Two Shuffles Make a RAM: Improved Constant Overhead Zero Knowledge RAM

Yibin Yang, Georgia Tech
David Heath, UIUC

Zero-Knowledge Proof [GMR85]

Zero-Knowledge Proof [GMR85]

Verifier

Prover

Zero-Knowledge Proof [GMR85]

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Statement

Verifier

Prover

Zero-Knowledge Proof [GMR85]

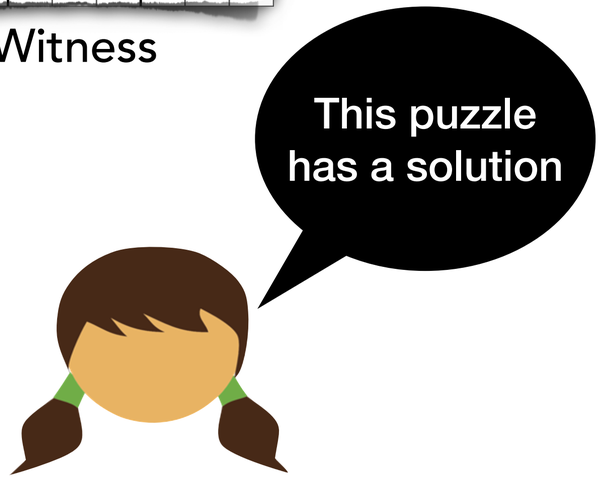
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Statement

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Witness

V



P

Zero-Knowledge Proof [GMR85]

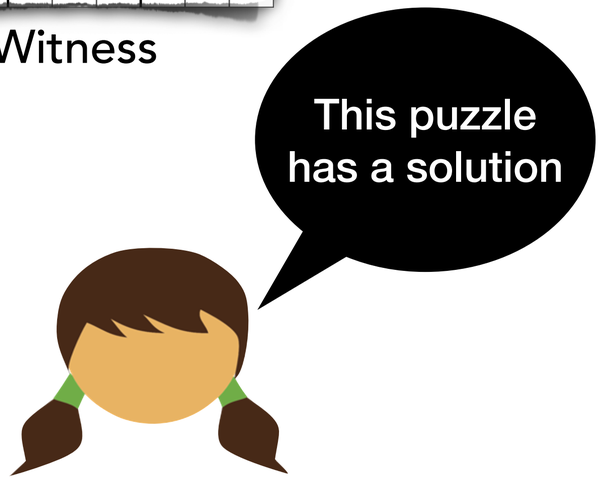
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Statement

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Witness

V



P

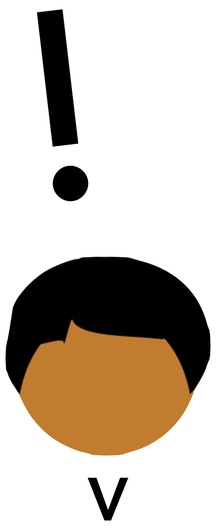
Zero-Knowledge Proof [GMR85]

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

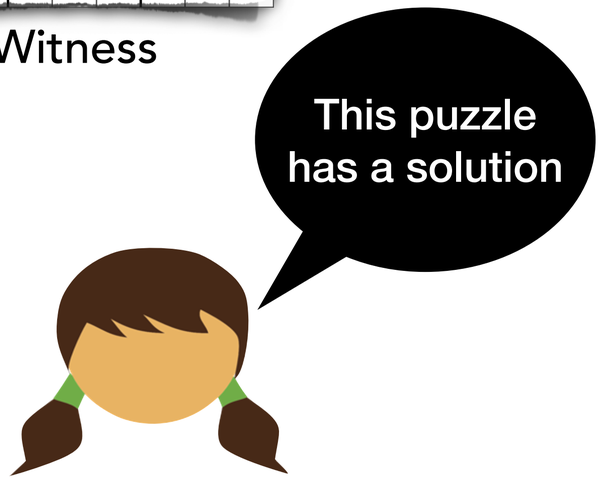
Statement

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Witness

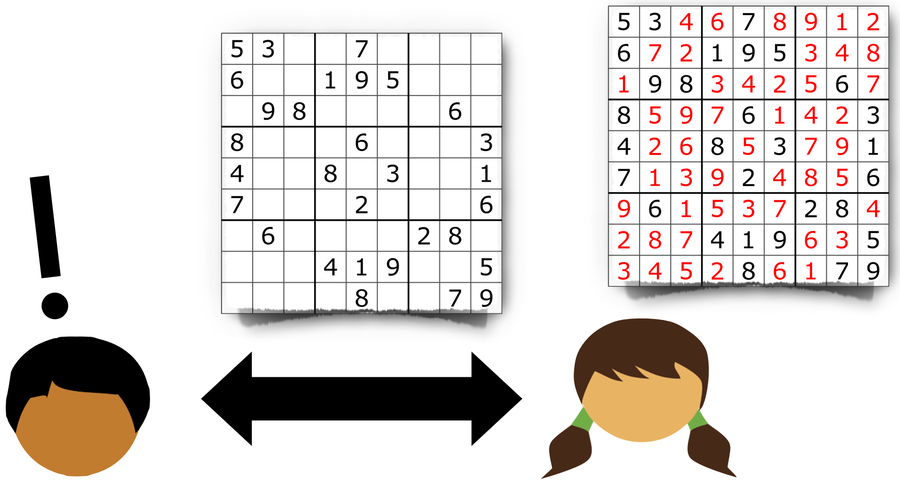


V



P

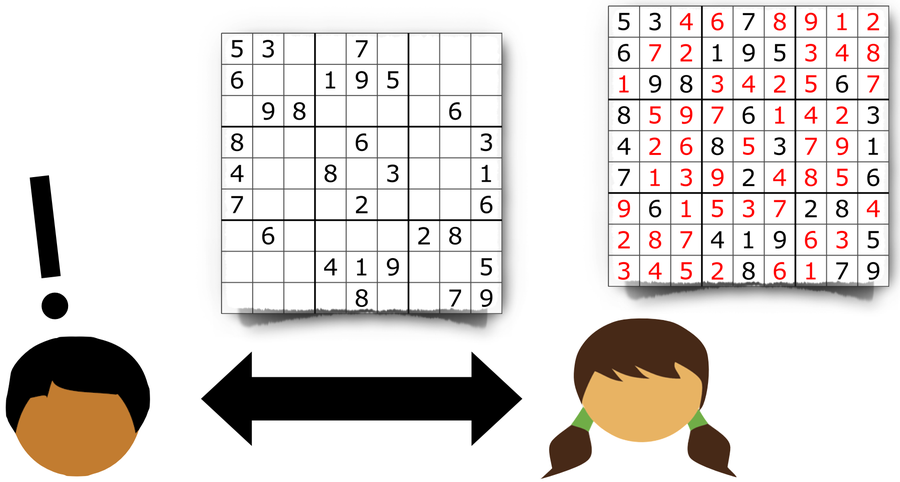
Zero-Knowledge Proof [GMR85]



Completeness

V always accepts valid statements

Zero-Knowledge Proof [GMR85]



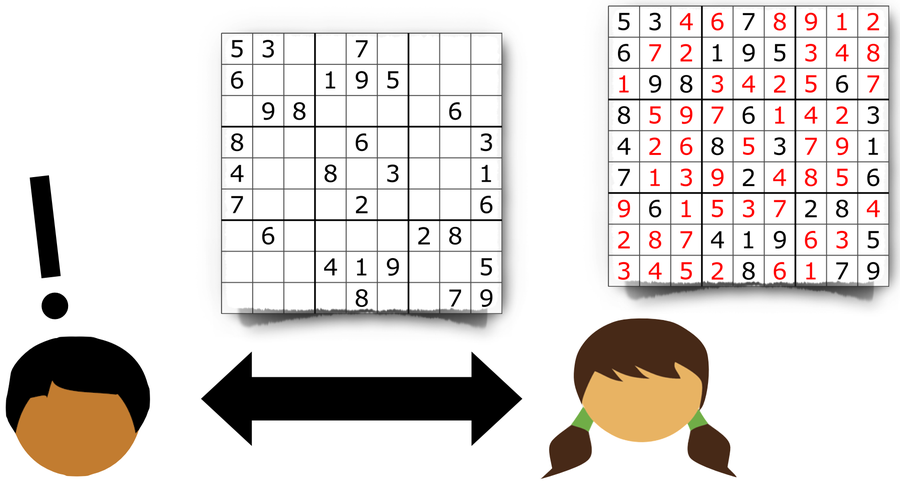
Completeness

V always accepts valid statements

Soundness

If P does not have a witness, V rejects statements with high probability

Zero-Knowledge Proof [GMR85]



Completeness

V always accepts valid statements

Soundness

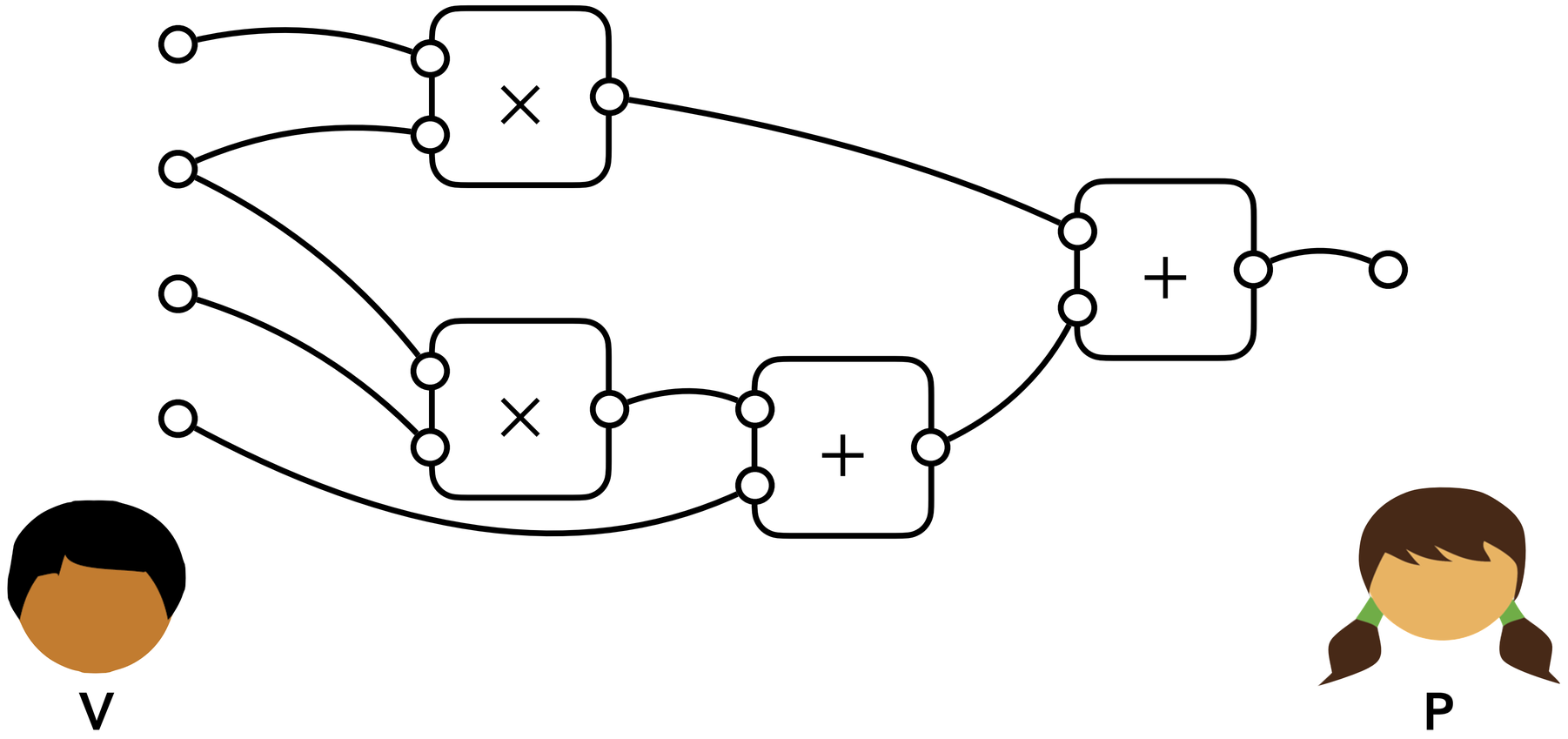
If P does not have a witness, V rejects statements with high probability

Zero Knowledge

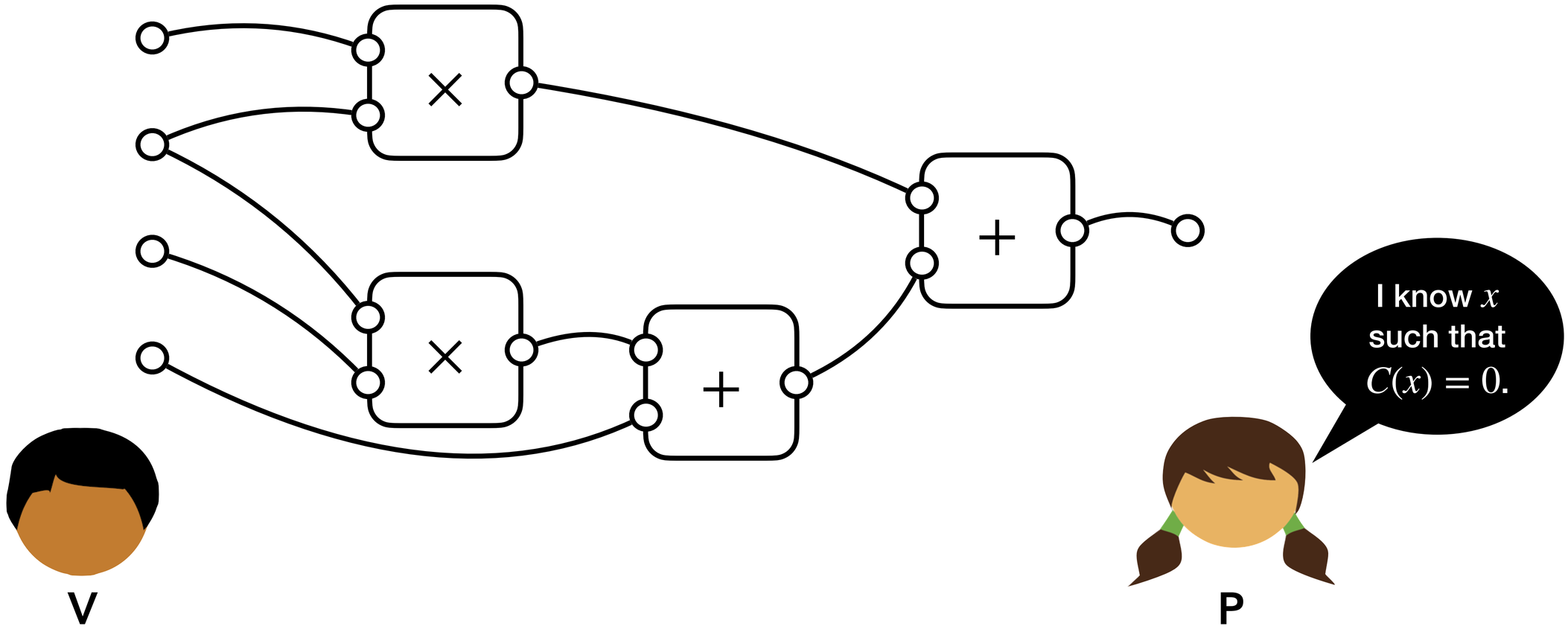
V learns nothing except that the statement is valid

