Polynomial IOPs for Memory Consistency Checks in Zero-Knowledge Virtual Machines

Yuncong Zhang[1], Shi-Feng Sun[1], Ren Zhang[2], Dawu Gu[1]

[1] Shanghai Jiao Tong University
[2] Nervos

December 7
Asiacrypt 2023

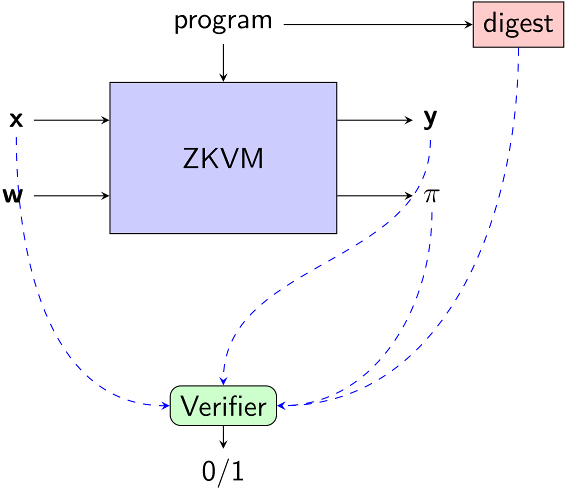
Outline

1. Background
2. Our Contribution
   - Formalizing Existing Constructions
   - Our New Method
3. Conclusion

Outline

1. Background
2. Our Contribution
   - Formalizing Existing Constructions
   - Our New Method
3. Conclusion

ZKVM



Applications of ZKVM

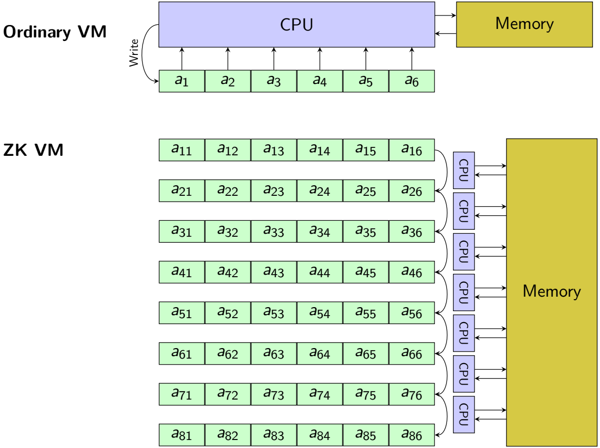
- **zkEVM**: execute EVM contracts with fewer gas and privacy
- **zkRollup**: promising scalability solutions to blockchain
- Privacy-related applications

How to Build a ZKVM

1. Unroll the intermediate values to execution trace

How to Build a ZKVM

1. Unroll the intermediate values to execution trace



How to Build a ZKVM

2. Design a Polynomial IOP (PIOP) to prove the trace satisfies some constraints

How to Build a ZKVM

2. Design a Polynomial IOP (PIOP) to prove the trace satisfies some constraints
    - The prover sends potentially large vectors (polynomials) to the verifier.

How to Build a ZKVM

2. Design a Polynomial IOP (PIOP) to prove the trace satisfies some constraints
   - The prover sends potentially large vectors (polynomials) to the verifier.
   - The verifier checks various relations over them, without reading the whole vectors.

How to Build a ZKVM

2. Design a Polynomial IOP (PIOP) to prove the trace satisfies some constraints
    - The prover sends potentially large vectors (polynomials) to the verifier.
    - The verifier checks various relations over them, without reading the whole vectors.
    - We have many ready-to-use building-blocks.

