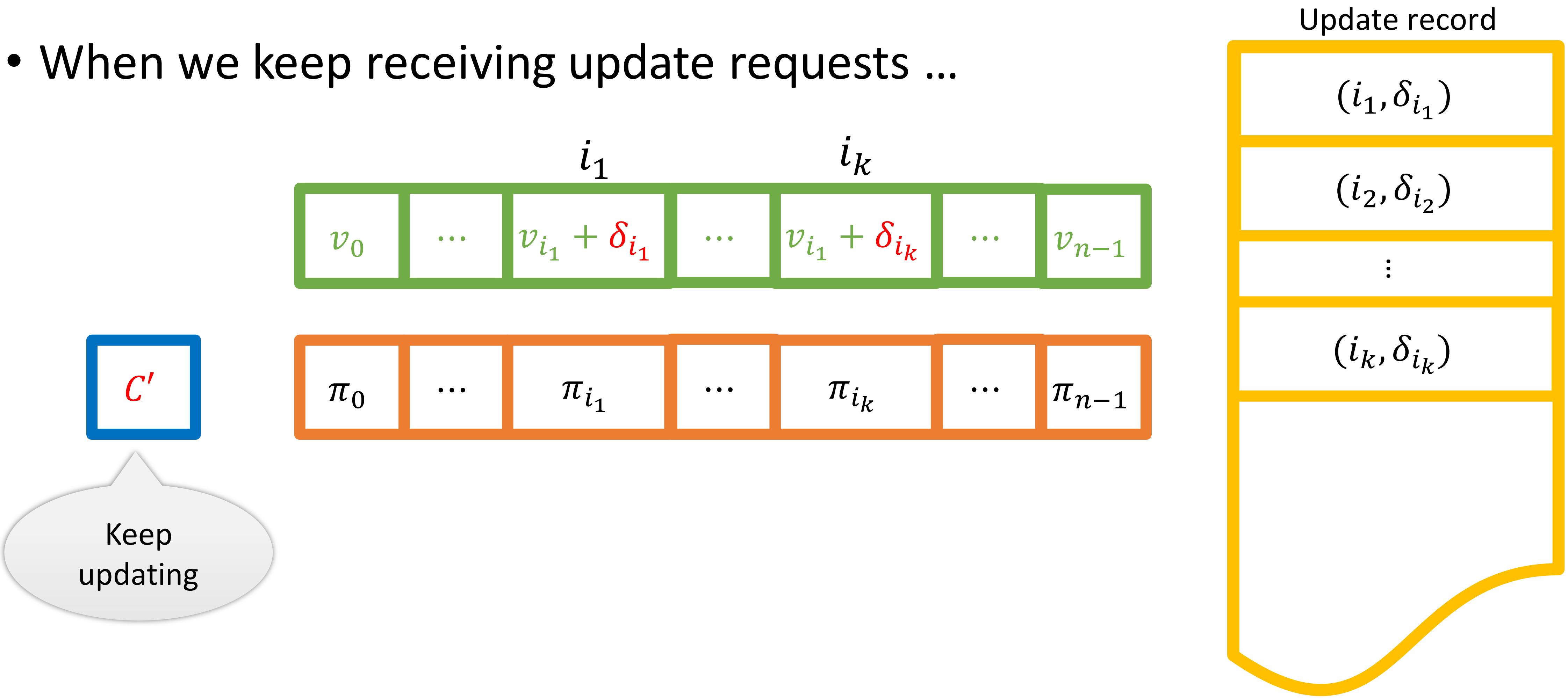
- When we receive an update request (i_1, δ_{i_1}) …