Generic Zero-Knowledge Proof

Generic Zero-Knowledge Proof

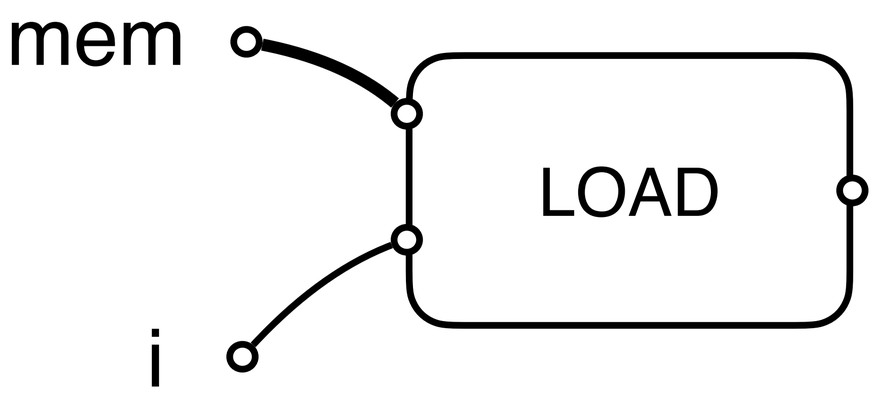


Generic Zero-Knowledge Proof

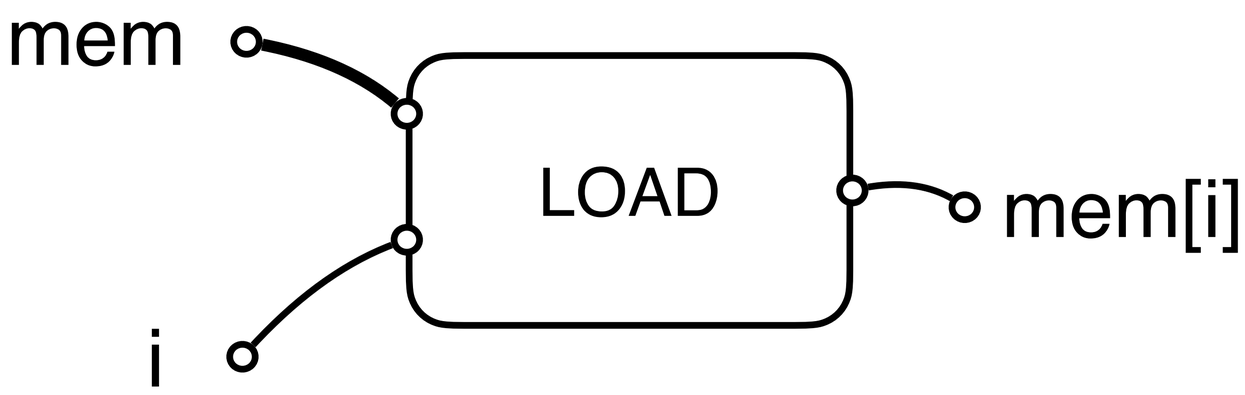


Problem: ZK RAM

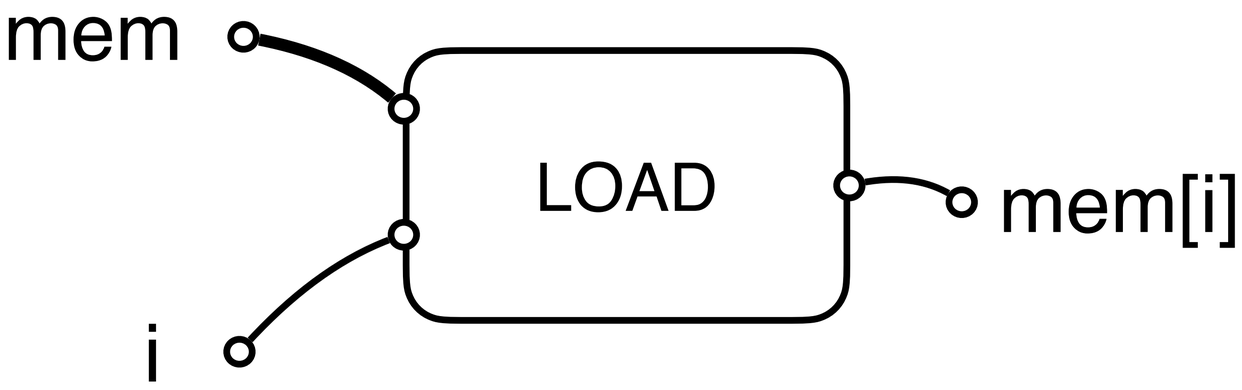
Problem: ZK RAM

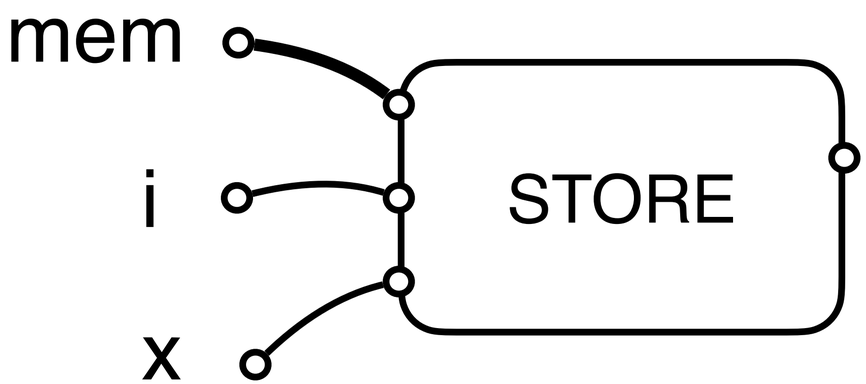


Problem: ZK RAM

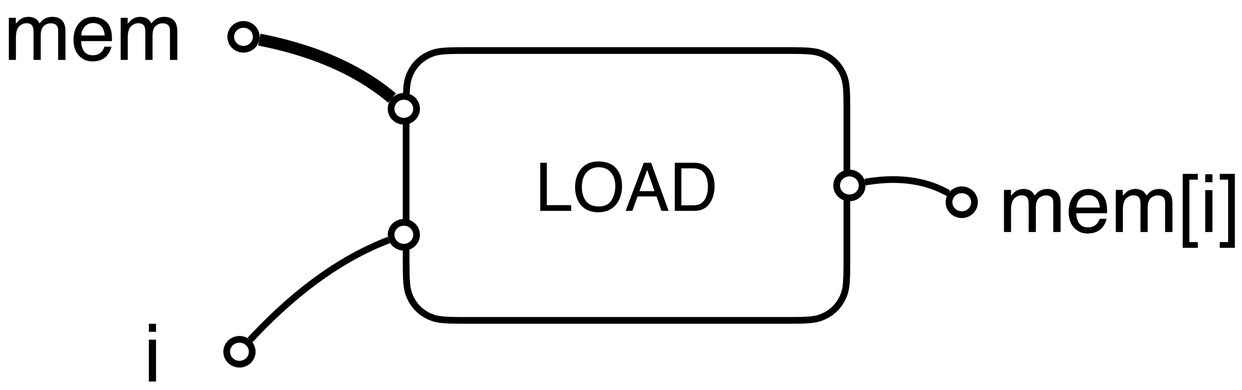


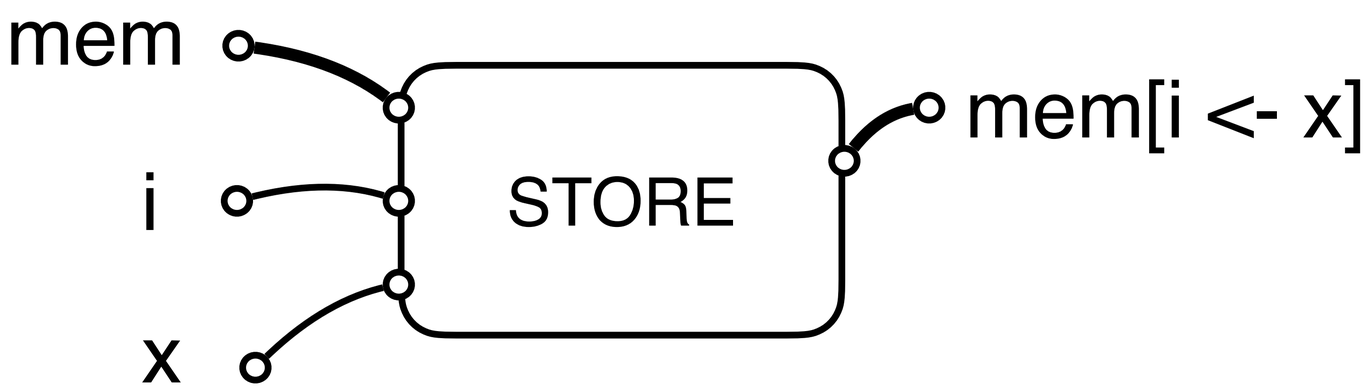
Problem: ZK RAM



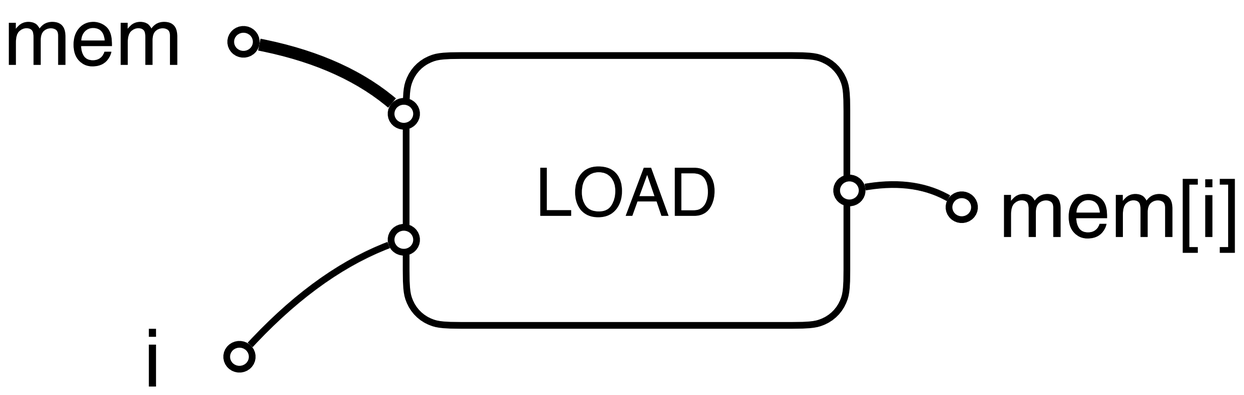


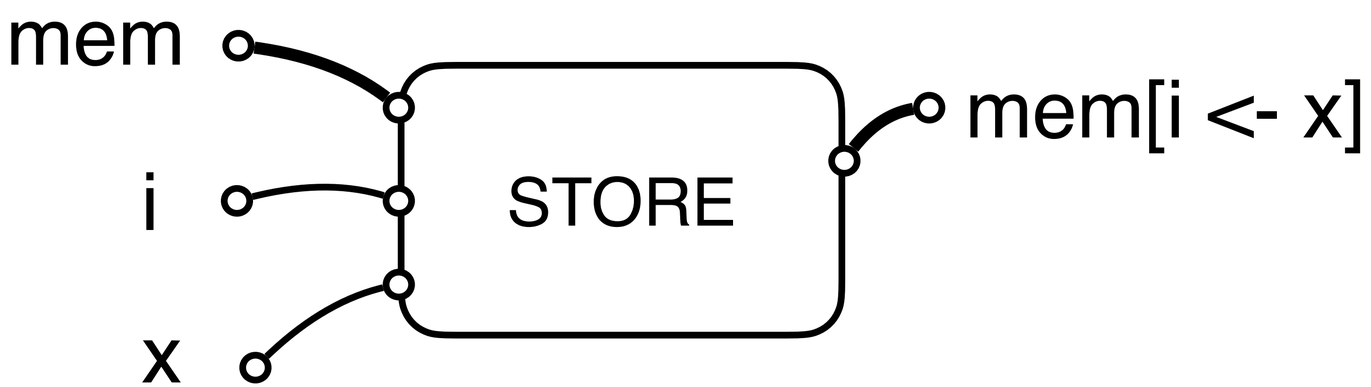
Problem: ZK RAM

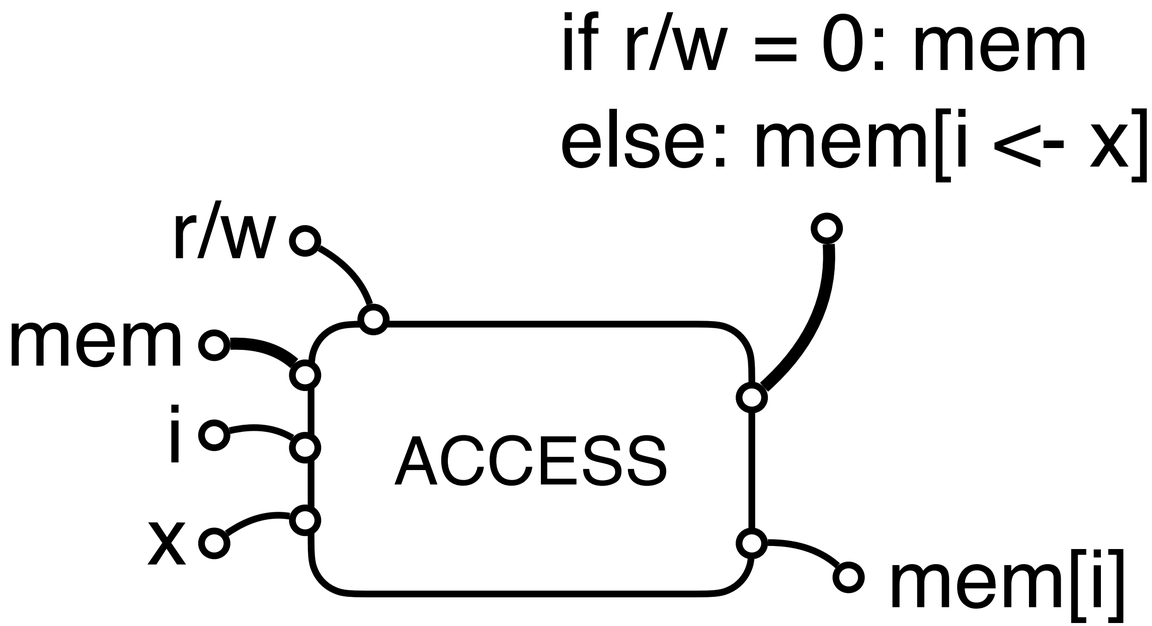




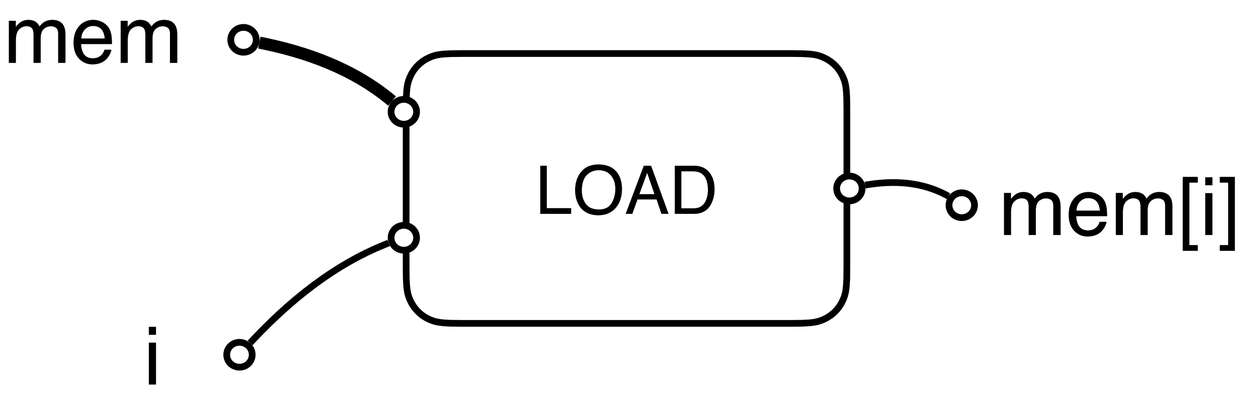
Problem: ZK RAM

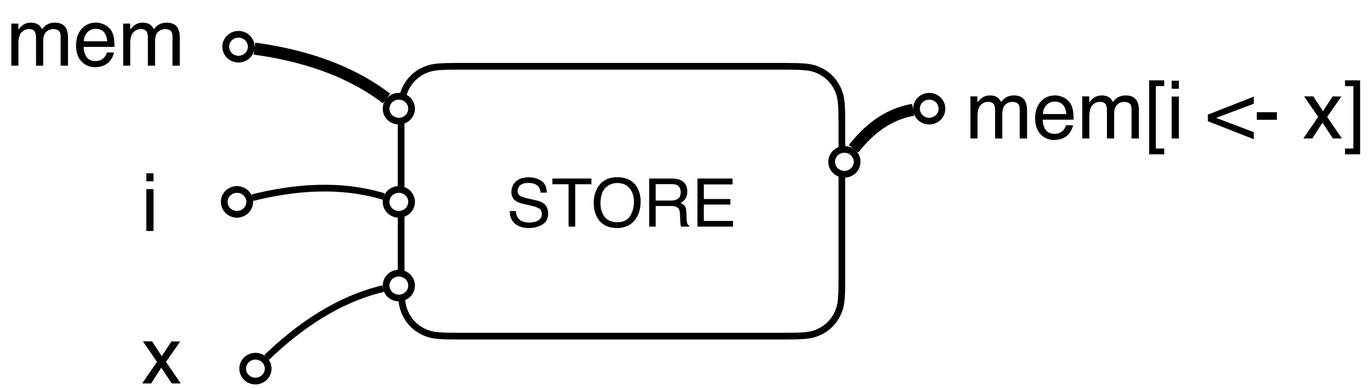


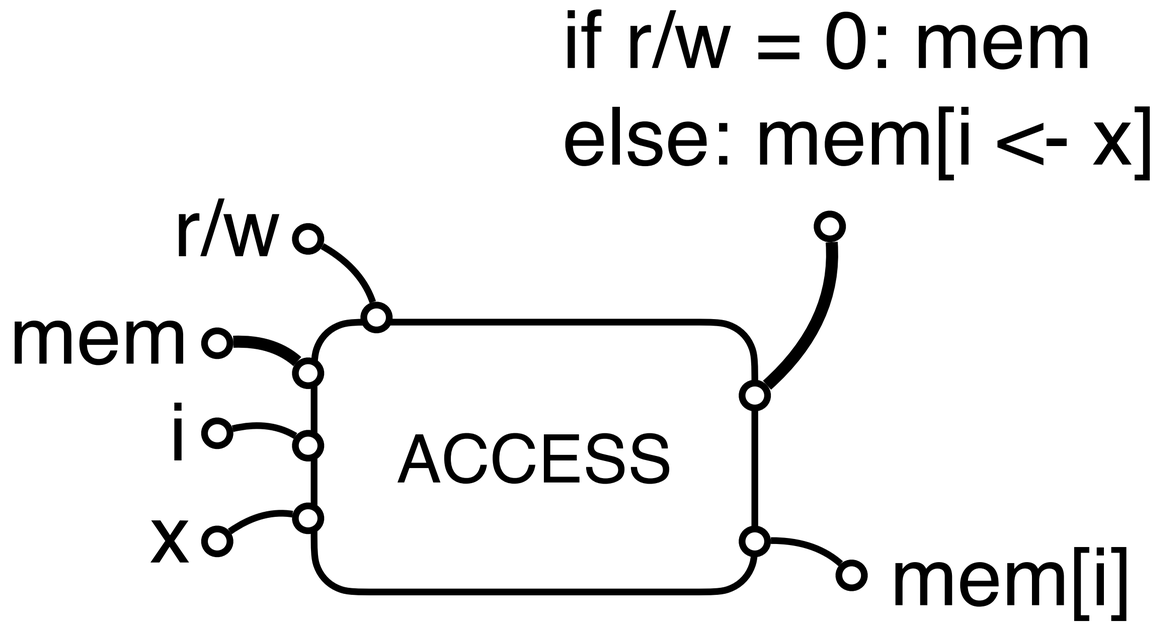




Problem: ZK RAM







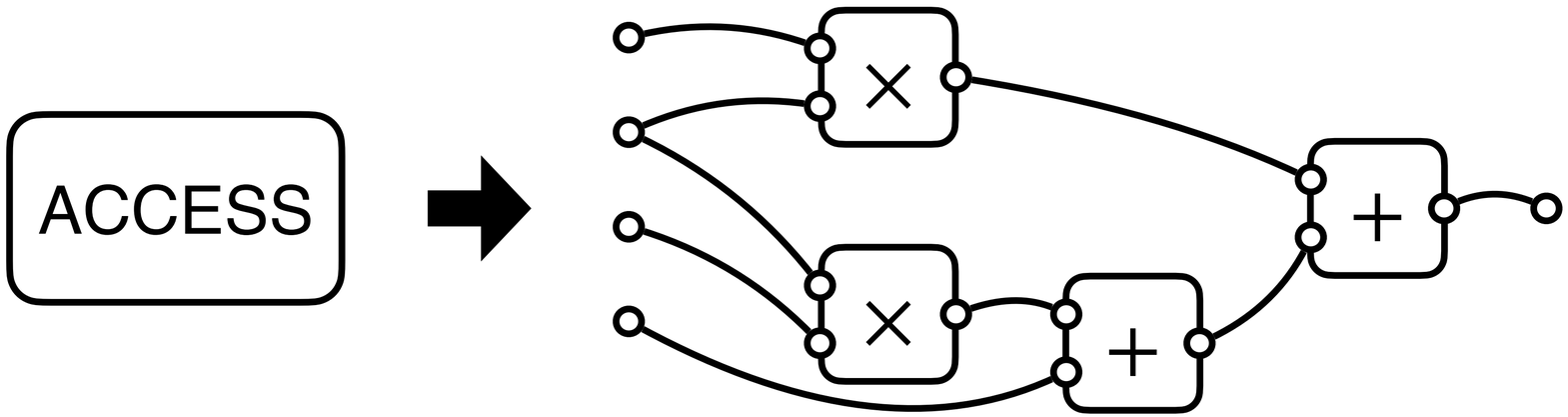
Our ZK RAM in Recent Works

ZK CPU [YHHKV, CCS '24]

ZK ML [HCLWZYZ, USENIX Security '24]

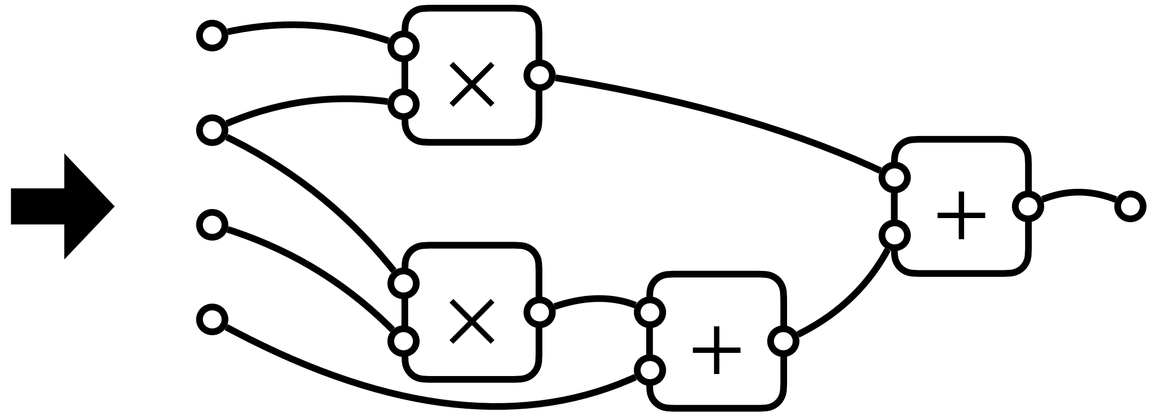
Terminology

Terminology



Terminology

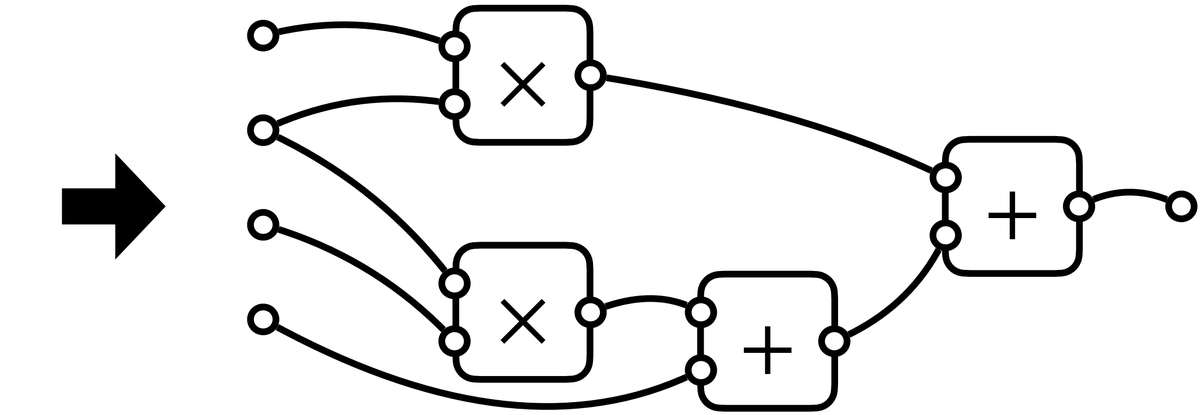
ACCESS



completeness
soundness
~~zero-knowledge~~

Terminology

ACCESS

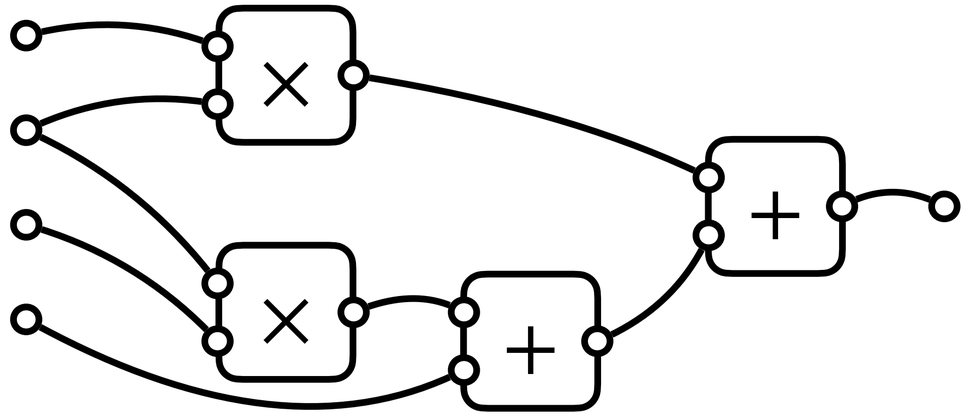


completeness
soundness
~~zero-knowledge~~

Over some large field $\mathbb{F}$
(i.e. $|\mathbb{F}| = \lambda^{\omega(1)}$)