Prove Vector Relations in PIOP

Using PIOP, the verifier can verify

- **Vector equation**: $\mathbf{a} + \mathbf{b} \circ \mathbf{c} = \mathbf{0}$

Prove Vector Relations in PIOP

Using PIOP, the verifier can verify

- **Vector equation**: $\mathbf{a} + \mathbf{b} \circ \mathbf{c} = \mathbf{0}$
- **Permutation relation**: $\mathbf{a} \sim \mathbf{b}$, for example, $\mathbf{a} = (1, 2, 2, 3)$ and $\mathbf{b} = (2, 3, 1, 2)$

Prove Vector Relations in PIOP

Using PIOP, the verifier can verify

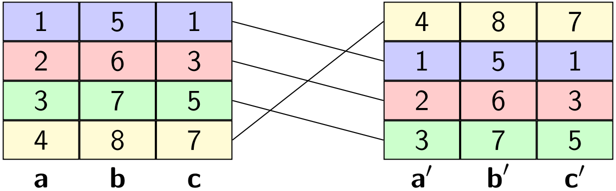
- **Vector equation**: $\mathbf{a} + \mathbf{b} \circ \mathbf{c} = \mathbf{0}$
- **Permutation relation**: $\mathbf{a} \sim \mathbf{b}$, for example, $\mathbf{a} = (1, 2, 2, 3)$ and $\mathbf{b} = (2, 3, 1, 2)$
- **Lookup relation**: $\mathbf{a} \subset \mathbf{b}$, for example, $\mathbf{a} = (1, 2, 2, 3)$ and $\mathbf{b} = (1, 2, 3, 4)$

Prove Vector Relations in PIOP

Using PIOP, the verifier can verify

- **Vector equation**: $\mathbf{a} + \mathbf{b} \circ \mathbf{c} = \mathbf{0}$
- **Permutation relation**: $\mathbf{a} \sim \mathbf{b}$, for example, $\mathbf{a} = (1, 2, 2, 3)$ and $\mathbf{b} = (2, 3, 1, 2)$
- **Lookup relation**: $\mathbf{a} \subset \mathbf{b}$, for example, $\mathbf{a} = (1, 2, 2, 3)$ and $\mathbf{b} = (1, 2, 3, 4)$

These relations can be extended to <u>tables</u>. For example, $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \sim (\mathbf{a}', \mathbf{b}', \mathbf{c}')$.



a	b	c
1	5	1
2	6	3
3	7	5
4	8	7

a′	b′	c′
4	8	7
1	5	1
2	6	3
3	7	5

ZKVM Constraints

The ZKVM constraints are very complex, so divide them into components

ZKVM Constraints

The ZKVM constraints are very complex, so divide them into components

- **Instruction fetching**

ZKVM Constraints

The ZKVM constraints are very complex, so divide them into components

- **Instruction fetching**
- **Arithmetic operation**

ZKVM Constraints

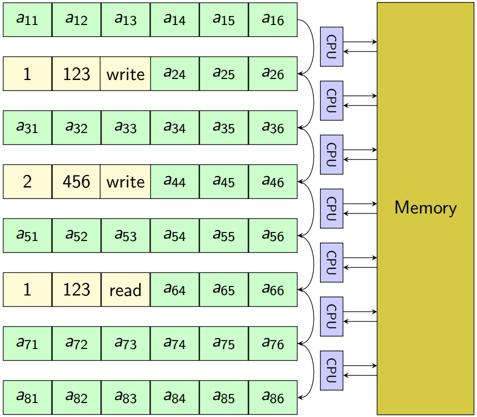
The ZKVM constraints are very complex, so divide them into components

- **Instruction fetching**
- **Arithmetic operation**
- **Memory**
- and so on

Memory Consistency Check

Show that if $a_{i3} = \text{write}$,
$a_{k3} \neq \text{write}$ or $a_{k1} \neq a_{i1}$ for $i < k < j$
$a_{j3} = \text{read}$ and $a_{j1} = a_{i1}$,
then $a_{j2} = a_{i2}$

ZK VM



addr	val	op			
a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}
1	123	write	a_{24}	a_{25}	a_{26}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}
2	456	write	a_{44}	a_{45}	a_{46}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}
1	123	read	a_{64}	a_{65}	a_{66}
a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}
a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}

Current Status of Memory Consistency Check

The memory is a history-dependent component:

Current Status of Memory Consistency Check

The memory is a history-dependent component:

- One row depends potentially on any previous row.