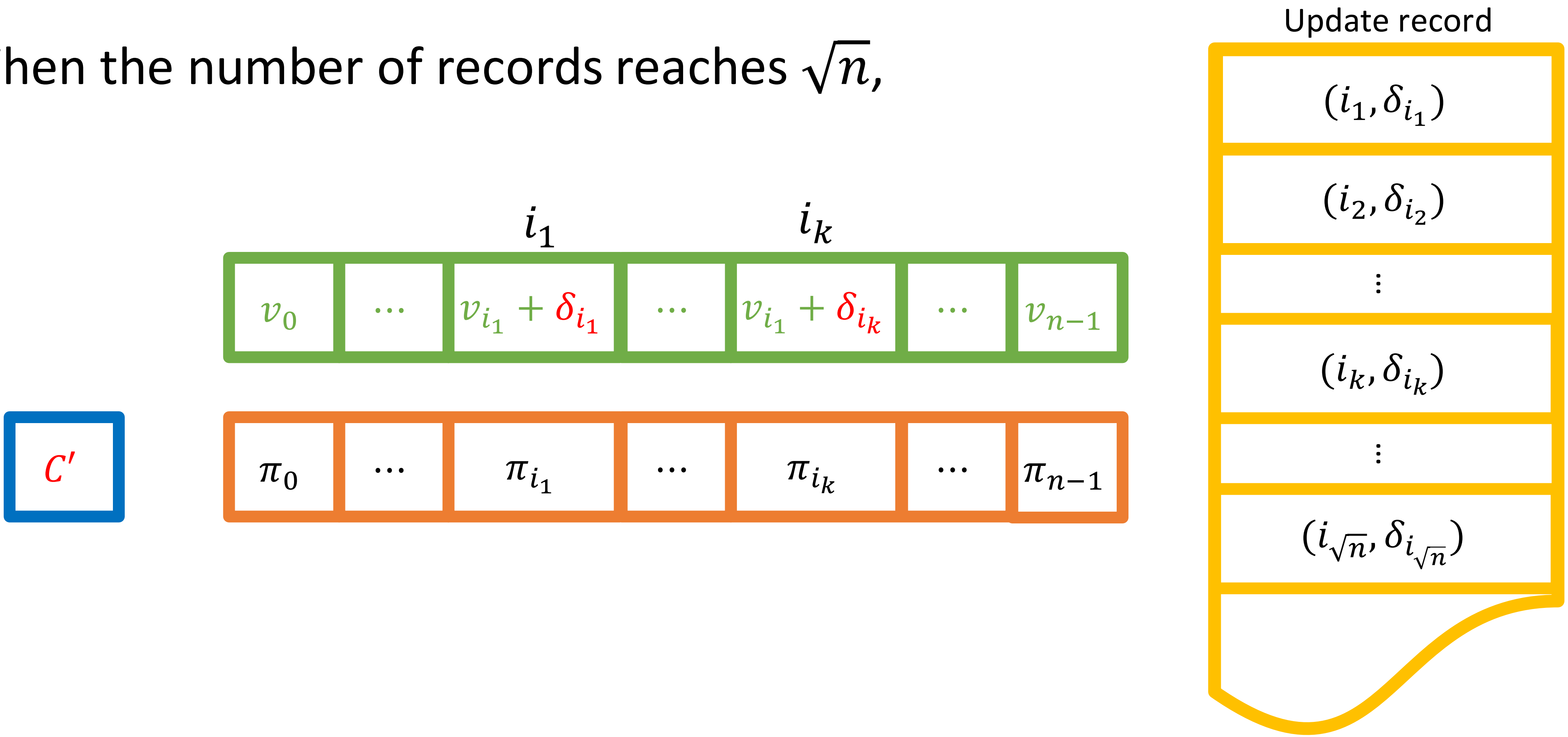
How our compiler works

- When we keep receiving update requests …



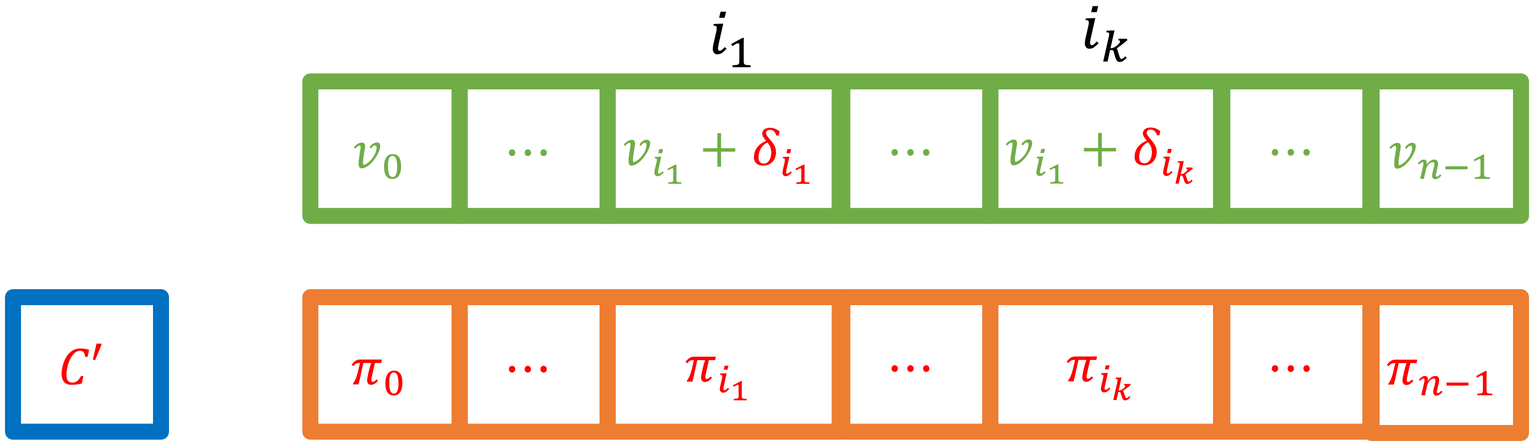