Terminology

ACCESS



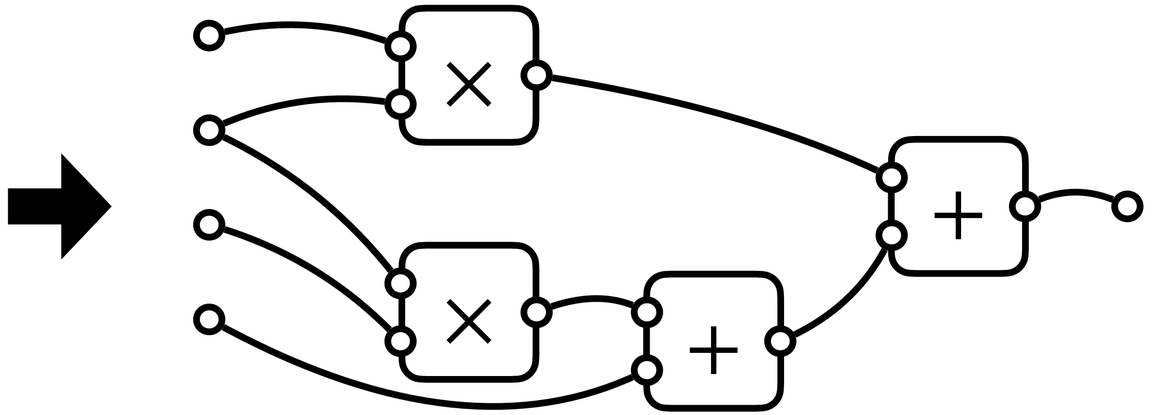
completeness
soundness
~~zero-knowledge~~

Over some large field $\mathbb{F}$
(i.e. $|\mathbb{F}| = \lambda^{\omega(1)}$)

Nonlinear gates:
MUL, INPUT
Linear gates:
ADD, SCALE

Terminology

ACCESS



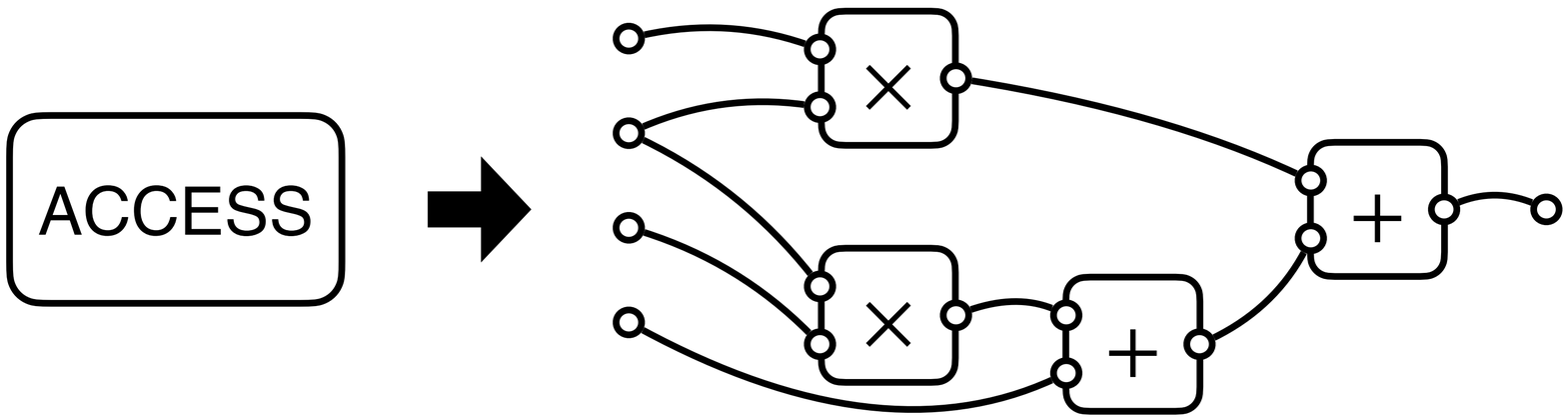
completeness
soundness
~~zero-knowledge~~

Over some large field $\mathbb{F}$
(i.e. $|\mathbb{F}| = \lambda^{\omega(1)}$)

Nonlinear gates:
MUL, INPUT

Linear gates:
ADD, SCALE

Terminology



completeness
soundness
~~zero-knowledge~~

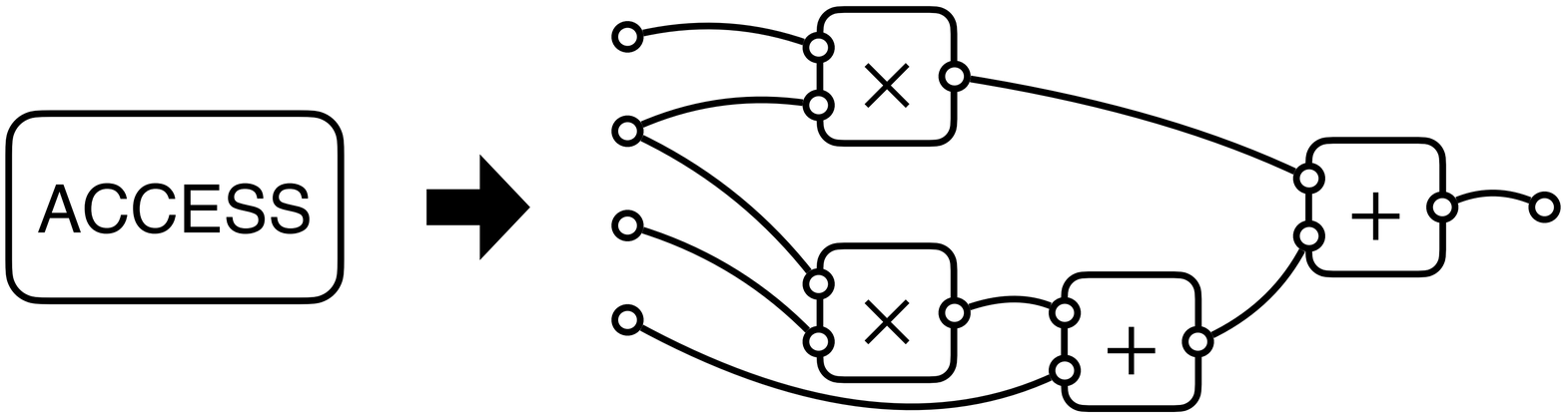
Over some large field $\mathbb{F}$
(i.e. $|\mathbb{F}| = \lambda^{\omega(1)}$)

Nonlinear gates:
MUL, INPUT

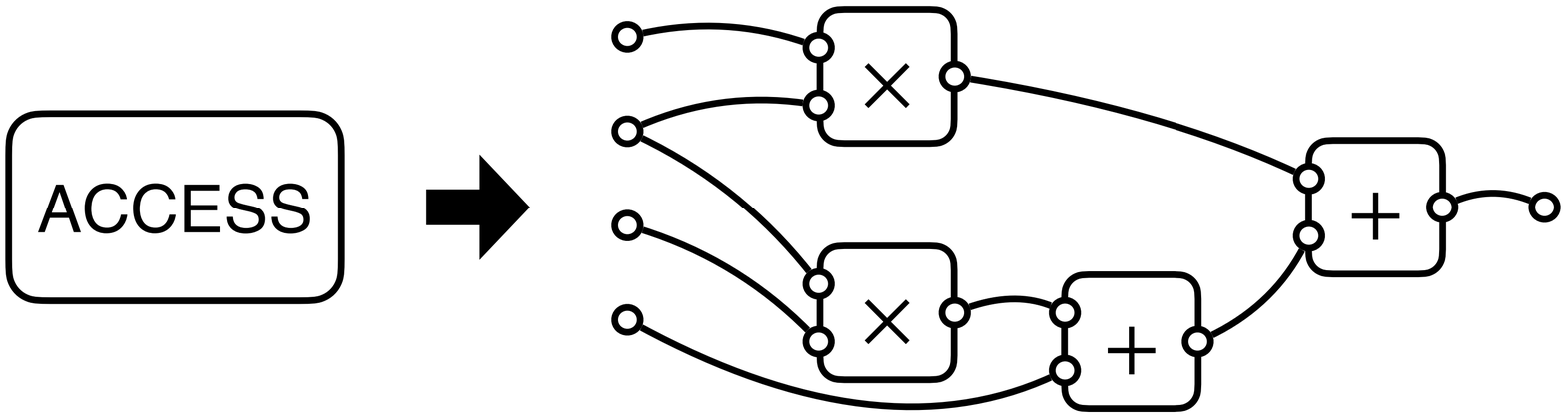
Linear gates:
ADD, SCALE

 # memory slots: n
 # accesses: T

Our Result

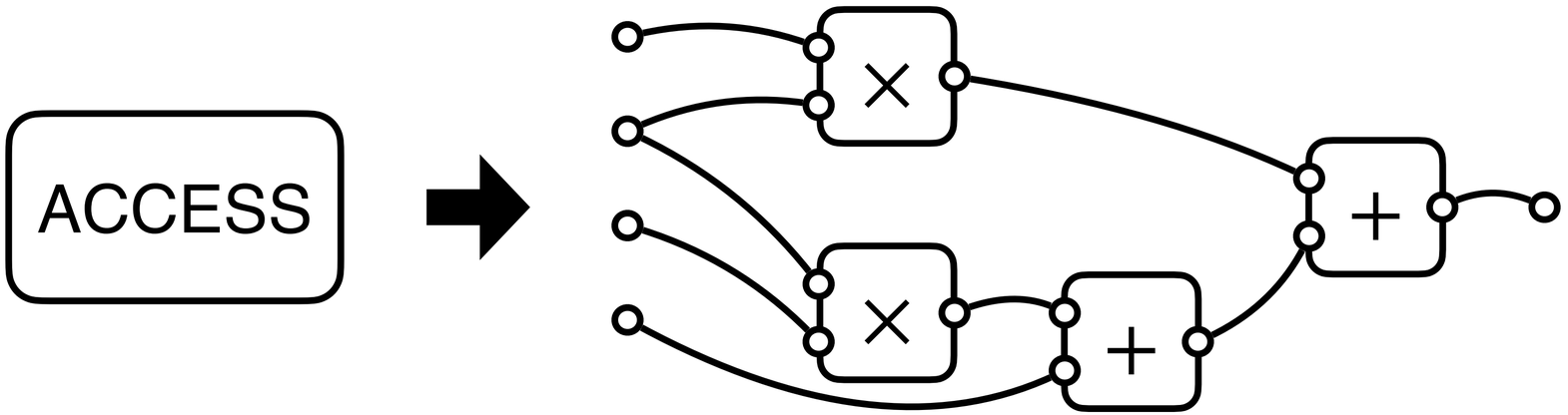


Our Result



Naïve linear scan: $O(Tn)$

Our Result

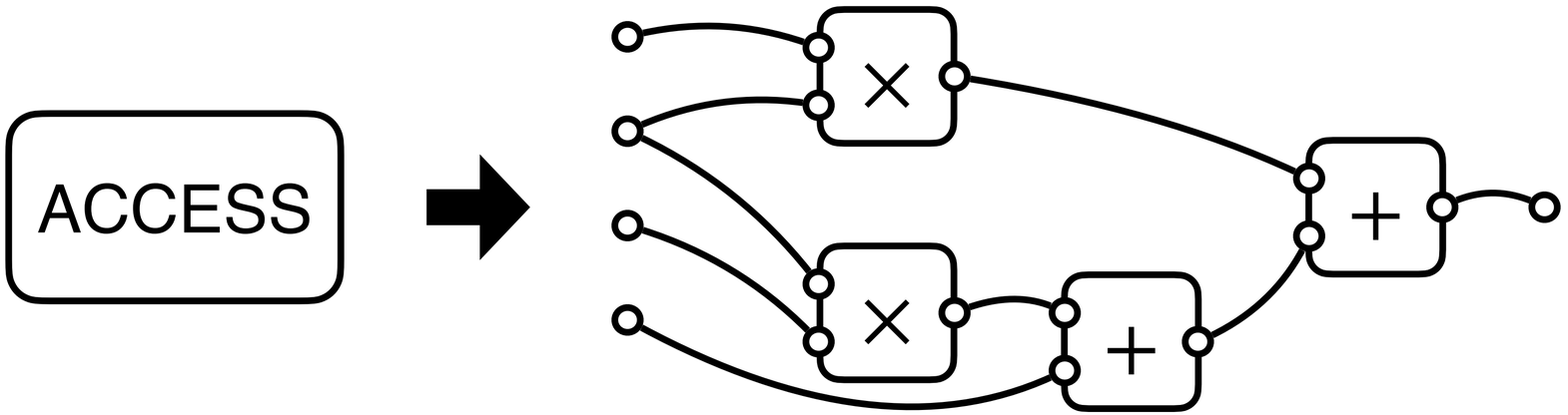


Naïve linear scan: $O(Tn)$

v.s.

Our construction: $O(T + n)$

Our Result



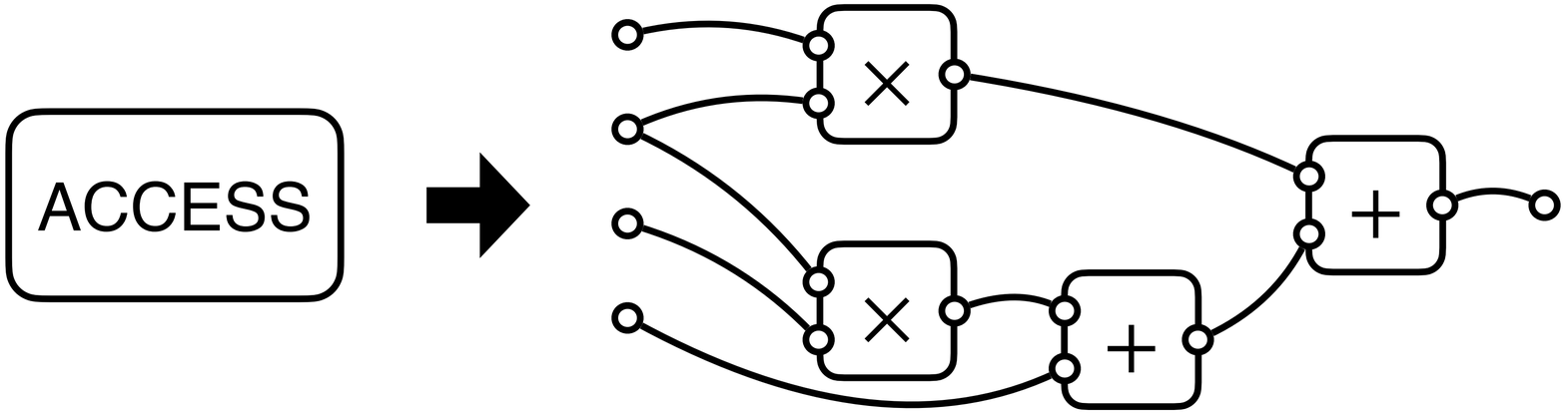
Naïve linear scan: $O(Tn)$

v.s.

Our construction: $O(T + n)$

Hidden constant: 10

Our Result

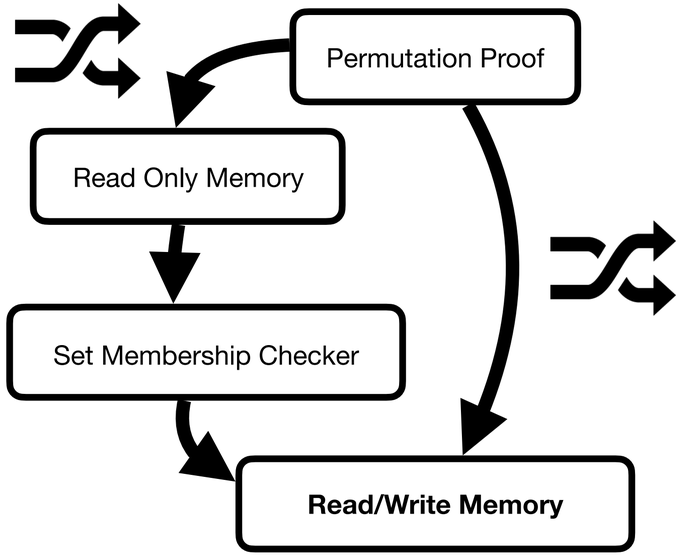


Naïve linear scan: $O(Tn)$

v.s.

Our construction: $O(T + n)$

Hidden constant: 10



Constant-Overhead Zero-Knowledge for RAM Programs

Nicholas Franzese* Jonathan Katz† Steve Lu‡ Rafail Ostrovsky§ Xiao Wang*
Chenkai Weng*

Abstract

We show a *constant-overhead* interactive zero-knowledge (ZK) proof system for RAM programs, that is, a ZK proof in which the communication complexity as well as the running times of the prover and verifier scale linearly in the size of the memory N and the running time T of the underlying RAM program. Besides yielding an asymptotic improvement of prior work, our implementation gives concrete performance improvements for RAM-based ZK proofs. In particular, our implementation supports ZK proofs of private read/write accesses to 64 MB of memory (2^{24} 32-bit words) using only 34 bytes of communication per access, a more than 80× improvement compared to the recent BubbleRAM protocol. We also design a lightweight RISC CPU that can efficiently emulate the MIPS-I instruction set, and for which our ZK proof communicates only ≈ 320 bytes per cycle, more than 10× less than the BubbleRAM CPU. In a 100 Mbps network, we can perform zero-knowledge executions of our CPU (with 64 MB of main memory and 4 MB of program memory) at a clock rate of 6.6 KHz.

1 Introduction

Zero-knowledge (ZK) proofs enable a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that the prover knows a witness w on which a particular program P evaluates to 1 without revealing anything additional about w. A series of works over the past several years (e.g., [GGPR13, JKO13, BCC+16, Gro16, AHIV17, KKW18, BBB+18, XZZ+19, TS20, WYKW21]) has shown several highly efficient ZK protocols, however for the most part these improved protocols have focused on the case where the program P is represented as a boolean or arithmetic circuit. This makes such proof systems somewhat difficult to apply to the arguably more natural setting where P is a program intended to be run on a general-purpose CPU, that is, when P is represented as a program in the *random-access machine (RAM)* model of computation. Although any such program can be converted to a circuit, doing so can be challenging and time consuming; more importantly, it can lead to sub-optimal performance as a general RAM program running in time T and using memory of size N requires a circuit of size $\Theta(TN)$ to verify its execution.

Some prior work has shown ZK proofs in the RAM model of computation. Hu, Mohassel, and Rosulek [HMR15], and subsequently Mohassel, Rosulek and Scafuro [MRS17], proposed an approach in which the addresses of memory accesses are revealed to the verifier; to account for that, they use oblivious RAM to make the original RAM program oblivious. Their focus was on asymptotic performance, and to the best of our knowledge their protocols have never been implemented. The TinyRAM framework [BCG+13, BCTV14, WSR+15] avoids the use of oblivious

*Northwestern University, {nicholasfranzese2026@u,wangxiao@cs,ckweng@u}.northwestern.edu
†University of Maryland, jkatz2@gmail.com
‡Stealth Software Technologies, Inc., steve@stealthsoftwareinc.com
§UCLA, rafail@cs.ucla.edu

Efficient Proof of RAM Programs from Any Public-Coin Zero-Knowledge System

Cyprien Delpech de Saint Guilhem Emmanuela Orsini Titouan Tanguy
Michiel Verbauwhede

imec-COSIC, KU Leuven, Belgium
firstname.lastname@kuleuven.be

Abstract

We show a compiler that allows to prove the correct execution of RAM programs using any zero-knowledge system for circuit satisfiability. At the core of this work is an arithmetic circuit which verifies the consistency of a list of memory access tuples in zero-knowledge.