Current Status of Memory Consistency Check

The memory is a history-dependent component:

- One row depends potentially on any previous row.
- Difficult to capture the memory constraint by simple vector equations.

Current Status of Memory Consistency Check

The memory is a history-dependent component:

- One row depends potentially on any previous row.
- Difficult to capture the memory constraint by simple vector equations.

There are solutions developed in the live ZKVM projects in industry, but

Current Status of Memory Consistency Check

The memory is a history-dependent component:

- One row depends potentially on any previous row.
- Difficult to capture the memory constraint by simple vector equations.

There are solutions developed in the live ZKVM projects in industry, but

- guided by intuition, and never formalized or analyzed.

Current Status of Memory Consistency Check

The memory is a history-dependent component:

- One row depends potentially on any previous row.
- Difficult to capture the memory constraint by simple vector equations.

There are solutions developed in the live ZKVM projects in industry, but

- guided by intuition, and never formalized or analyzed.
- relatively expensive.

Our Contribution

This work is the first academic treatment on this component.

Our Contribution

This work is the first academic treatment on this component.

- Formalize the constructions in the industry, and prove their security formally.

Our Contribution

This work is the first academic treatment on this component.

- Formalize the constructions in the industry, and prove their security formally.
- Reveal new direction for potentially better constructions.

Outline

1. Background

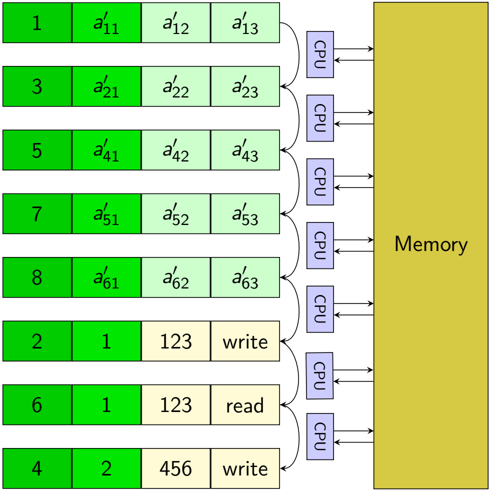
2. Our Contribution
    - Formalizing Existing Constructions
    - Our New Method

3. Conclusion

The Sorting Technique

ZK VM

Sort by: address, cycle



1	a'_{11}	a'_{12}	a'_{13}
3	a'_{21}	a'_{22}	a'_{23}
5	a'_{41}	a'_{42}	a'_{43}
7	a'_{51}	a'_{52}	a'_{53}
8	a'_{61}	a'_{62}	a'_{63}
2	1	123	write
6	1	123	read
4	2	456	write

The Sorting Paradigm

We propose sorting paradigm that captures all existing constructions

The Sorting Paradigm

We propose sorting paradigm that captures all existing constructions

1. The prover sorts $\textbf{cycle}, \textbf{op}, \textbf{addr}, \textbf{val}$ by $(\textbf{addr}, \textbf{cycle})$ to obtain $\widetilde{\textbf{cycle}}, \widetilde{\textbf{op}}, \widetilde{\textbf{addr}}, \widetilde{\textbf{val}}$.

The Sorting Paradigm

We propose sorting paradigm that captures all existing constructions

1. The prover sorts $\mathbf{cycle}, \mathbf{op}, \mathbf{addr}, \mathbf{val}$ by $(\mathbf{addr}, \mathbf{cycle})$ to obtain $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$.
2. The verifier checks memory consistency of sorted trace.

$$\mathbf{a} \circ \widetilde{\mathbf{op}}^{\leftarrow 1} + \mathbf{b} \circ (\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}) - (\widetilde{\mathbf{val}}^{\leftarrow 1} - \widetilde{\mathbf{val}}) = \mathbf{0}$$

The Sorting Paradigm

We propose sorting paradigm that captures all existing constructions

1. The prover sorts $\mathbf{cycle}, \mathbf{op}, \mathbf{addr}, \mathbf{val}$ by $(\mathbf{addr}, \mathbf{cycle})$ to obtain $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$.
2. The verifier checks memory consistency of sorted trace.