How our compiler works

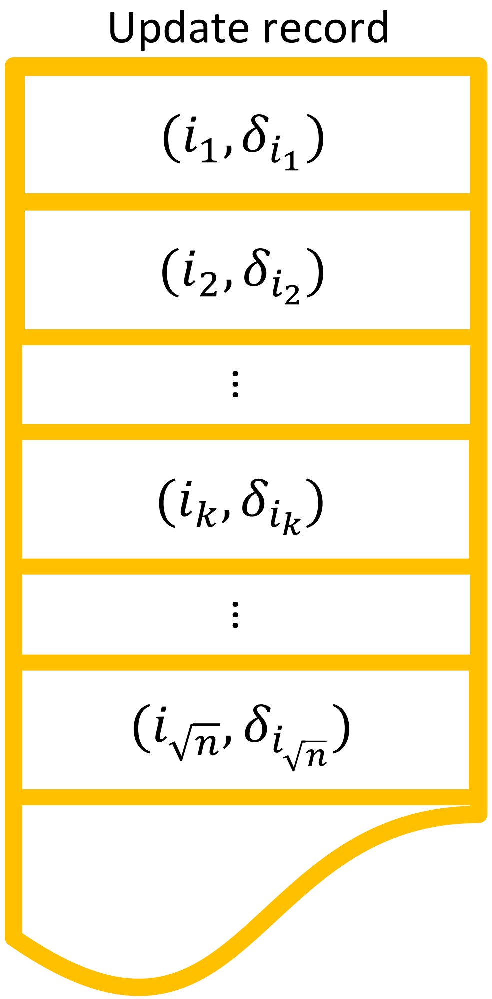
- When the number of records reaches $\sqrt{n}$,