Using such a circuit, we obtain the first constant-round and concretely efficient zero-knowledge proof protocol for RAM programs using any *stateless* zero-knowledge proof system for Boolean or arithmetic circuits. Both the communication complexity and the prover and verifier run times asymptotically scale linearly in the size of the memory and the run time of the RAM program; we demonstrate concrete efficiency with performance results of our `C++` implementation.

We concretely instantiate our construction with an efficient MPC-in-the-Head proof system, Limbo (ACM CCS 2021). The `C++` implementation of our access protocol extends that of Limbo and provides interactive proofs with 40 bits of statistical security with an amortized cost of 0.42ms of prover time and 2.8KB of communication per memory access, independently of the size of the memory; with multi-threading, this cost is reduced to 0.12ms and 1.8KB respectively. This performance of our *public-coin* protocol approaches that of *private-coin* protocol BubbleRAM (ACM CCS 2020, 0.15ms and 1.5KB per access).

1 Introduction

A zero-knowledge (ZK) proof is a fundamental cryptographic tool which proves that a statement is true without revealing any other information. Since their introduction by Goldwasser, Micali and Rackoff [GMR85], ZK proofs have had a significant impact on cryptography and have been the object of intense research work due to their theoretical importance and varied applicability.

Many types of ZK proof systems exist, each presenting different trade-offs between several efficiency measures. While in blockchain applications, the main focus is on succinct proofs of small statements [GGPR13, Gro16, Set20], another line of research has focused on prover efficiency [JKO13, AHIV17, KKW18, BCR+19, DIO21, dOT21], while other works have successfully constructed ZK proof systems for very large statements with good concrete efficiency [WYKW21, YSWW21, WYX+21, BMRS21].

Unfortunately, these works focus mostly on statements represented as circuits, either Boolean or arithmetic, which can incur a significant overhead to prove properties of large statements that are more naturally represented as random-access machine (RAM) programs. Many interesting functions and applications, such as private database search or verification of program execution,

Constant-Overhead Zero-Knowledge for RAM Programs

Nicholas Franzese* Jonathan Katz† Steve Lu‡ Rafail Ostrovsky§ Xiao Wang*
Chenkai Weng*

Abstract

We show a *constant-overhead* interactive zero-knowledge (ZK) proof system for RAM programs, that is, a ZK proof in which the communication complexity as well as the running times of the prover and verifier scale linearly in the size of the memory N and the running time T of the underlying RAM program. Besides yielding an asymptotic improvement of prior work, our implementation gives concrete performance improvements for RAM-based ZK proofs. In particular, our implementation supports ZK proofs of private read/write accesses to 64 MB of memory (2^{24} 32-bit words) using only 34 bytes of communication per access, a more than 80× improvement compared to the recent BubbleRAM protocol. We also design a lightweight RISC CPU that can efficiently emulate the MIPS-I instruction set, and for which our ZK proof communicates only ≈ 320 bytes per cycle, more than 10× less than the BubbleRAM CPU. In a 100 Mbps network, we can perform zero-knowledge executions of our CPU (with 64 MB of main memory and 4 MB of program memory) at a clock rate of 6.6 KHz.

1 Introduction

Zero-knowledge (ZK) proofs enable a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that the prover knows a witness w on which a particular program P evaluates to 1 without revealing anything additional about w. A series of works over the past several years (e.g., [GGPR13, JKO13, BCC+16, Gro16, AHIV17, KKW18, BBB+18, XZZ+19, TS20, WYKW21]) has shown several highly efficient ZK protocols, however for the most part these improved protocols have focused on the case where the program P is represented as a boolean or arithmetic circuit. This makes such proof systems somewhat difficult to apply to the arguably more natural setting where P is a program intended to be run on a general-purpose CPU, that is, when P is represented as a program in the *random-access machine (RAM)* model of computation. Although any such program can be converted to a circuit, doing so can be challenging and time consuming; more importantly, it can lead to sub-optimal performance as a general RAM program running in time T and using memory of size N requires a circuit of size $\Theta(TN)$ to verify its execution.

Some prior work has shown ZK proofs in the RAM model of computation. Hu, Mohassel, and Rosulek [HMR15], and subsequently Mohassel, Rosulek and Scafuro [MRS17], proposed an approach in which the addresses of memory accesses are revealed to the verifier; to account for that, they use oblivious RAM to make the original RAM program oblivious. Their focus was on asymptotic performance, and to the best of our knowledge their protocols have never been implemented. The TinyRAM framework [BCG+13, BCTV14, WSR+15] avoids the use of oblivious

*Northwestern University, {nicholasfranzese2026@u,wangxiao@cs,ckweng@u}.northwestern.edu
†University of Maryland, jkatz2@gmail.com
‡Stealth Software Technologies, Inc., steve@stealthsoftwareinc.com
§UCLA, rafail@cs.ucla.edu

Efficient Proof of RAM Programs from Any Public-Coin Zero-Knowledge System

Cyprien Delpech de Saint Guilhem Emmanuela Orsini Titouan Tanguy
Michiel Verbauwhede

imec-COSIC, KU Leuven, Belgium
firstname.lastname@kuleuven.be

Abstract

We show a compiler that allows to prove the correct execution of RAM programs using any zero-knowledge system for circuit satisfiability. At the core of this work is an arithmetic circuit which verifies the consistency of a list of memory access tuples in zero-knowledge.

Using such a circuit, we obtain the first constant-round and concretely efficient zero-knowledge proof protocol for RAM programs using any *stateless* zero-knowledge proof system for Boolean or arithmetic circuits. Both the communication complexity and the prover and verifier run times asymptotically scale linearly in the size of the memory and the run time of the RAM program; we demonstrate concrete efficiency with performance results of our `C++` implementation.

We concretely instantiate our construction with an efficient MPC-in-the-Head proof system, `Limbo` (ACM CCS 2021). The `C++` implementation of our access protocol extends that of `Limbo` and provides interactive proofs with 40 bits of statistical security with an amortized cost of 0.42ms of prover time and 2.8KB of communication per memory access, independently of the size of the memory; with multi-threading, this cost is reduced to 0.12ms and 1.8KB respectively. This performance of our *public-coin* protocol approaches that of *private-coin* protocol BubbleRAM (ACM CCS 2020, 0.15ms and 1.5KB per access).

1 Introduction

A zero-knowledge (ZK) proof is a fundamental cryptographic tool which proves that a statement is true without revealing any other information. Since their introduction by Goldwasser, Micali and Rackoff [GMR85], ZK proofs have had a significant impact on cryptography and have been the object of intense research work due to their theoretical importance and varied applicability.

Many types of ZK proof systems exist, each presenting different trade-offs between several efficiency measures. While in blockchain applications, the main focus is on succinct proofs of small statements [GGPR13, Gro16, Set20], another line of research has focused on prover efficiency [JKO13, AHIV17, KKW18, BCR+19, DIO21, dOT21], while other works have successfully constructed ZK proof systems for very large statements with good concrete efficiency [WYKW21, YSWW21, WYX+21, BMRS21].

Unfortunately, these works focus mostly on statements represented as circuits, either Boolean or arithmetic, which can incur a significant overhead to prove properties of large statements that are more naturally represented as random-access machine (RAM) programs. Many interesting functions and applications, such as private database search or verification of program execution,

Two Shuffles Make a RAM: Improved Constant Overhead Zero Knowledge RAM

Yibin Yang* David Heath†

July 17, 2023

Abstract

We optimize Zero Knowledge (ZK) proofs of statements expressed as RAM programs over arithmetic values. Our arithmetic-circuit-based read/write memory uses only 4 input gates and 6 multiplication gates per memory access. This is an almost 3× total gate improvement over prior state of the art (Delpech de Saint Guilhem et al., SCN'22).

We implemented our memory in the context of ZK proofs based on vector oblivious linear evaluation (VOLE), and we further optimize based on techniques available in the VOLE setting. Our experiments show that (1) our total runtime improves over that of the prior best VOLE-ZK RAM (Franzese et al., CCS'21) by up to 20× and (2) on a typical hardware setup, we can achieve ≈ $600K$ RAM accesses per second.

We also develop improved read-only memory and set ZK data structures. These are used internally in our read/write memory and improve over prior work.

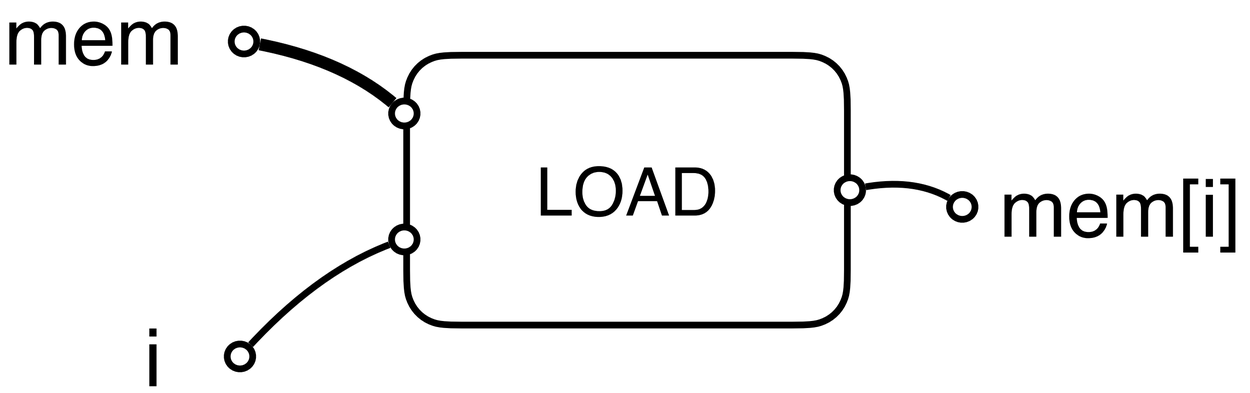
*Georgia Institute of Technology, yyang811@gatech.edu.
†University of Illinois Urbana-Champaign, daheath@illinois.edu

Up to ≈ 20 × improvement

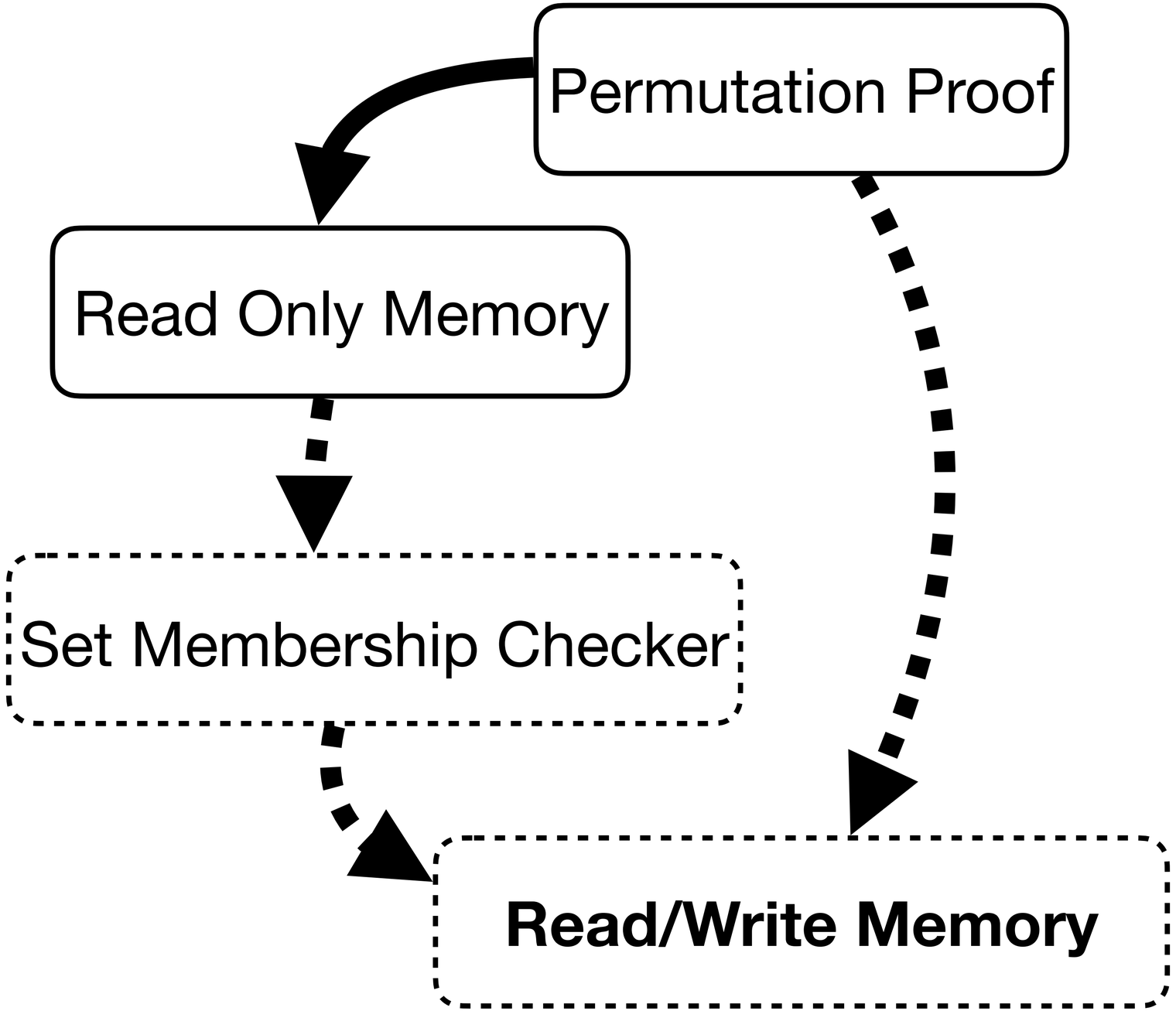
≈ 3 × improvement

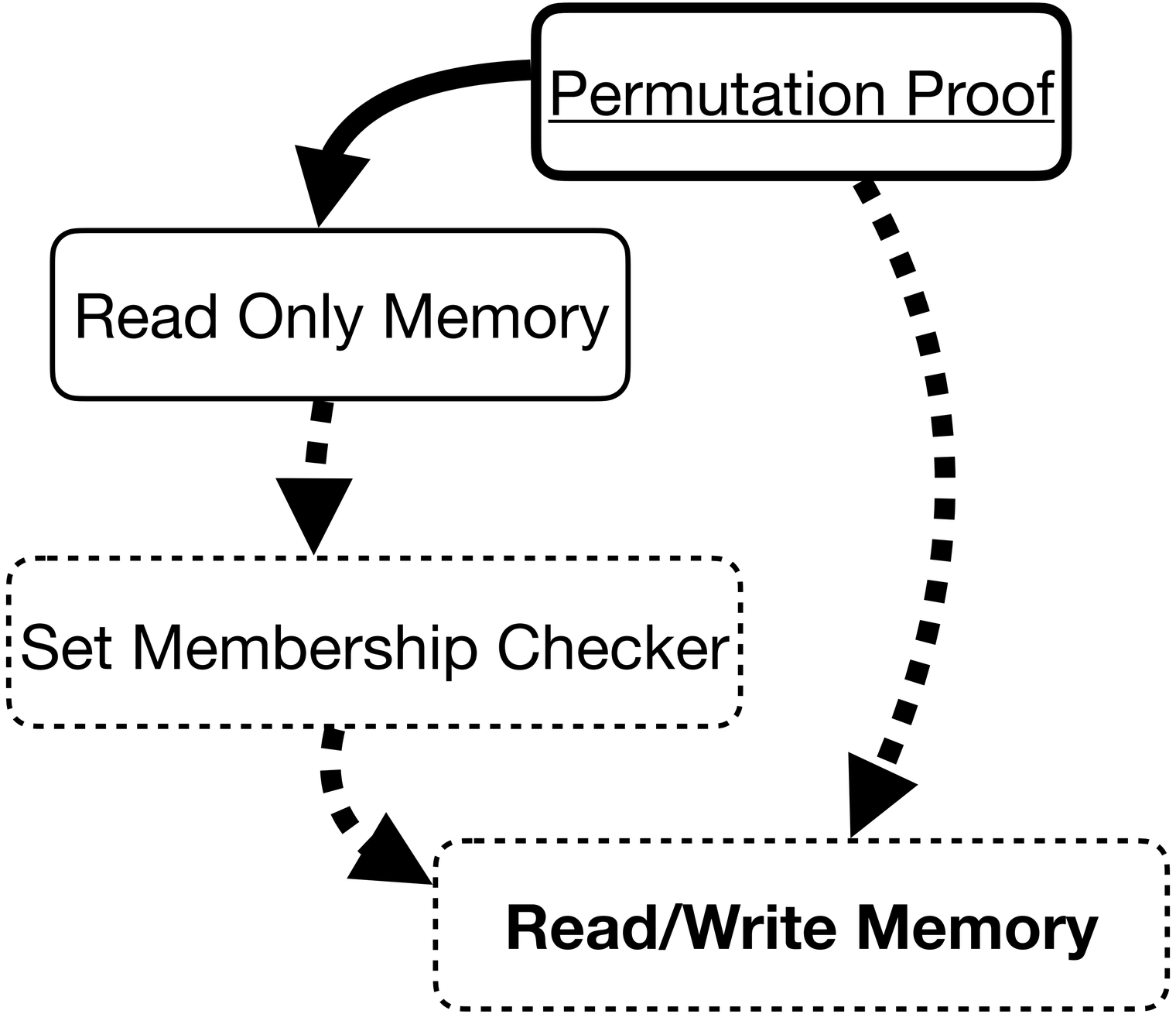
Today's Focus

Only



"One shuffle makes a ROM"


Permutation Proof
Read Only Memory
Set Membership Checker
Read/Write Memory


Permutation Proof
Read Only Memory
Set Membership Checker
Read/Write Memory

Preliminaries: **Proof of a Permutation**

Consider vectors of $\mathbb{F}$ elements $\overrightarrow{x}, \overrightarrow{y}$

Are $\overrightarrow{x}$ and $\overrightarrow{y}$ related by a permutation?

$$\overrightarrow{x} \sim \overrightarrow{y}$$

Preliminaries: **Proof of a Permutation**

Consider vectors of $\mathbb{F}$ elements $\vec{x}, \vec{y}$

Are $\vec{x}$ and $\vec{y}$ related by a permutation?

$$\vec{x} \sim \vec{y}$$

$$p(X) = \prod_i (X - x_i) \qquad \vec{x} \sim \vec{y} \iff p = q$$

$$q(X) = \prod_i (X - y_i)$$

Preliminaries: **Proof of a Permutation**

Consider vectors of $\mathbb{F}$ elements $\vec{x}, \vec{y}$

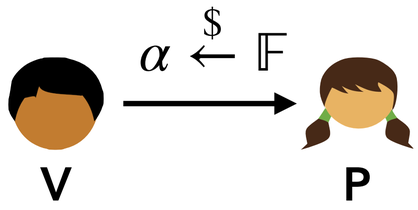
Are $\vec{x}$ and $\vec{y}$ related by a permutation?

$$\vec{x} \sim \vec{y}$$

$$p(X) = \prod_i (X - x_i)$$

$$q(X) = \prod_i (X - y_i)$$

$$\vec{x} \sim \vec{y} \iff p = q$$



Proof: $p(\alpha) = q(\alpha)$

Based on the SZDL Lemma [DL78,Sch80,Zip89]

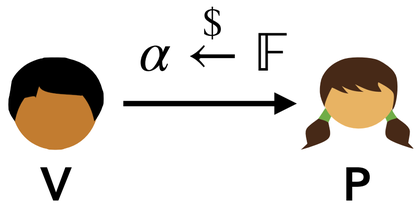
Preliminaries: **Proof of a Permutation**

Proving $\overrightarrow{x} \sim \overrightarrow{y}$ uses only $2n - 2$ MUL gates

$$p(X) = \prod_i (X - x_i)$$

$$q(X) = \prod_i (X - y_i)$$

$$\overrightarrow{x} \sim \overrightarrow{y} \iff p = q$$



Proof: $p(\alpha) = q(\alpha)$

Based on the SZDL Lemma [DL78,Sch80,Zip89]