$$\mathbf{a} \circ \widetilde{\mathbf{op}}^{\leftarrow 1} + \mathbf{b} \circ (\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}) - (\widetilde{\mathbf{val}}^{\leftarrow 1} - \widetilde{\mathbf{val}}) = \mathbf{0}$$

3. The verifier checks that $\mathbf{cycle}, \mathbf{op}, \mathbf{addr}, \mathbf{val}$ and $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$ are permutations of each other.

The Sorting Paradigm

We propose sorting paradigm that captures all existing constructions

1. The prover sorts $\mathbf{cycle}, \mathbf{op}, \mathbf{addr}, \mathbf{val}$ by $(\mathbf{addr}, \mathbf{cycle})$ to obtain $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$.
2. The verifier checks memory consistency of sorted trace.

$$\mathbf{a} \circ \widetilde{\mathbf{op}}^{\leftarrow 1} + \mathbf{b} \circ (\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}) - (\widetilde{\mathbf{val}}^{\leftarrow 1} - \widetilde{\mathbf{val}}) = \mathbf{0}$$

3. The verifier checks that $\mathbf{cycle}, \mathbf{op}, \mathbf{addr}, \mathbf{val}$ and $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$ are permutations of each other.
4. The final step:

The Sorting Paradigm

We propose sorting paradigm that captures all existing constructions

1. The prover sorts $\mathbf{cycle}, \mathbf{op}, \mathbf{addr}, \mathbf{val}$ by $(\mathbf{addr}, \mathbf{cycle})$ to obtain $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$.
2. The verifier checks memory consistency of sorted trace.

$$\mathbf{a} \circ \widetilde{\mathbf{op}}^{\leftarrow 1} + \mathbf{b} \circ (\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}) - (\widetilde{\mathbf{val}}^{\leftarrow 1} - \widetilde{\mathbf{val}}) = \mathbf{0}$$

3. The verifier checks that $\mathbf{cycle}, \mathbf{op}, \mathbf{addr}, \mathbf{val}$ and $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$ are permutations of each other.
4. The final step:
    - Checks that $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$ is sorted by $(\mathbf{addr}, \mathbf{cycle})$

The Sorting Paradigm

We propose sorting paradigm that captures all existing constructions

1. The prover sorts $\mathbf{cycle}, \mathbf{op}, \mathbf{addr}, \mathbf{val}$ by $(\mathbf{addr}, \mathbf{cycle})$ to obtain $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$.
2. The verifier checks memory consistency of sorted trace.

$$\mathbf{a} \circ \widetilde{\mathbf{op}}^{\leftarrow 1} + \mathbf{b} \circ (\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}) - (\widetilde{\mathbf{val}}^{\leftarrow 1} - \widetilde{\mathbf{val}}) = \mathbf{0}$$

3. The verifier checks that $\mathbf{cycle}, \mathbf{op}, \mathbf{addr}, \mathbf{val}$ and $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$ are permutations of each other.
4. The final step:
    - Checks that $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$ is sorted by $(\mathbf{addr}, \mathbf{cycle})$
    - most expensive

The Sorting Paradigm

We propose sorting paradigm that captures all existing constructions

1. The prover sorts $\mathbf{cycle}, \mathbf{op}, \mathbf{addr}, \mathbf{val}$ by $(\mathbf{addr}, \mathbf{cycle})$ to obtain $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$.
2. The verifier checks memory consistency of sorted trace.

$$\mathbf{a} \circ \widetilde{\mathbf{op}}^{\leftarrow 1} + \mathbf{b} \circ (\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}) - (\widetilde{\mathbf{val}}^{\leftarrow 1} - \widetilde{\mathbf{val}}) = \mathbf{0}$$