How our compiler works

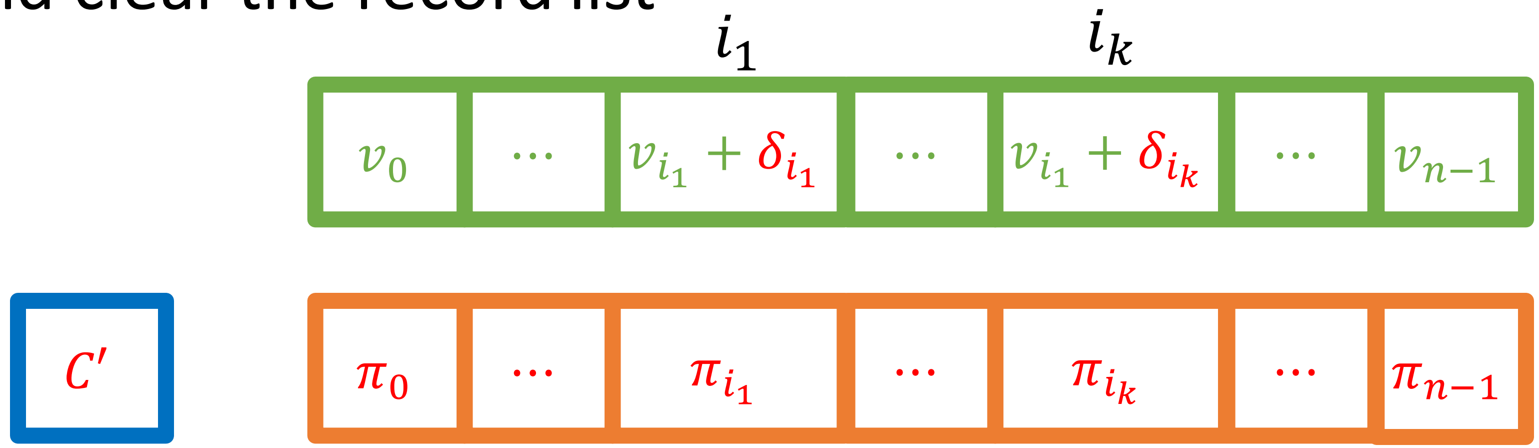
- When the number of records reaches $\sqrt{n}$, use $O(n \log n)$ time to open all proofs