Preliminaries: **Proof of a Permutation**

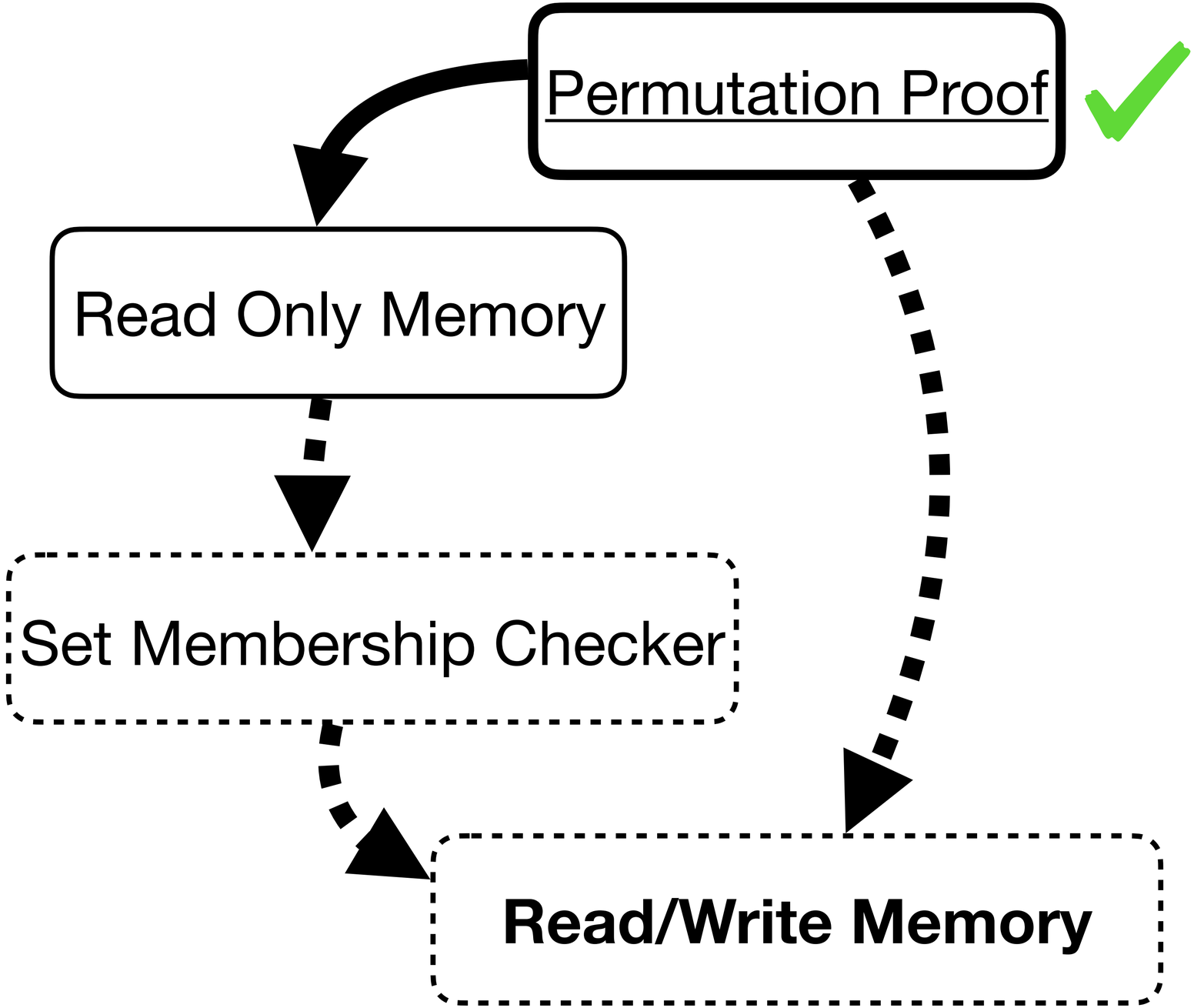
Proving $\vec{x} \sim \vec{y}$ uses only $2n - 2$ MUL gates

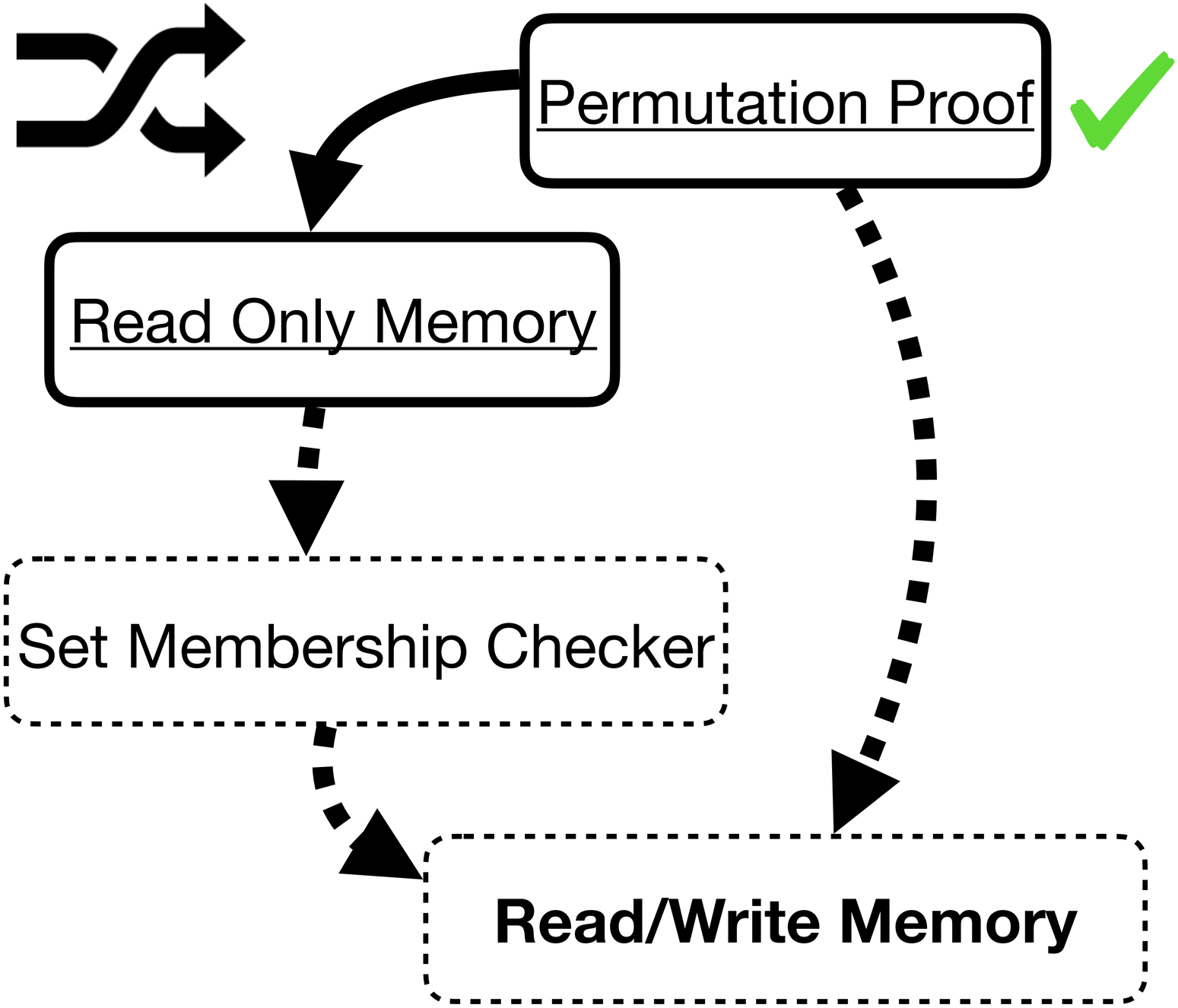
Generalizes to vectors of tuples [DdSGOTV22]

$$p(X) = \prod_i (X - \vec{x}_i)$$

$$q(X) = \prod_i (X - \vec{y}_i)$$

$$\vec{x} \sim \vec{y} \iff p = q$$


Permutation Proof
Read Only Memory
Set Membership Checker
Read/Write Memory


Permutation Proof
Read Only Memory
Set Membership Checker
Read/Write Memory

Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

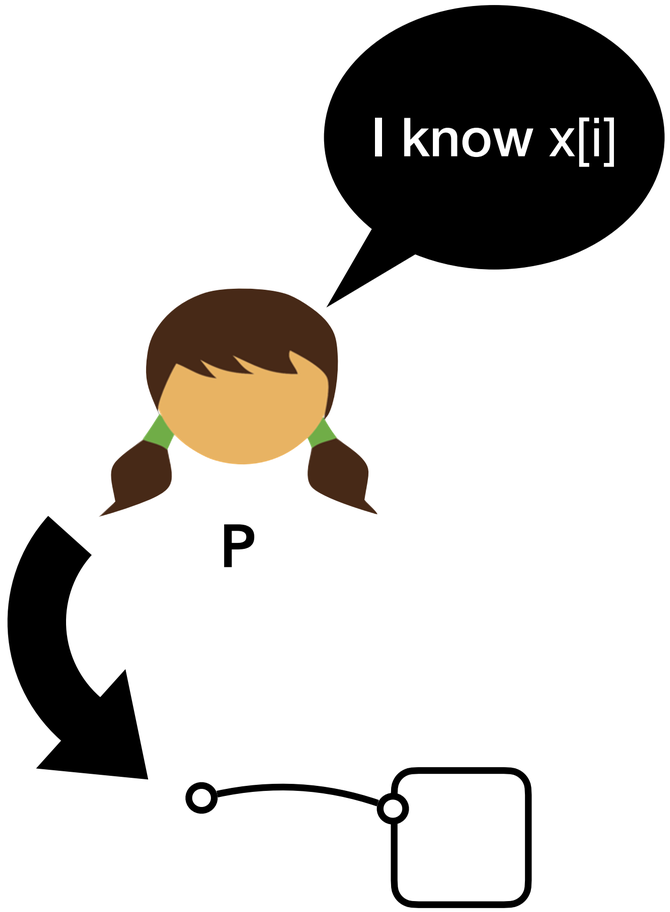
LOAD(i)

Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

LOAD(i)

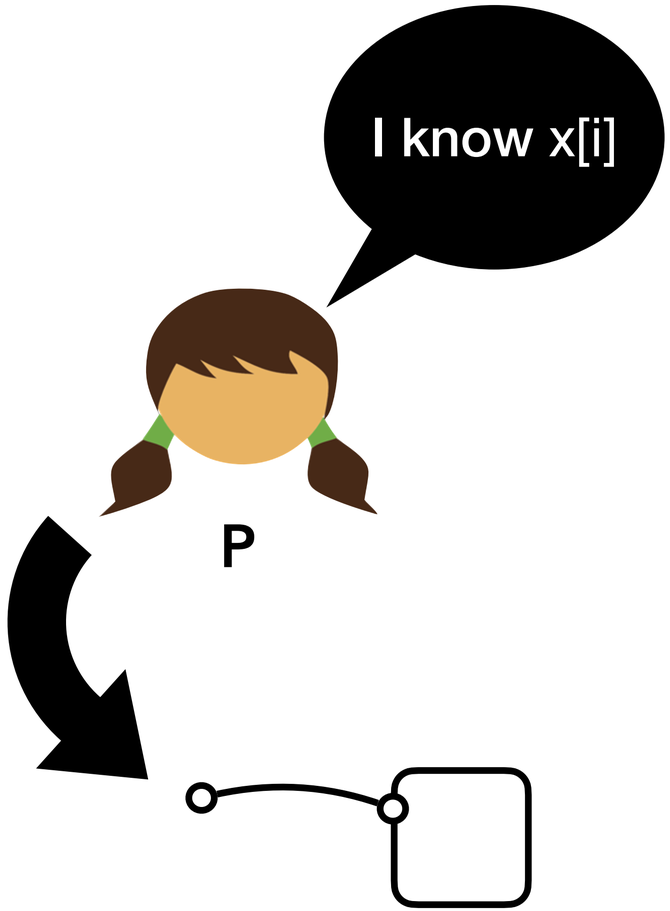
Key insight:
We just need to check P does not cheat



Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

LOAD(i)



Key insight:
We just need to check P does not cheat

How?
Construct the “one-time” readable entries, and create a “reading history log”

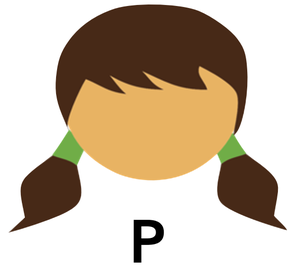
Read-Only Memory

Make vectors of triples: (index, value, version)

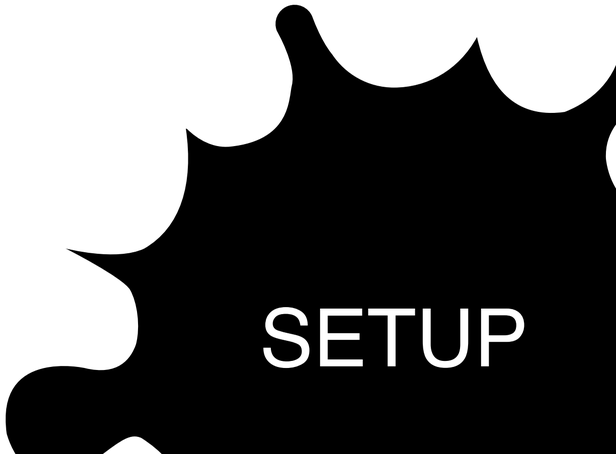
Key	Value
0	x[0]
1	x[1]
2	x[2]

writes

reads



P



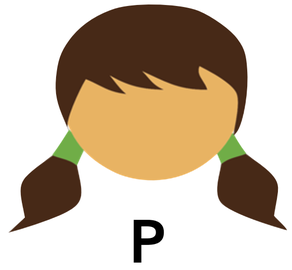
Read-Only Memory

Make vectors of triples: (index, value, version)

Key	Value
0	x[0]
1	x[1]
2	x[2]

writes 0,x[0],0 | 1,x[1],0 | 2,x[2],0

reads



P

SETUP

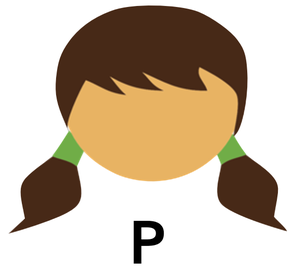
Read-Only Memory

Make vectors of triples: (index, value, version)

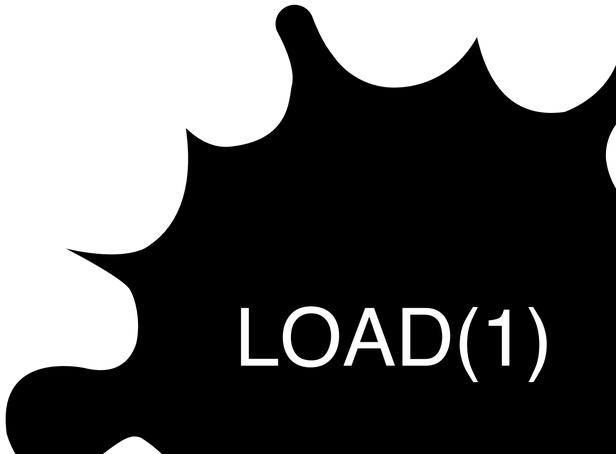
Key	Value
0	x[0]
1	x[1]
2	x[2]

writes 0,x[0],0 | 1,x[1],0 | 2,x[2],0

reads



P



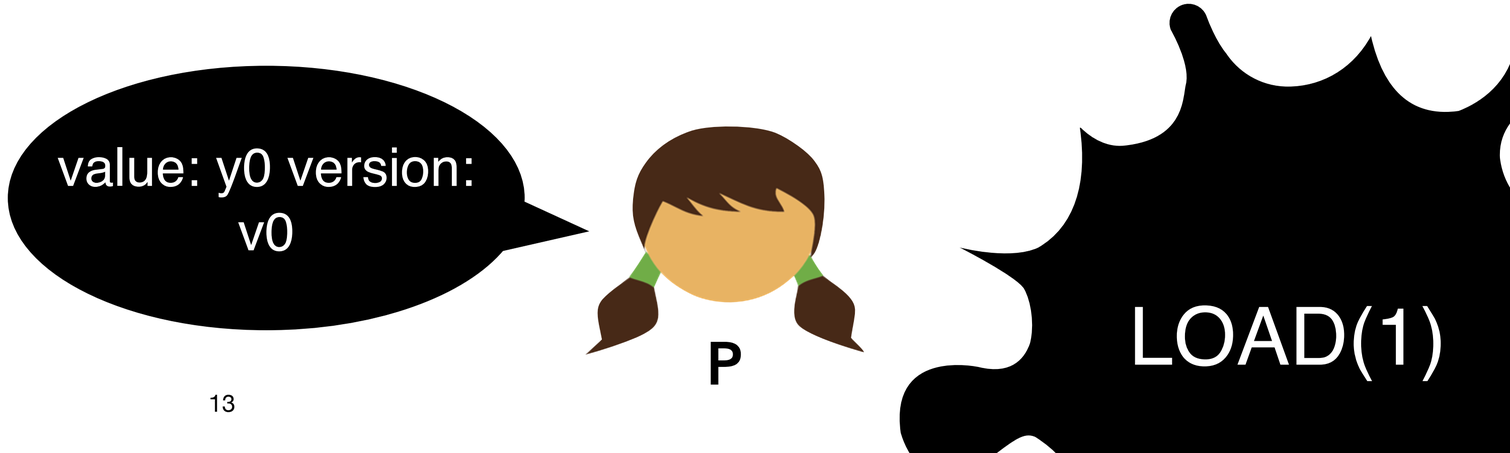
Read-Only Memory

Make vectors of triples: (index, value, version)

Key	Value
0	x[0]
1	x[1]
2	x[2]

writes 0,x[0],0 | 1,x[1],0 | 2,x[2],0

reads



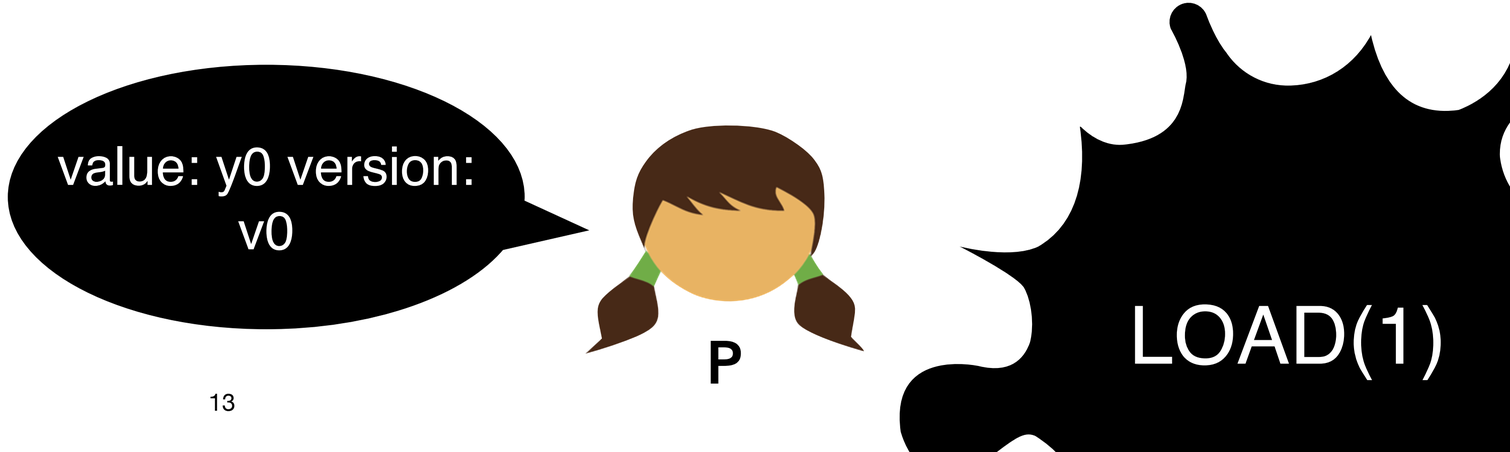
Read-Only Memory

Make vectors of triples: (index, value, version)