3. The verifier checks that $\mathbf{cycle}, \mathbf{op}, \mathbf{addr}, \mathbf{val}$ and $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$ are permutations of each other.
4. The final step:
   - Checks that $\widetilde{\mathbf{cycle}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$ is sorted by $(\mathbf{addr}, \mathbf{cycle})$
   - most expensive
   - varies in different ZKVMs

Contiguous Read-Only Memory

The simplest case, adopted by Cairo

Contiguous Read-Only Memory

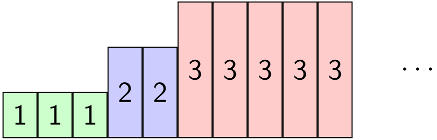
The simplest case, adopted by Cairo

- The addresses form a contiguous subset of $\mathbb{F}$

Contiguous Read-Only Memory

The simplest case, adopted by Cairo

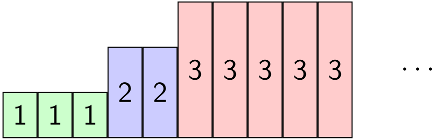
- The addresses form a contiguous subset of $\mathbb{F}$
- The sorted address becomes



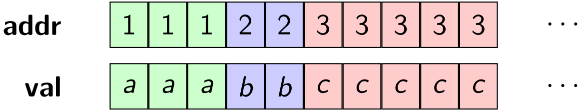
Contiguous Read-Only Memory

The simplest case, adopted by Cairo

- The addresses form a contiguous subset of $\mathbb{F}$
- The sorted address becomes



- No need for **op**, $\widetilde{\textbf{op}}$, **cycle** and $\widetilde{\textbf{cycle}}$

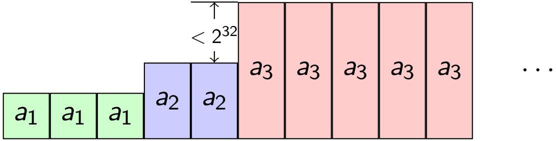


addr	1	1	1	2	2	3	3	3	3	3	. . .
val	a	a	a	b	b	c	c	c	c	c	. . .

Read-Write Memory with 32-bit Space

The most widely used case, and the most expensive

- The addresses fall in $[0..2^{32}-1]$
- Adjacent addresses in the sorted trace differ by $[0..2^{32}-1]$



- Range check over the vector $\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}$

Can be extended to 256-bit, but very expensive

Read-Write Memory with Full Address Space

Efficiently support full address space.

- First proposed and only adopted in Triton VM
- The entire finite field $\mathbb{F}$ as the address space

Read-Write Memory with Full Address Space

Efficiently support full address space.

- First proposed and only adopted in Triton VM
- The entire finite field $\mathbb{F}$ as the address space
- Usually $|\mathbb{F}| \approx 2^{256}$
  - Even $|\mathbb{F}| \approx 2^{64}$ is usually sufficient.
  - Can be extended to $\mathbb{F}^c$ by random-linear-combination.

Read-Write Memory with Full Address Space

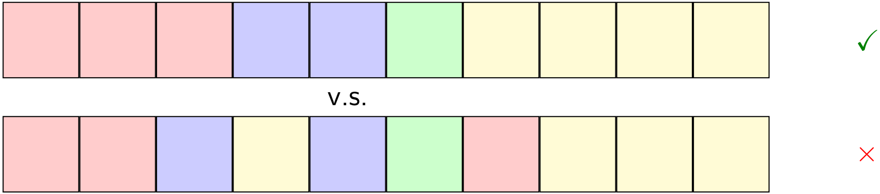
Efficiently support full address space.

- First proposed and only adopted in Triton VM
- The entire finite field $\mathbb{F}$ as the address space
- Usually $|\mathbb{F}| \approx 2^{256}$
  - Even $|\mathbb{F}| \approx 2^{64}$ is usually sufficient.
  - Can be extended to $\mathbb{F}^c$ by random-linear-combination.
- Challenge: hard to check address column is sorted.

Read-Write Memory with Full Address Space

Efficiently support full address space.