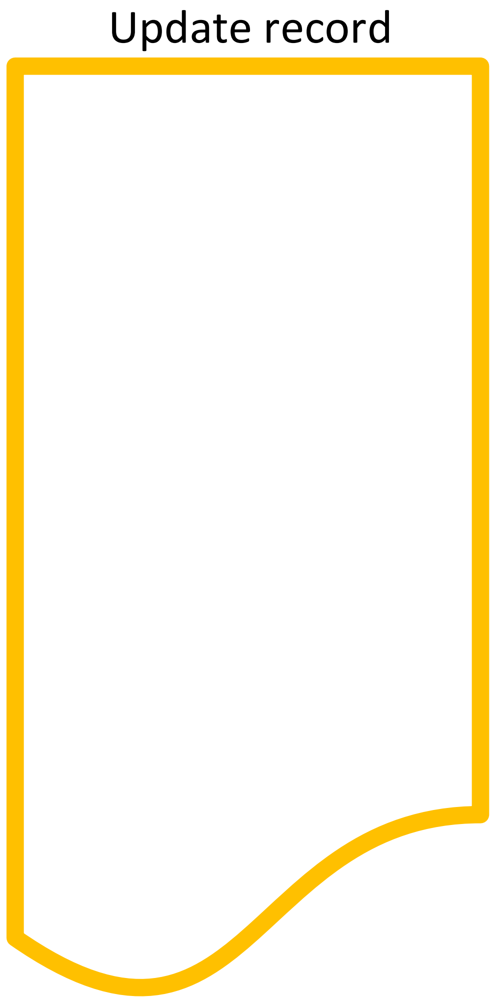




How our compiler works

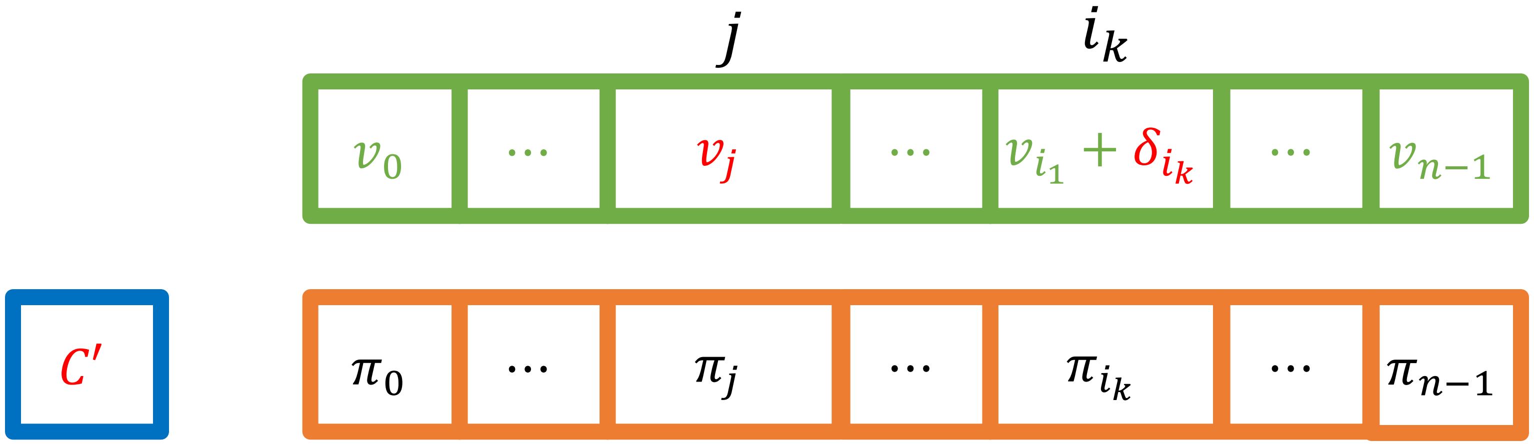
- When the number of records reaches $\sqrt{n}$, use $O(n \log n)$ time to open all proofs, and clear the record list