Key	Value
0	x[0]
1	x[1]
2	x[2]

writes 0,x[0],0 | 1,x[1],0 | 2,x[2],0 | 1,y0,v0+1

reads 1,y0,v0



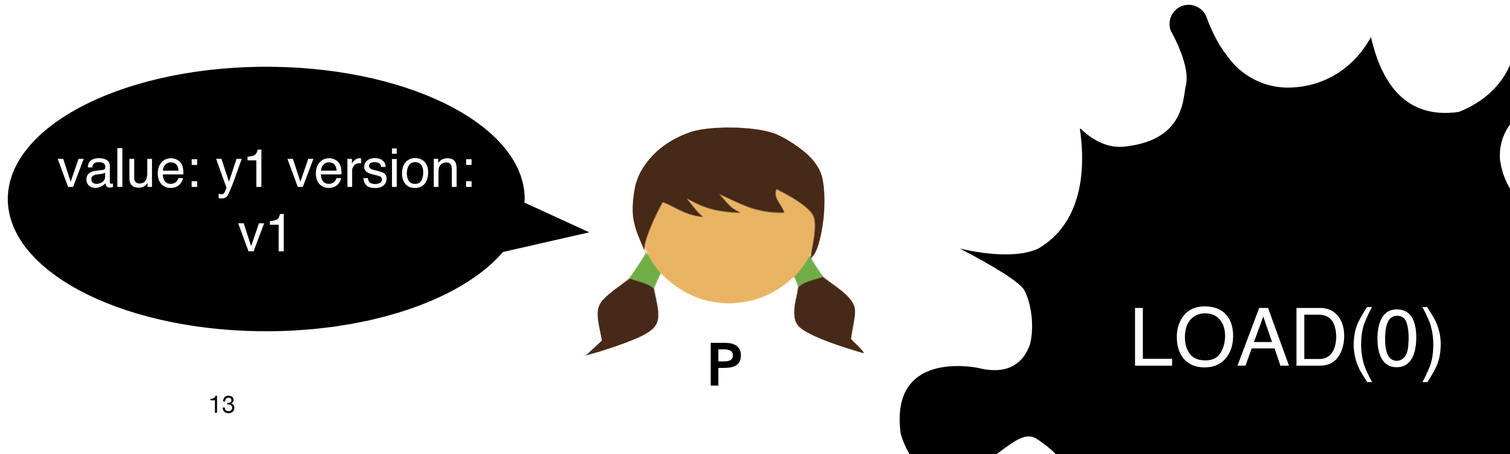
Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

Make vectors of triples: (index, value, version)

writes 0,x[0],0 | 1,x[1],0 | 2,x[2],0 | 1,y0,v0+1 | 0,y1,v1+1

reads 1,y0,v0 | 0,y1,v1

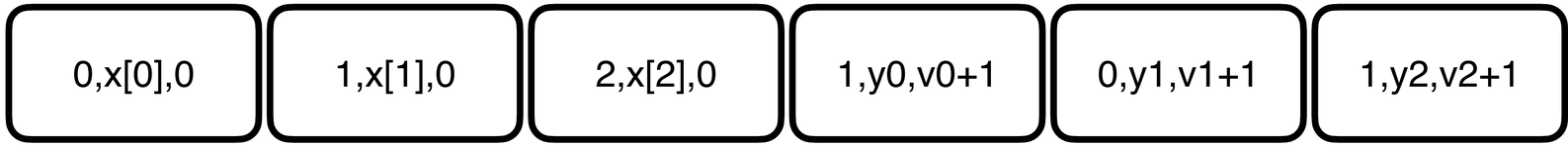


Read-Only Memory

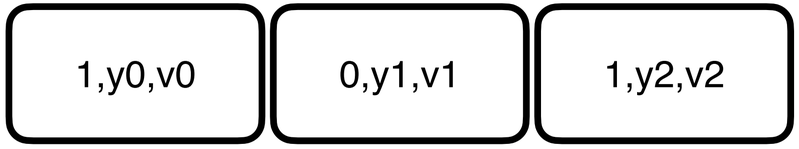
Key	Value
0	x[0]
1	x[1]
2	x[2]

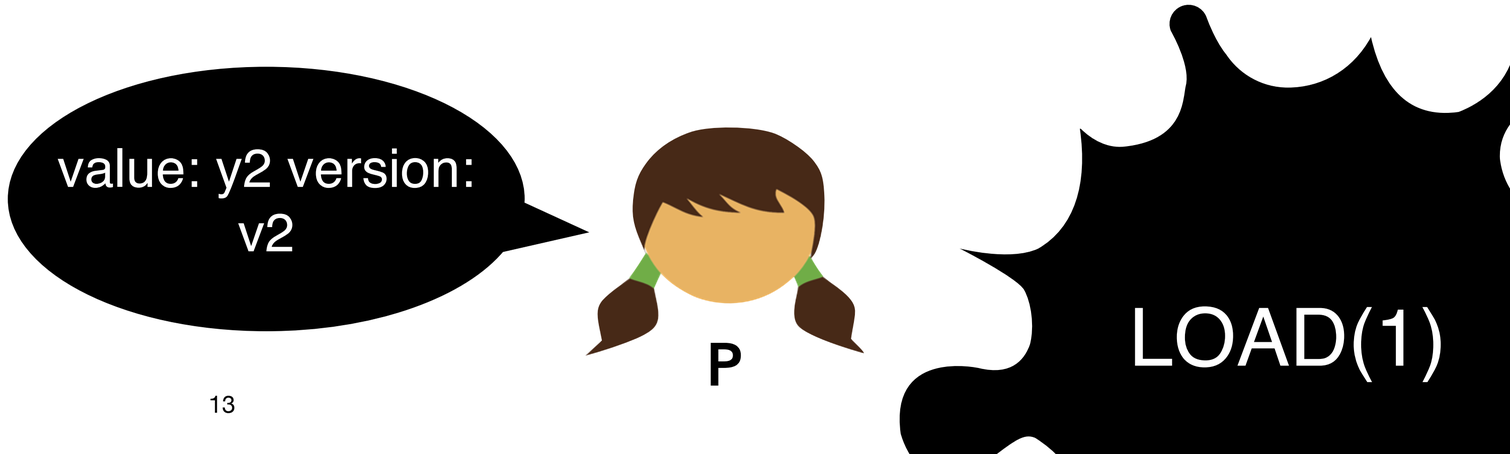
Make vectors of triples: (index, value, version)

writes



reads





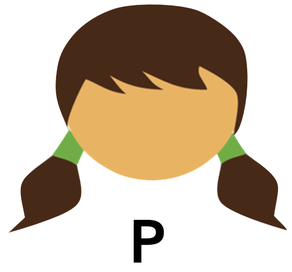
Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

Make vectors of triples: (index, value, version)

writes 0,x[0],0 | 1,x[1],0 | 2,x[2],0 | 1,y0,v0+1 | 0,y1,v1+1 | 1,y2,v2+1

reads 1,y0,v0 | 0,y1,v1 | 1,y2,v2



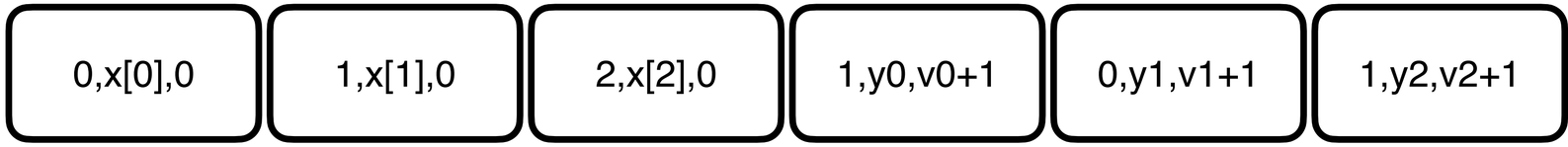
P

Read-Only Memory

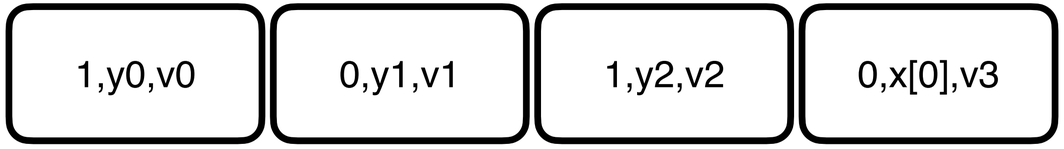
Key	Value
0	x[0]
1	x[1]
2	x[2]

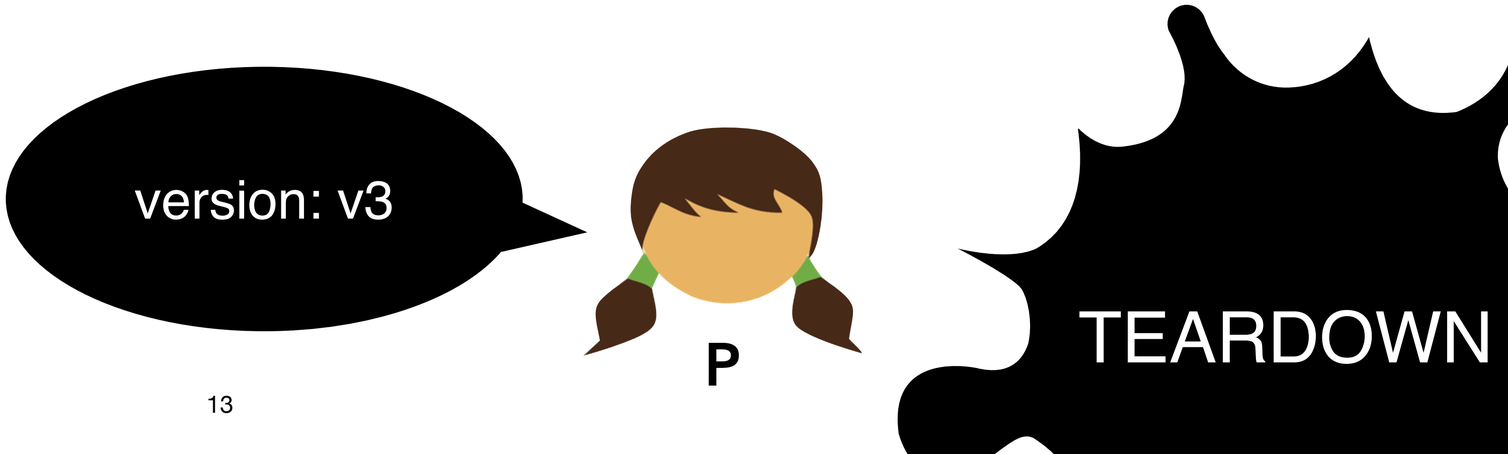
Make vectors of triples: (index, value, version)

writes



reads



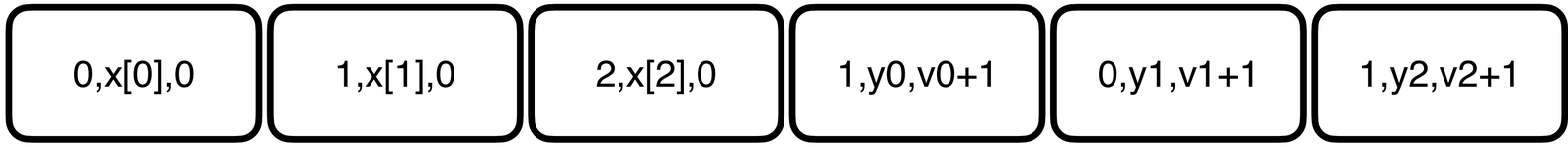


Read-Only Memory

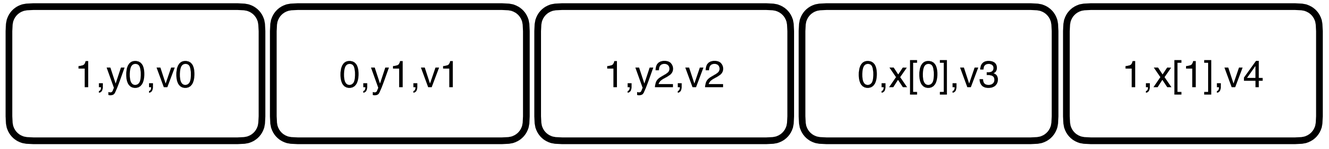
Key	Value
0	x[0]
1	x[1]
2	x[2]

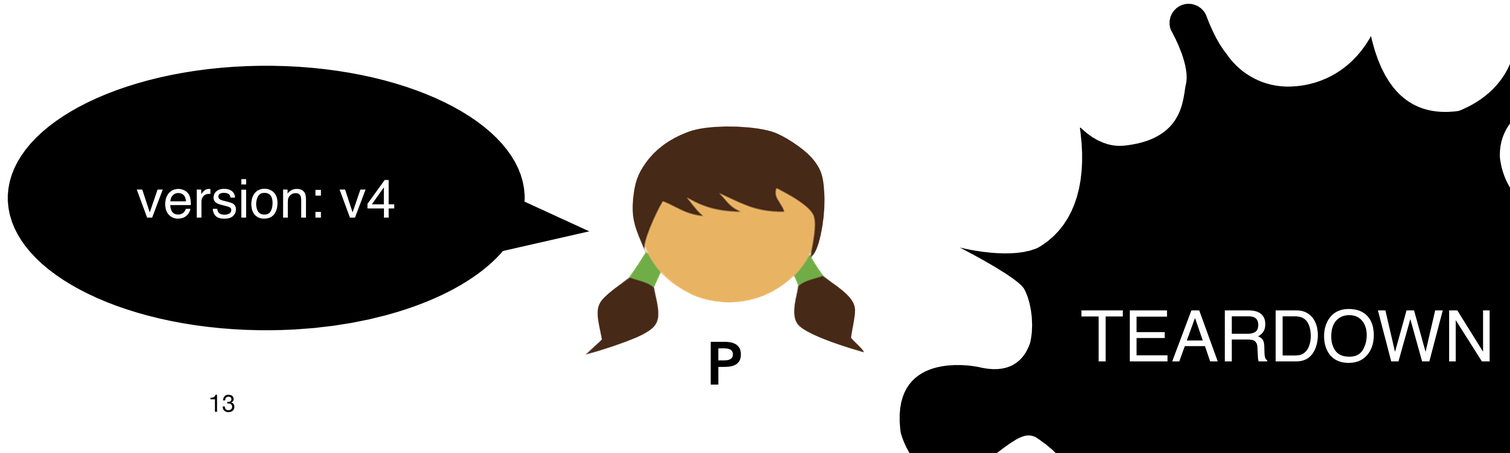
Make vectors of triples: (index, value, version)

writes



reads



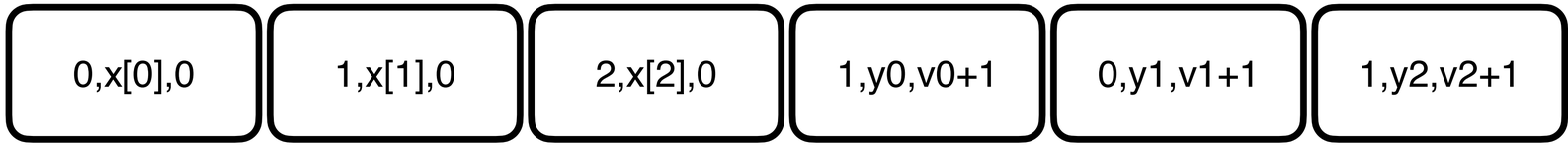


Read-Only Memory

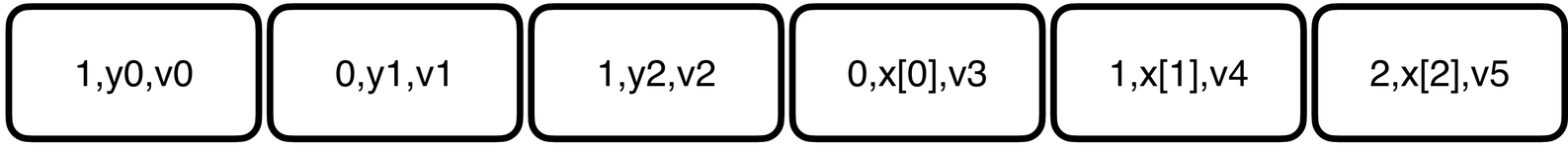
Key	Value
0	x[0]
1	x[1]
2	x[2]

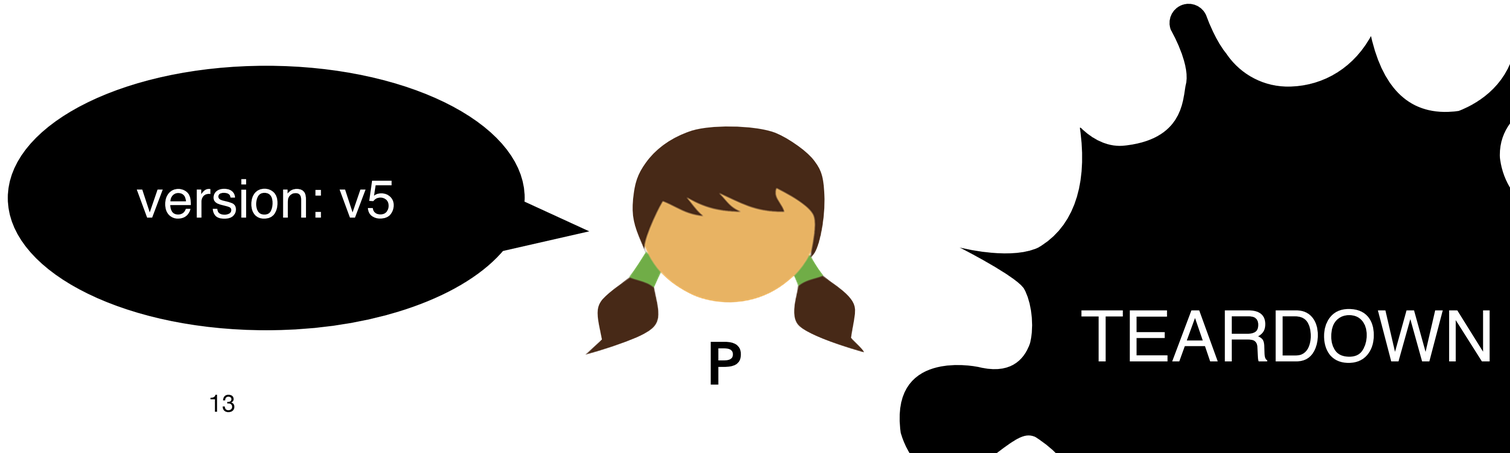
Make vectors of triples: (index, value, version)

writes



reads

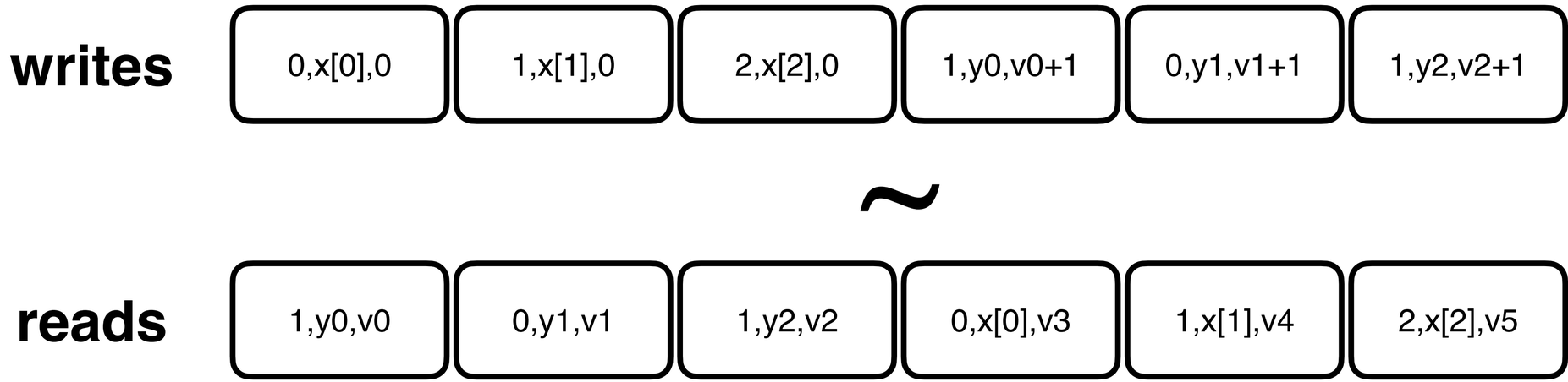




Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

Make vectors of triples: (index, value, version)