- First proposed and only adopted in Triton VM
- The entire finite field $\mathbb{F}$ as the address space
- Usually $|\mathbb{F}| \approx 2^{256}$
  - Even $|\mathbb{F}| \approx 2^{64}$ is usually sufficient.
  - Can be extended to $\mathbb{F}^c$ by random-linear-combination.
- Challenge: hard to check address column is sorted.
- The basic idea: contiguity check



Comparison of Existing Protocols

	Writable	Address Space	Efficiency
CROM	×	Contiguous	Efficient
32-bit RW	✓	$[0..2^{32}-1]$	2 Lookups
Full Address RW	✓	$\mathbb{F}$	1 Lookup

Address Cycle Method

Better way to eliminate the history-dependency

Address Cycle Method

Better way to eliminate the history-dependency

- **Basic idea**: making the identical addresses adjacent

Address Cycle Method

Better way to eliminate the history-dependency

- **Basic idea**: making the identical addresses adjacent
- **Sorting paradigm**: reorder the trace

Address Cycle Method

Better way to eliminate the history-dependency

- **Basic idea**: making the identical addresses adjacent
- **Sorting paradigm**: reorder the trace
    - Heavy: send a sorted copy for every column.

Address Cycle Method

Better way to eliminate the history-dependency

- **Basic idea**: making the identical addresses adjacent
- **Sorting paradigm**: reorder the trace
  - Heavy: send a sorted copy for every column.
- **Address cycle method**: redefine the adjacency

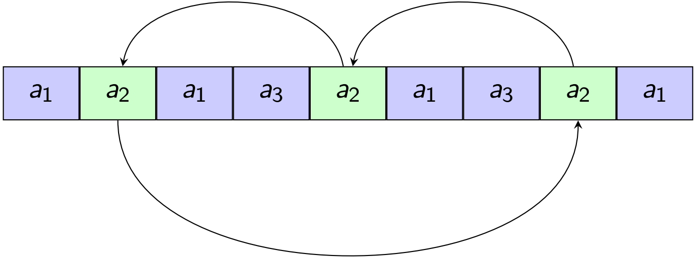
Address Cycle Method

Better way to eliminate the history-dependency

- **Basic idea**: making the identical addresses adjacent
- **Sorting paradigm**: reorder the trace
    - Heavy: send a sorted copy for every column.
- **Address cycle method**: redefine the adjacency
    - Lightweight: send a single vector representing a permutation.

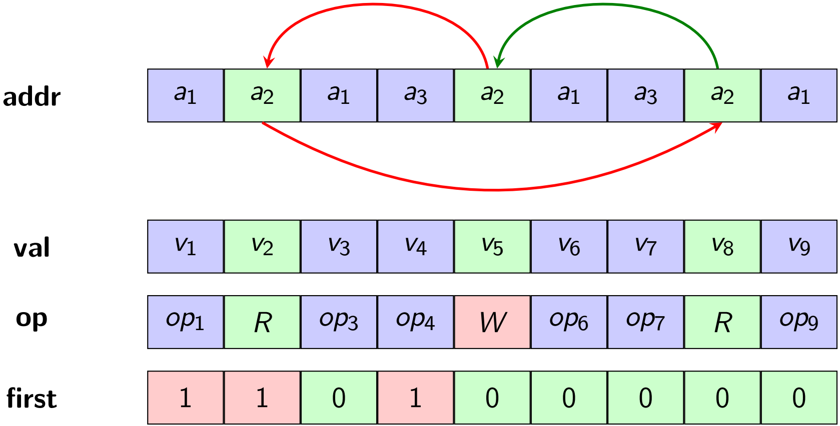
Address Cycle Method

Permutation σ



The **first** Vector

Permutation σ



addr	a_1	a_2	a_1	a_3	a_2	a_1	a_3	a_2	a_1
val	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
op	op_1	R	op_3	op_4	W	op_6	op_7	R	op_9
first	1	1	0	1	0	0	0	0	0

Permem

- **One vector equation**: $(\sigma(\mathbf{val}) - \mathbf{val}) \circ (\mathbf{op} - \mathbf{Write}) \circ (\mathbf{1} - \mathbf{first}) \overset{?}{=} \mathbf{0}$