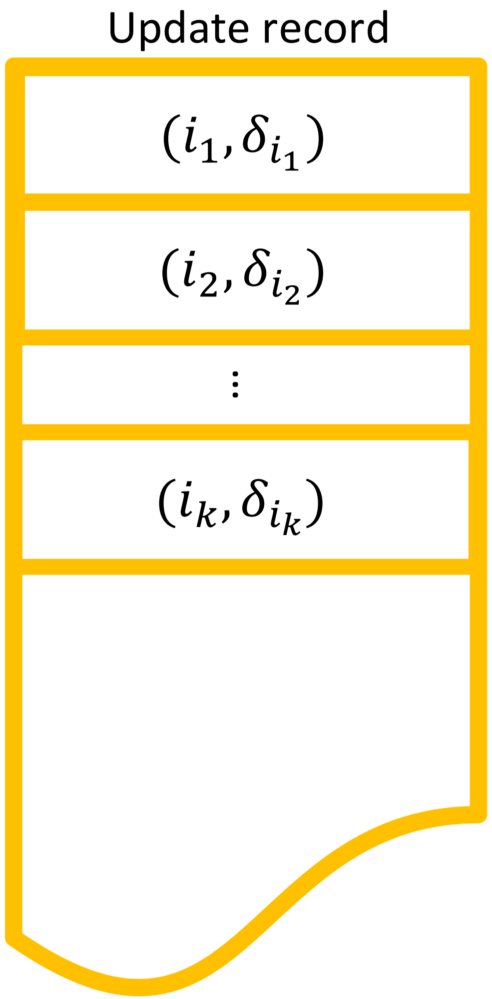




How our compiler works

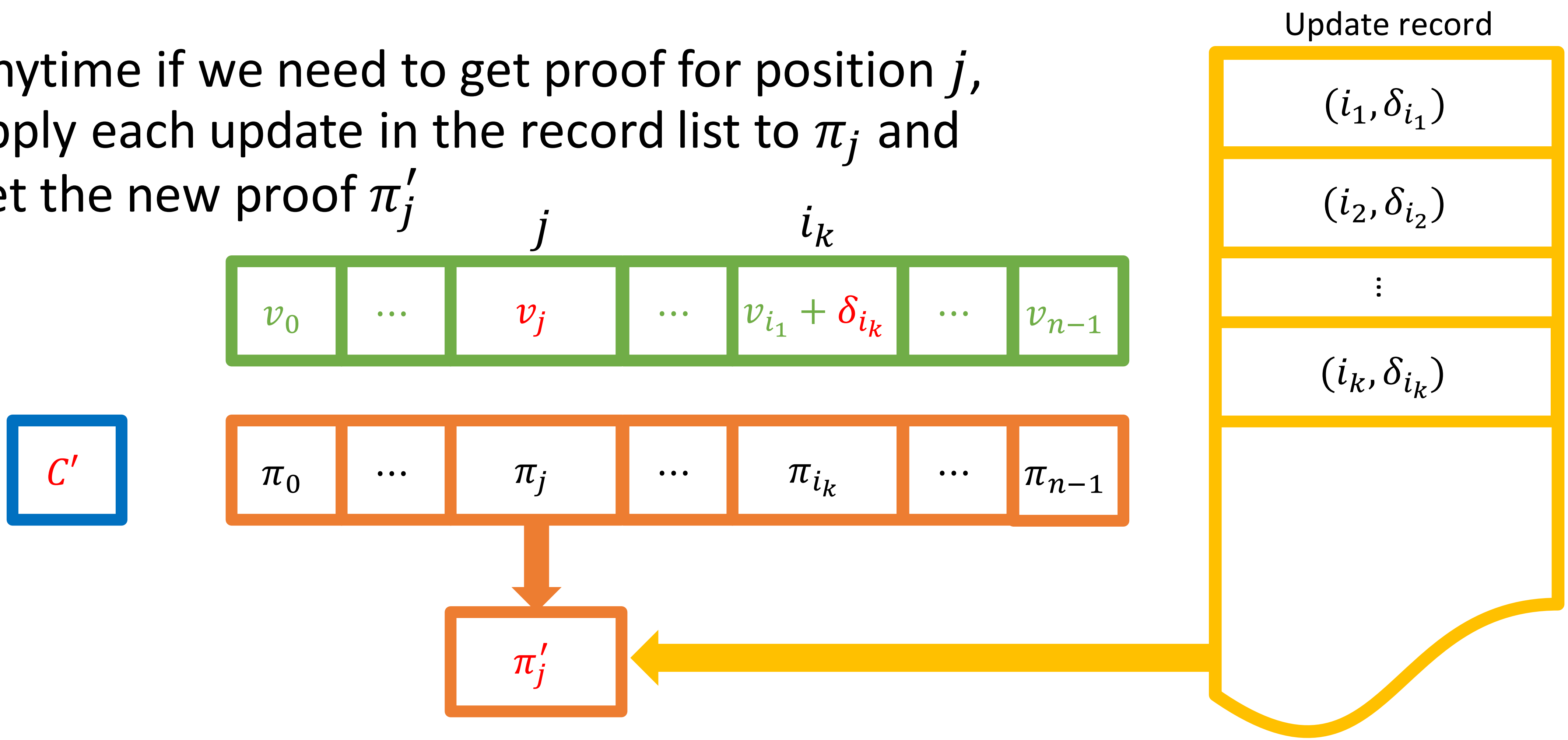
- Anytime if we need to get proof for position j,





How our compiler works

- Anytime if we need to get proof for position j, apply each update in the record list to π_j and get the new proof π_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