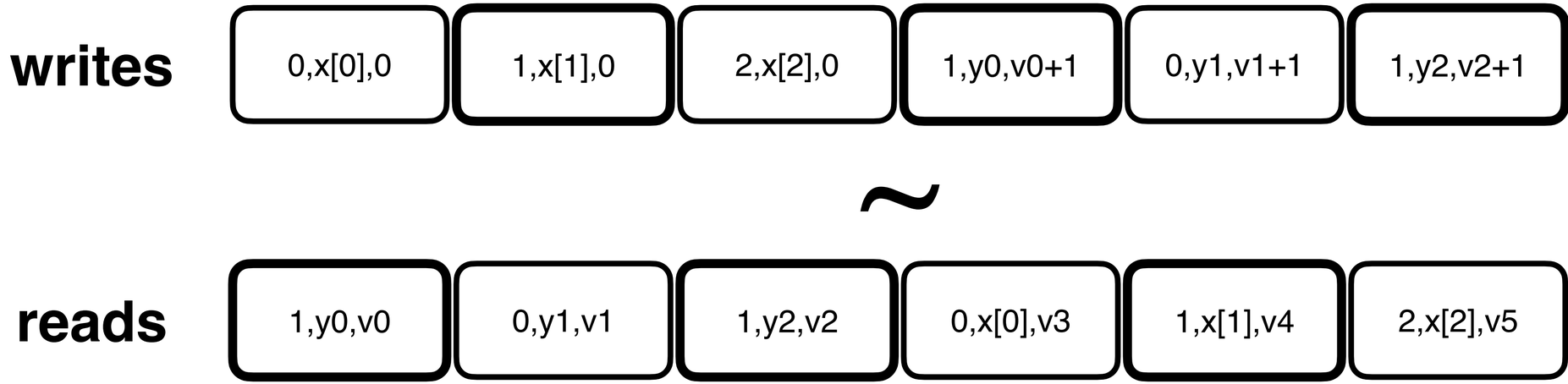
Use 2n+2T-2 MUL gates to check reads ~ writes

Claim: if reads ~ writes, P did not cheat

Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

Make vectors of triples: (index, value, version)



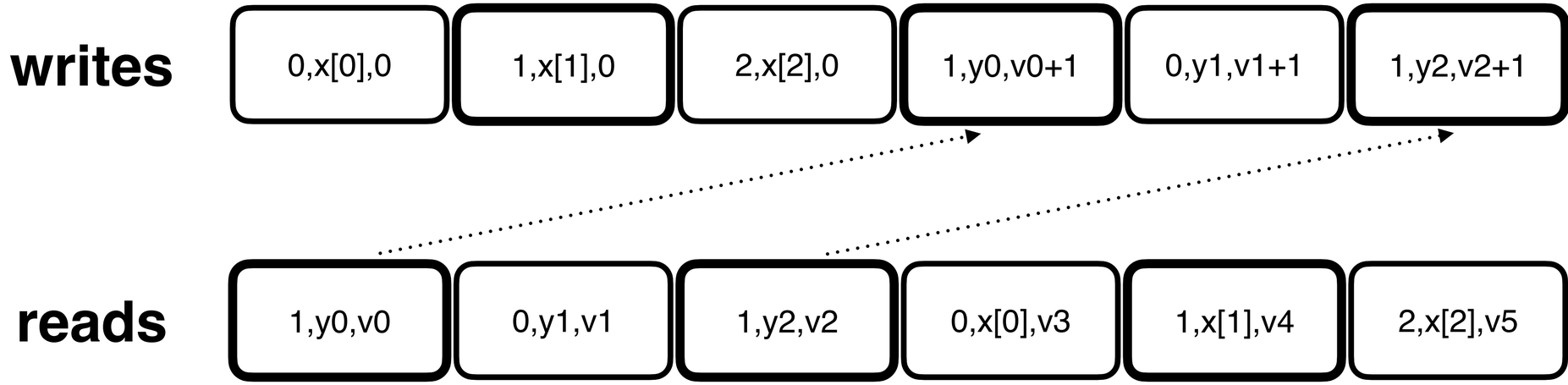
Use 2n+2T-2 MUL gates to check reads ~ writes

Claim: if reads ~ writes, P did not cheat

Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

Make vectors of triples: (index, value, version)



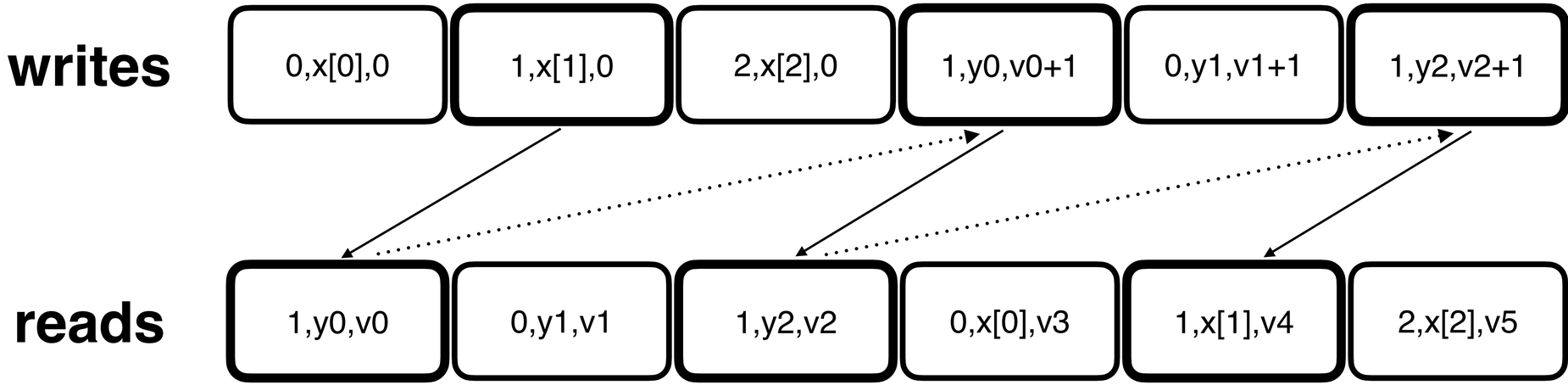
Use 2n+2T-2 MUL gates to check reads ~ writes

Claim: if reads ~ writes, P did not cheat

Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

Make vectors of triples: (index, value, version)



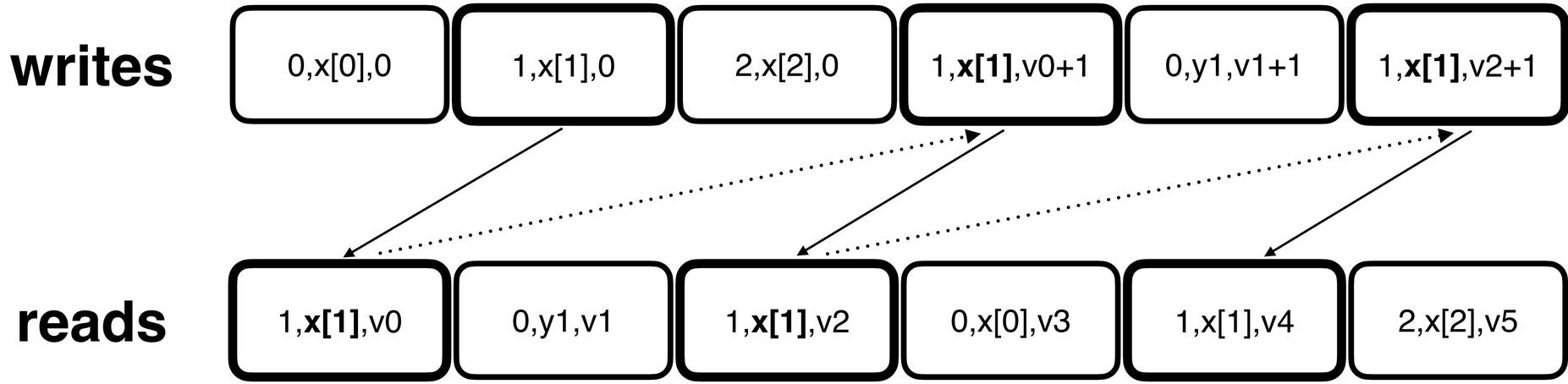
Use 2n+2T-2 MUL gates to check reads ~ writes

Claim: if reads ~ writes, P did not cheat

Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

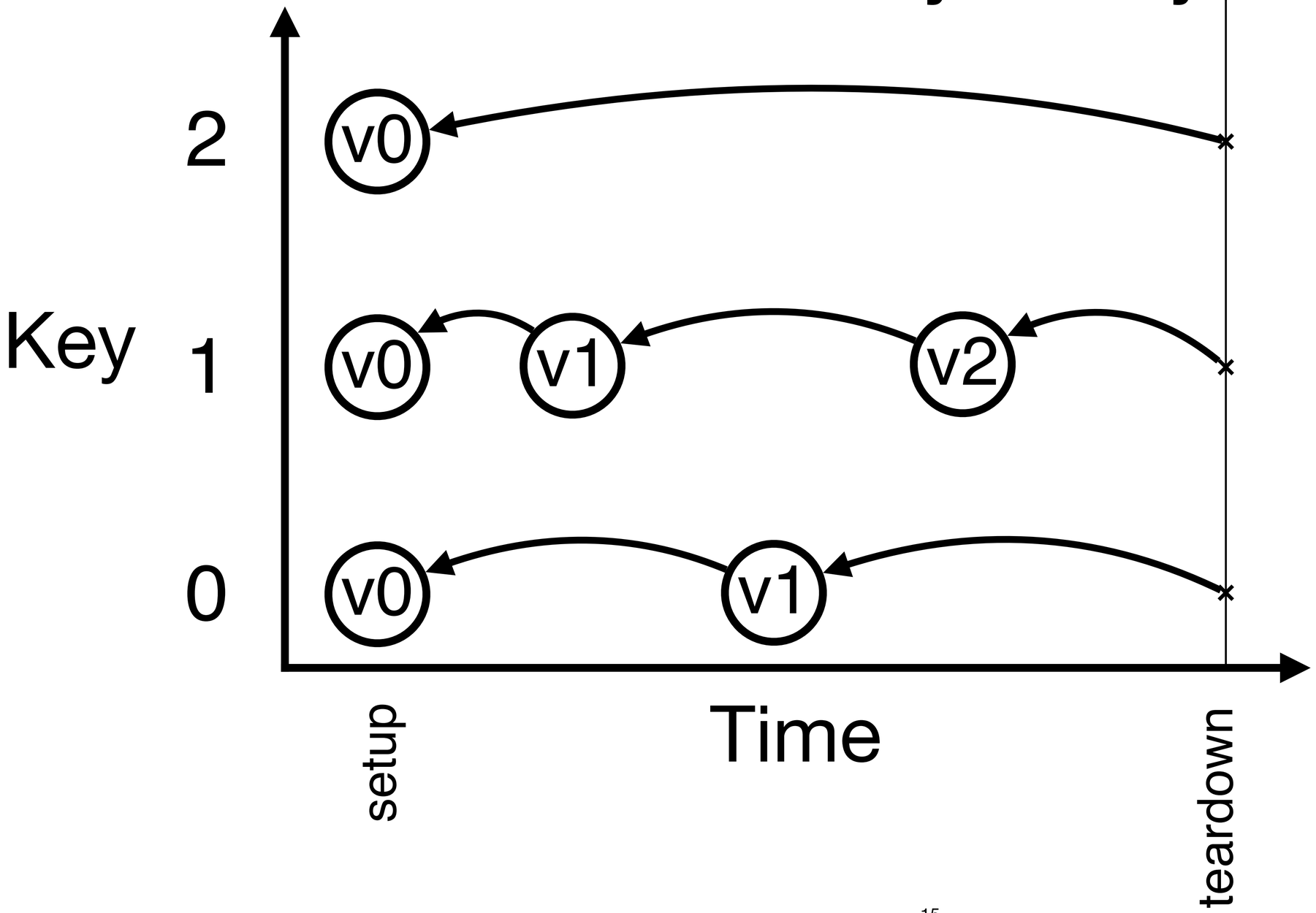
Make vectors of triples: (index, value, version)



Use 2n+2T-2 MUL gates to check reads ~ writes

Claim: if reads ~ writes, P did not cheat

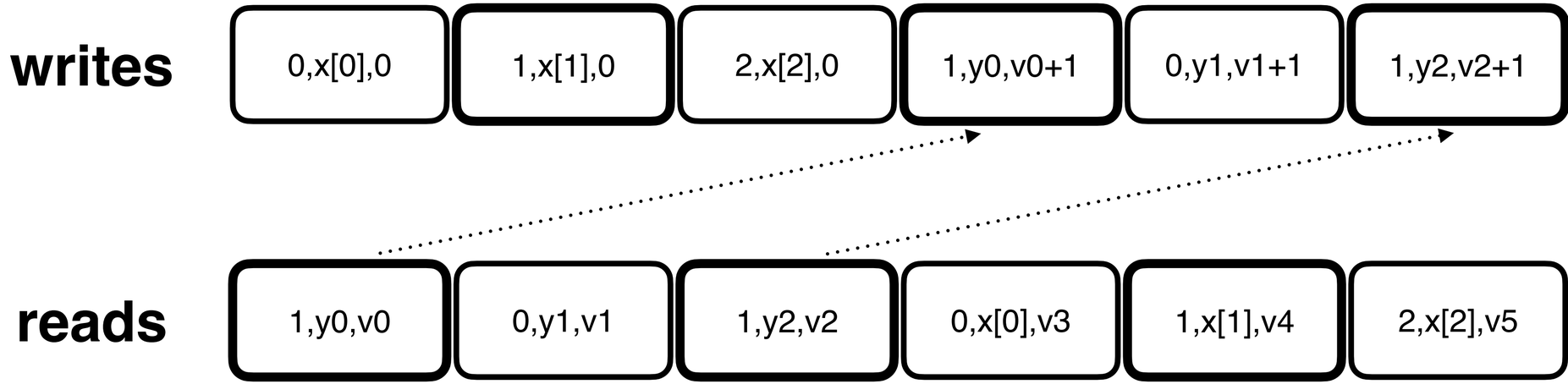
Read-Only Memory



Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

Make vectors of triples: (index, value, version)



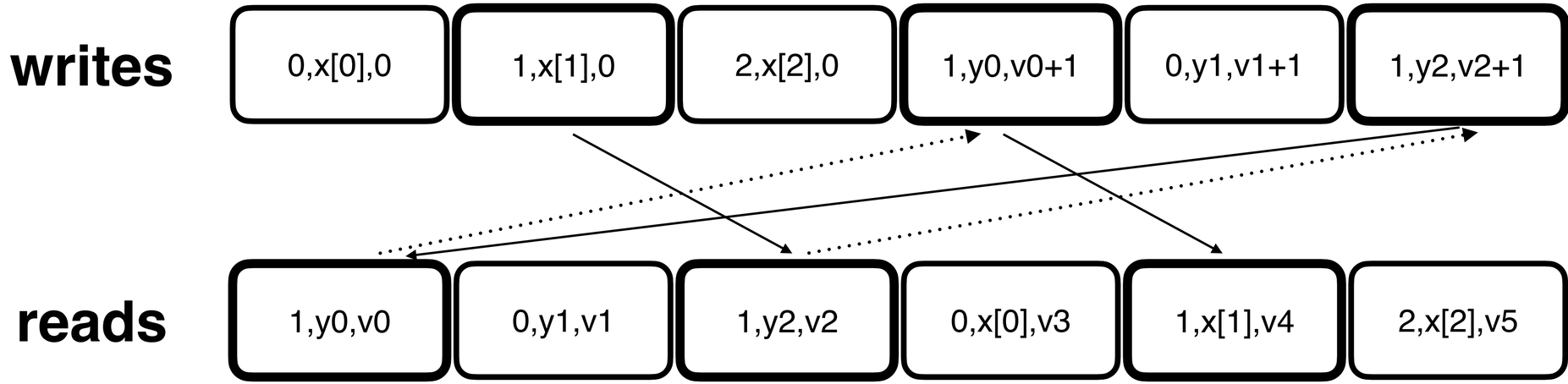
Use 2n+2T-2 MUL gates to check reads ~ writes

Claim: if reads ~ writes, P did not cheat

Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

Make vectors of triples: (index, value, version)



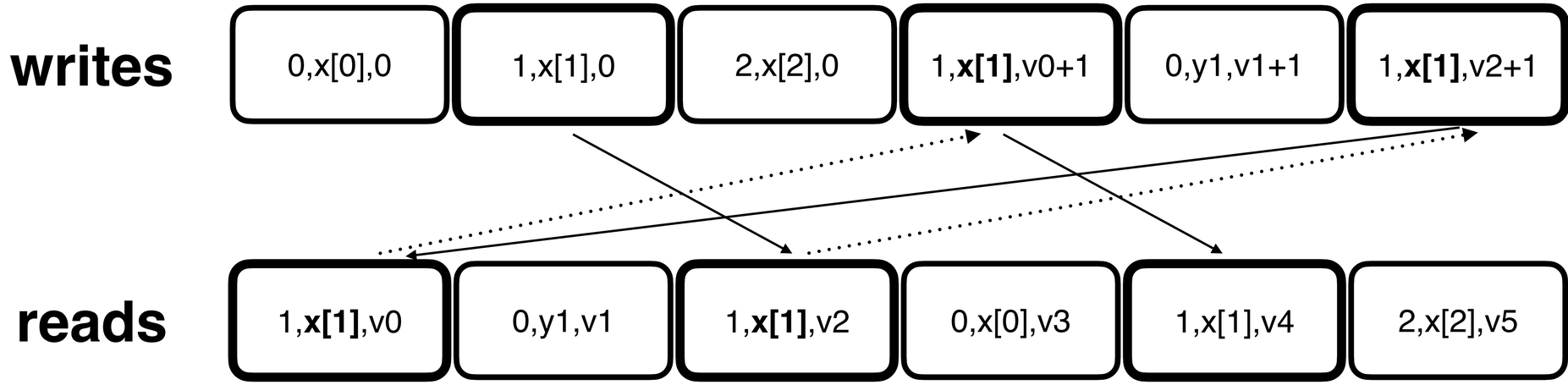
Use 2n+2T-2 MUL gates to check reads ~ writes

Claim: if reads ~ writes, P did not cheat

Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

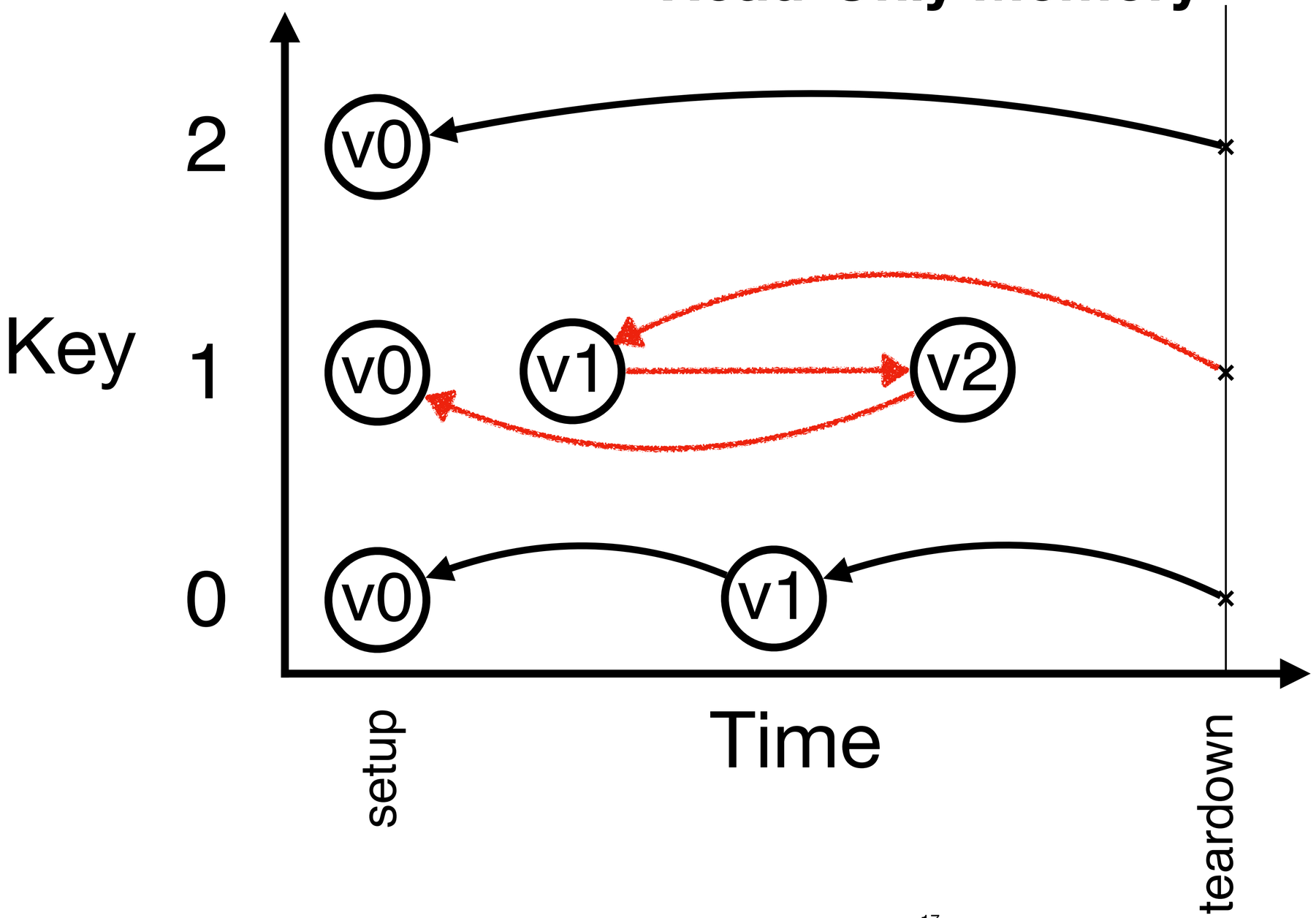
Make vectors of triples: (index, value, version)