Permem

- **One vector equation**: $(\sigma(\mathbf{val}) - \mathbf{val}) \circ (\mathbf{op} - \mathbf{Write}) \circ (\mathbf{1} - \mathbf{first}) \overset{?}{=} \mathbf{0}$
- **One permutation check**: $(\mathbf{addr}, \mathbf{cycle}, \mathbf{val}) \sim (\mathbf{addr}, \sigma(\mathbf{cycle}), \sigma(\mathbf{val}))$

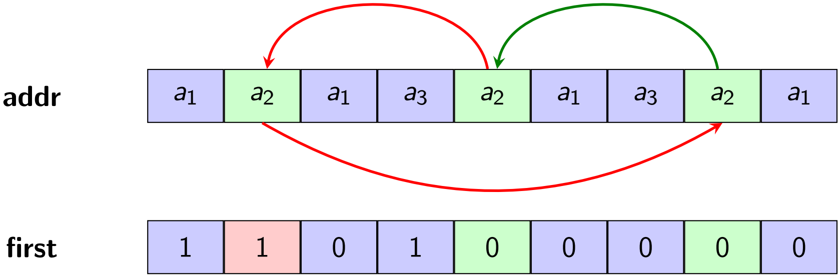
Permem

- **One vector equation**: $(\sigma(\textbf{val}) - \textbf{val}) \circ (\textbf{op} - \textbf{Write}) \circ (\textbf{1} - \textbf{first}) \overset{?}{=} \textbf{0}$
- **One permutation check**: $(\textbf{addr}, \textbf{cycle}, \textbf{val}) \sim (\textbf{addr}, \sigma(\textbf{cycle}), \sigma(\textbf{val}))$
- **The challenge**: Check σ, **first** are consistent with **addr**

Check σ and **first** Vector

The correctness of σ, **first** is guaranteed by

- **One lookup argument**: $\sigma(t) \geq t$ only at **first**
- **Distinctness check**: **addr** are distinct at **first**



addr	a_1	a_2	a_1	a_3	a_2	a_1	a_3	a_2	a_1
first	1	1	0	1	0	0	0	0	0

Distinctness Check

Generalization of the contiguity check

Distinctness Check

Generalization of the contiguity check

- Let $f(X) := \prod_{first_t = 1}(X - addr_t)$

Distinctness Check

Generalization of the contiguity check

- Let $f(X) := \prod_{first_t = 1}(X - addr_t)$
- Compute $g(X) = \frac{df(X)}{dX}$

Distinctness Check

Generalization of the contiguity check

- Let $f(X) := \prod_{first_t=1}(X - addr_t)$
- Compute $g(X) = \frac{df(X)}{dX}$
- Prover sends $s(X), t(X), g(X)$ to verifier such that $s(X)f(X) + t(X)g(X) = 1$

Distinctness Check

Generalization of the contiguity check

- Let $f(X) := \prod_{first_t=1}(X - addr_t)$
- Compute $g(X) = \frac{df(X)}{dX}$
- Prover sends $s(X), t(X), g(X)$ to verifier such that $s(X)f(X) + t(X)g(X) = 1$
- The verifier checks $g(X)$ and $s(z)f(z) + t(z)g(z) = 1$

Outline

1. Background
2. Our Contribution
   - Formalizing Existing Constructions
   - Our New Method
3. Conclusion

Comparison

Protocol	**Perm.**	**Lookup**	**Poly.**	**Queries**	$\mathcal{A}$	**Writable**
Contiguous ROM	1	0	4	0	Contiguous	No
($32k$)-bit RAM	1	$2k$	$7+2k$	0	$[0..2^{32k}-1]$	Yes
Full address RAM	1	1	$10+c$	2	$\mathbb{F}^c$	Yes
Our Permem	1	1	$6+c$	2	$\mathbb{F}^c$	Yes

Q&A

https://eprint.iacr.org/2023/1555