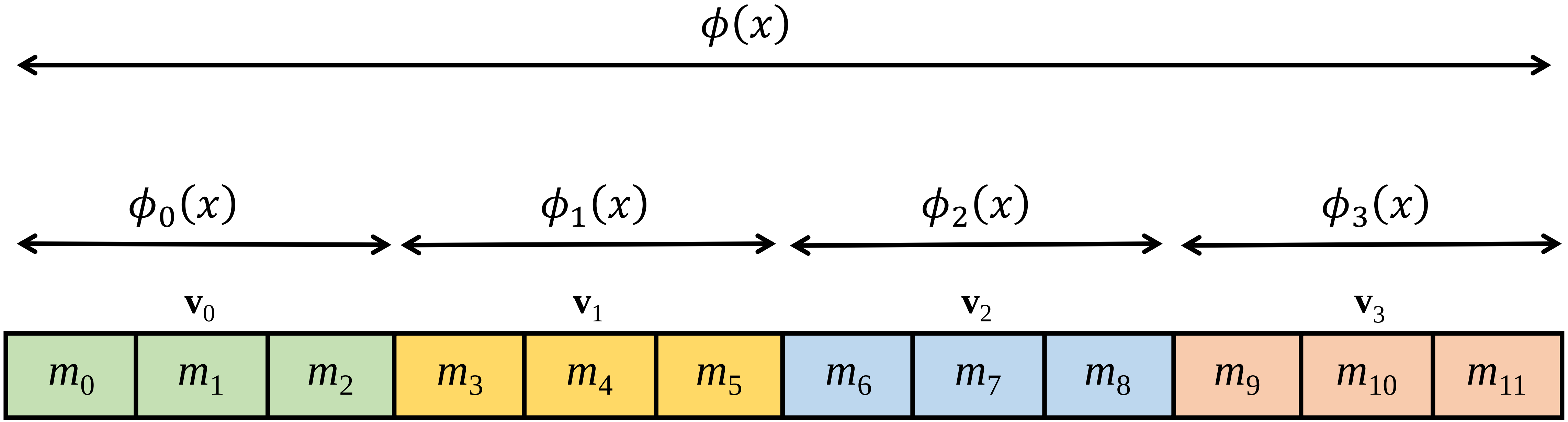
  - We need ~~amortized~~ $O\left(\frac{n \log n}{\sqrt{n}}\right) = O(\sqrt{n} \log n)$ time to do the update part
- Extend the size of update list to $2\sqrt{n}$
- When we have $\sqrt{n}$ records, separate the $O(n \log n)$ time computation in next $\sqrt{n}$ updates
- When we have $2\sqrt{n}$ records, clear the first $\sqrt{n}$ records in the list and start another $O(n \log n)$ time computation

Bucketing BalanceProofs

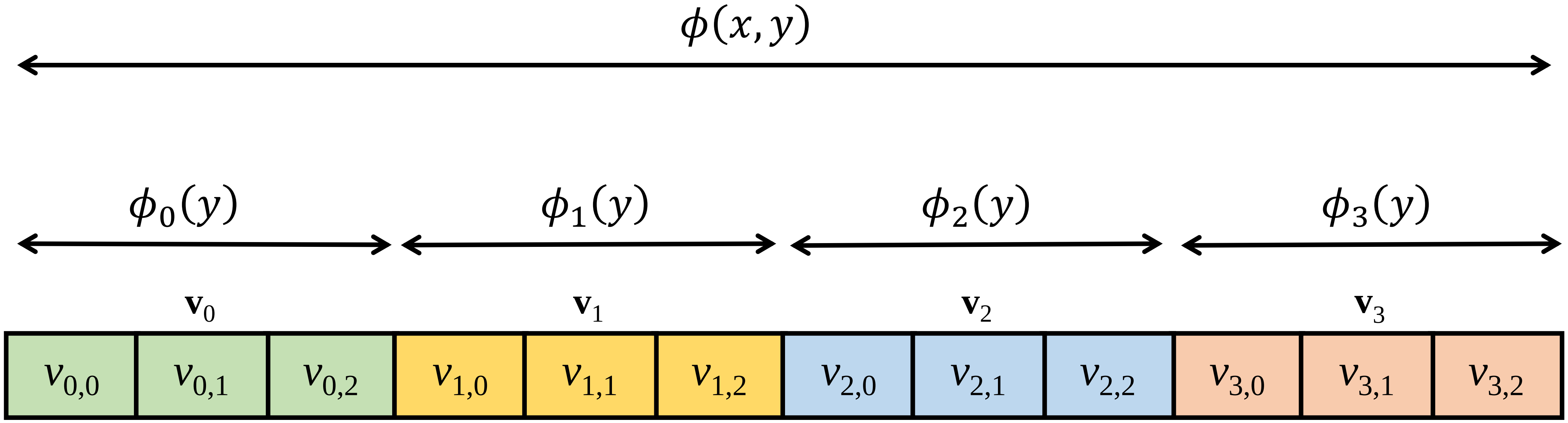
	$\mathbf{v}_0$			$\mathbf{v}_1$			$\mathbf{v}_2$			$\mathbf{v}_3$	
m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}	m_{11}

- Cut the vector into buckets
- Reduce the time of *UpdAll*
- Ensure that digest is still O(1) size

Basic Bucketing

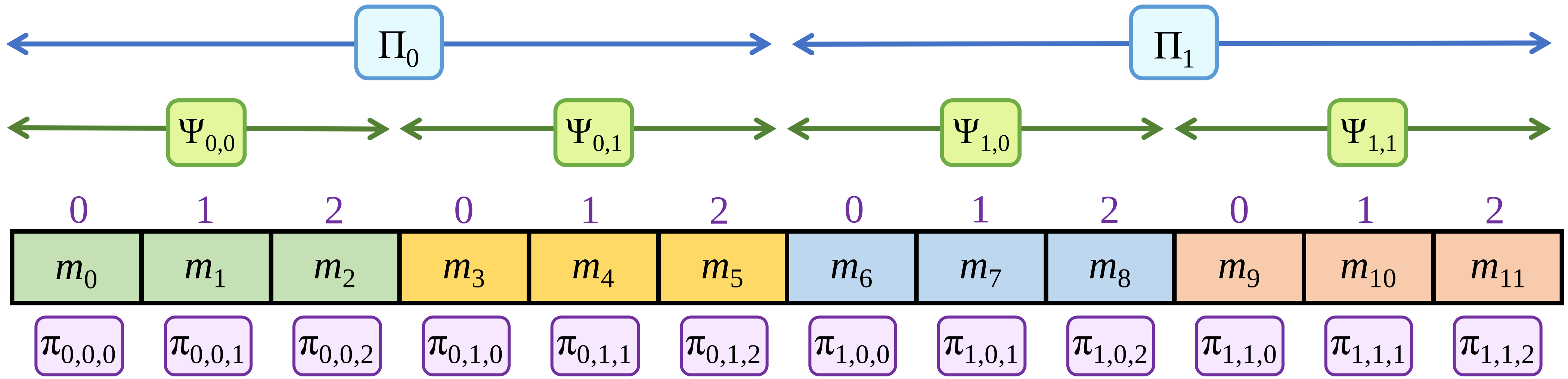


Space-efficient Bucketing



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

$n = 2^{20}$	Hyperproofs	aSVC	Basic compiler	Two-layer bucketing
UpdAll time	1.55 ms	98 s	3.03 s	4.60 ms
Batch proof size	51.6 KB	48 bytes		30~60 KB
Agg time	105 s	0.39 s		0.11 s
VrfyAgg time	12.9 s	0.43 s		0.20 s

$n = 2^{30}$	Hyperproofs	aSVC	Basic compiler	Two-layer bucketing
UpdAll time	2.58 ms	>20 hrs	136 s	19.0 ms
Batch proof size	51.6 KB	48 bytes		50~100 KB
Agg time	123 s	0.41 s		0.008 s
VrfyAgg time	17.4 s	0.44 s		0.11 s

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!