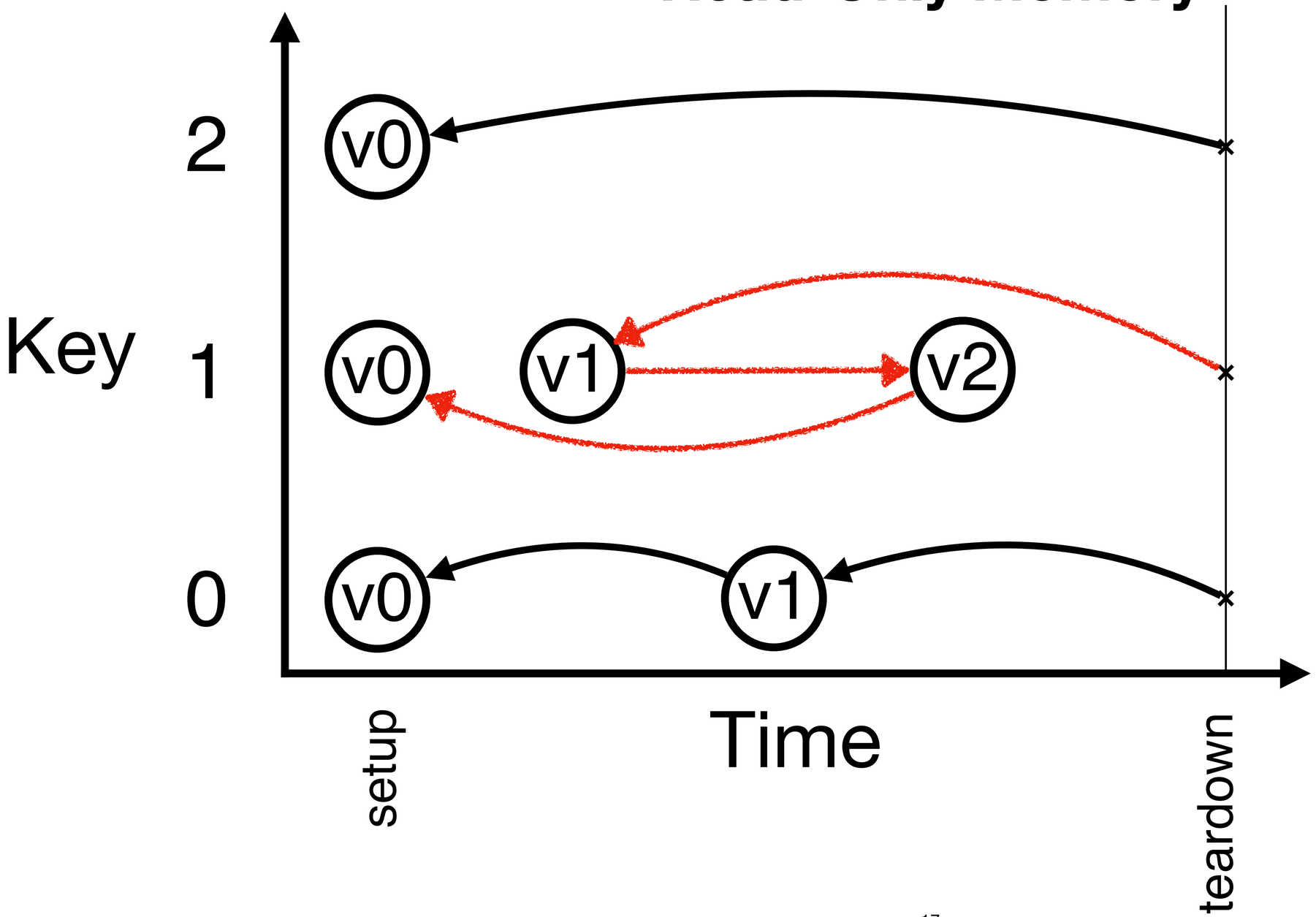
Use 2n+2T-2 MUL gates to check reads ~ writes

Claim: if reads ~ writes, P did not cheat

Read-Only Memory



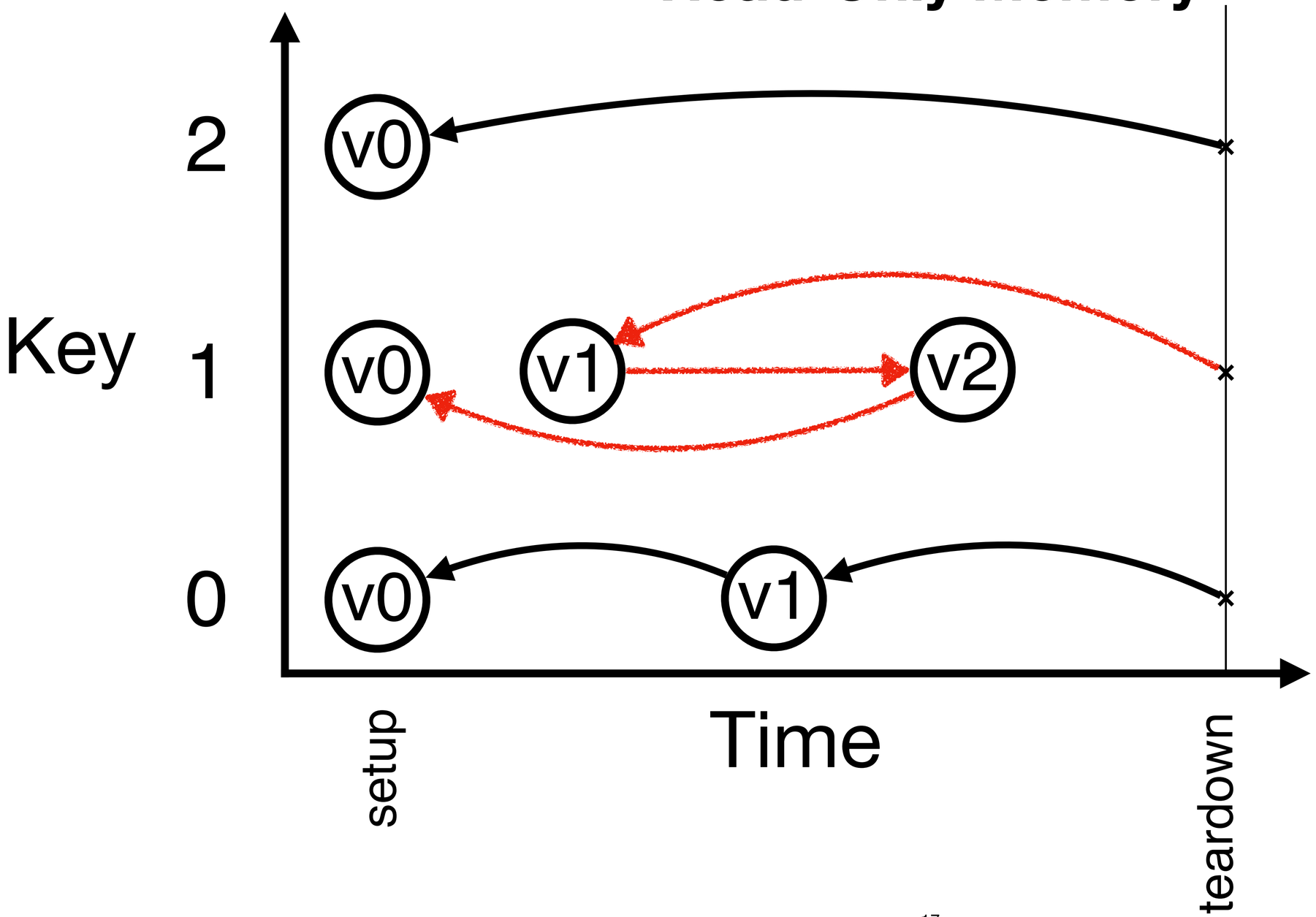
Read-Only Memory



Difference between ROM and RAM:

RAM must prevent P from reading from the future

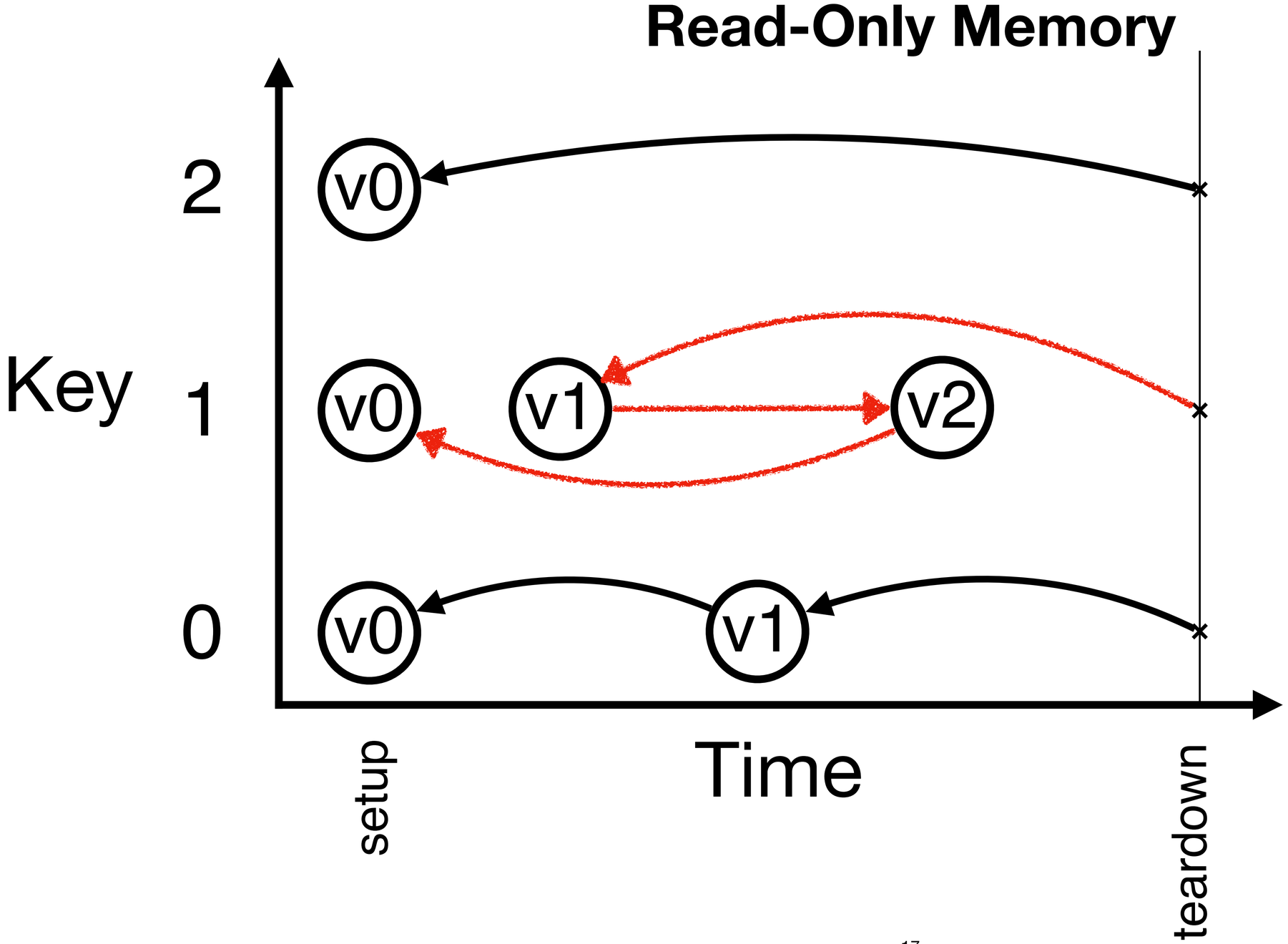
Read-Only Memory



Difference between ROM and RAM:

RAM must prevent P from reading from the future

How?

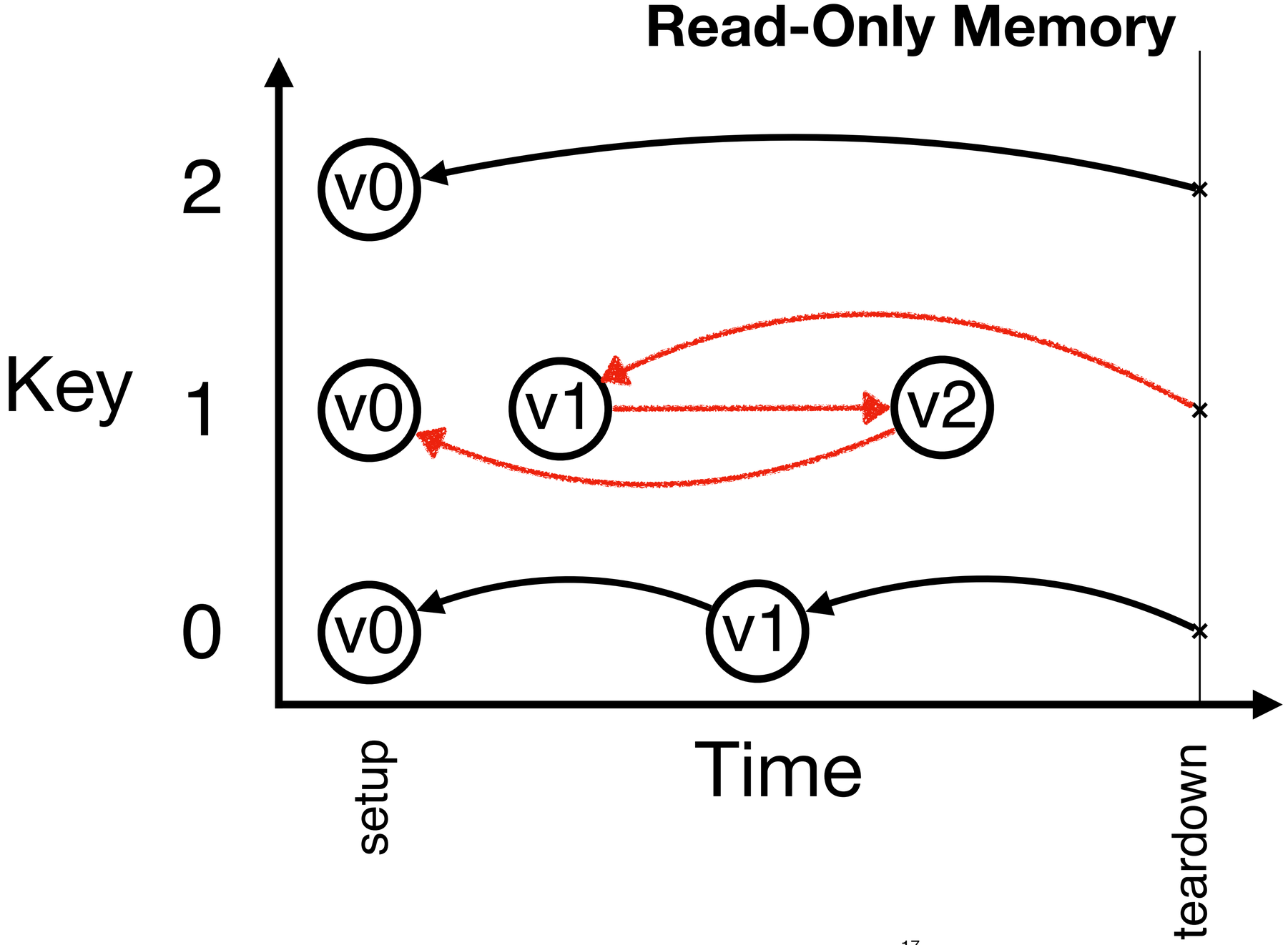


Difference between ROM and RAM:

RAM must prevent P from reading from the future

How?

Instead of “version”, consider “timestamp”. Then, prove the “time difference” is positive.



Difference between ROM and RAM:

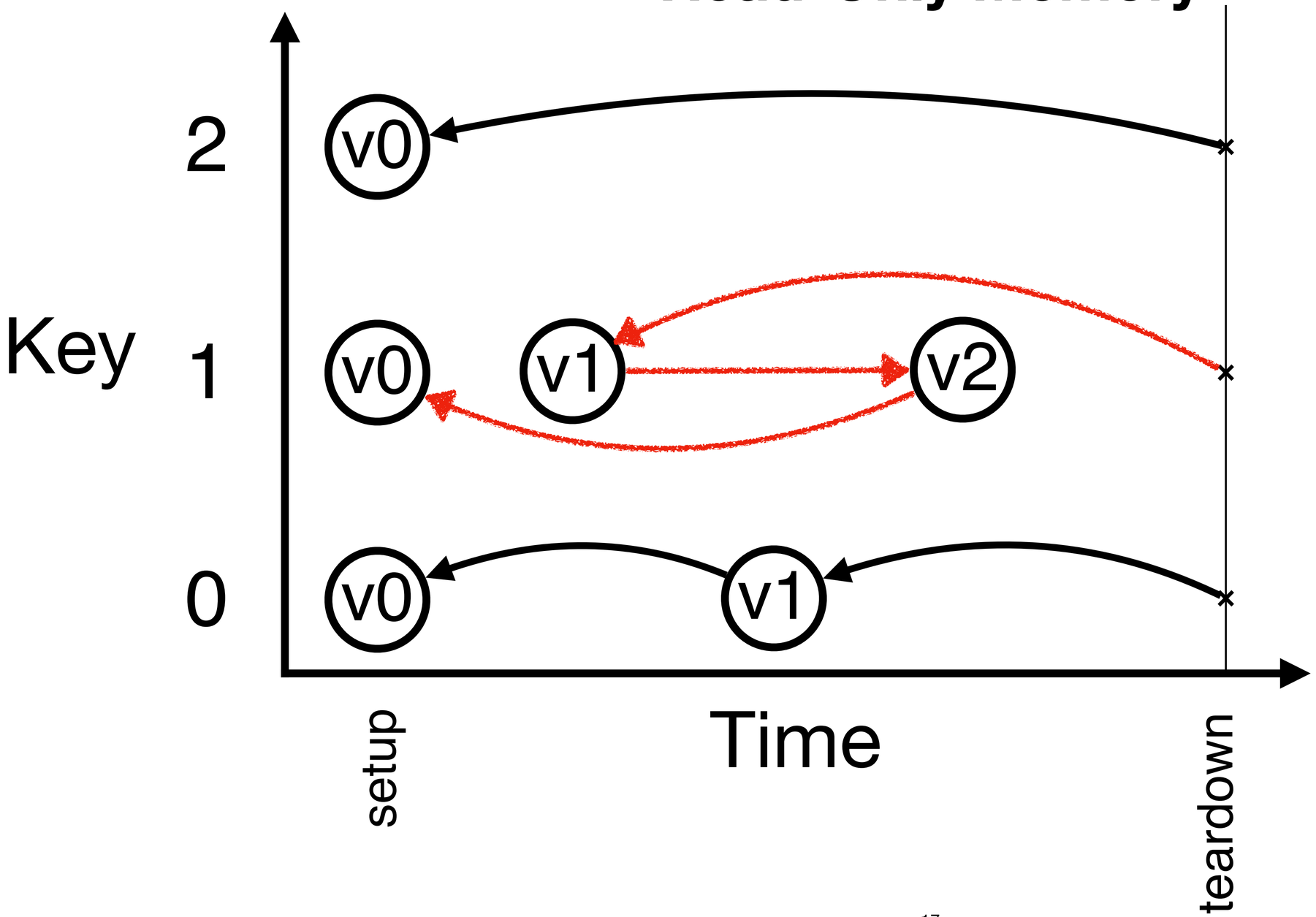
RAM must prevent P from reading from the future

How?

Instead of “version”, consider “timestamp”. Then, prove the “time difference” is positive.

How?

Read-Only Memory



Difference between ROM and RAM:

RAM must prevent P from reading from the future

How?

Instead of “version”, consider “timestamp”. Then, prove the “time difference” is positive.

How?

Hint: A ZK ROM with 1,2,…,T indexes

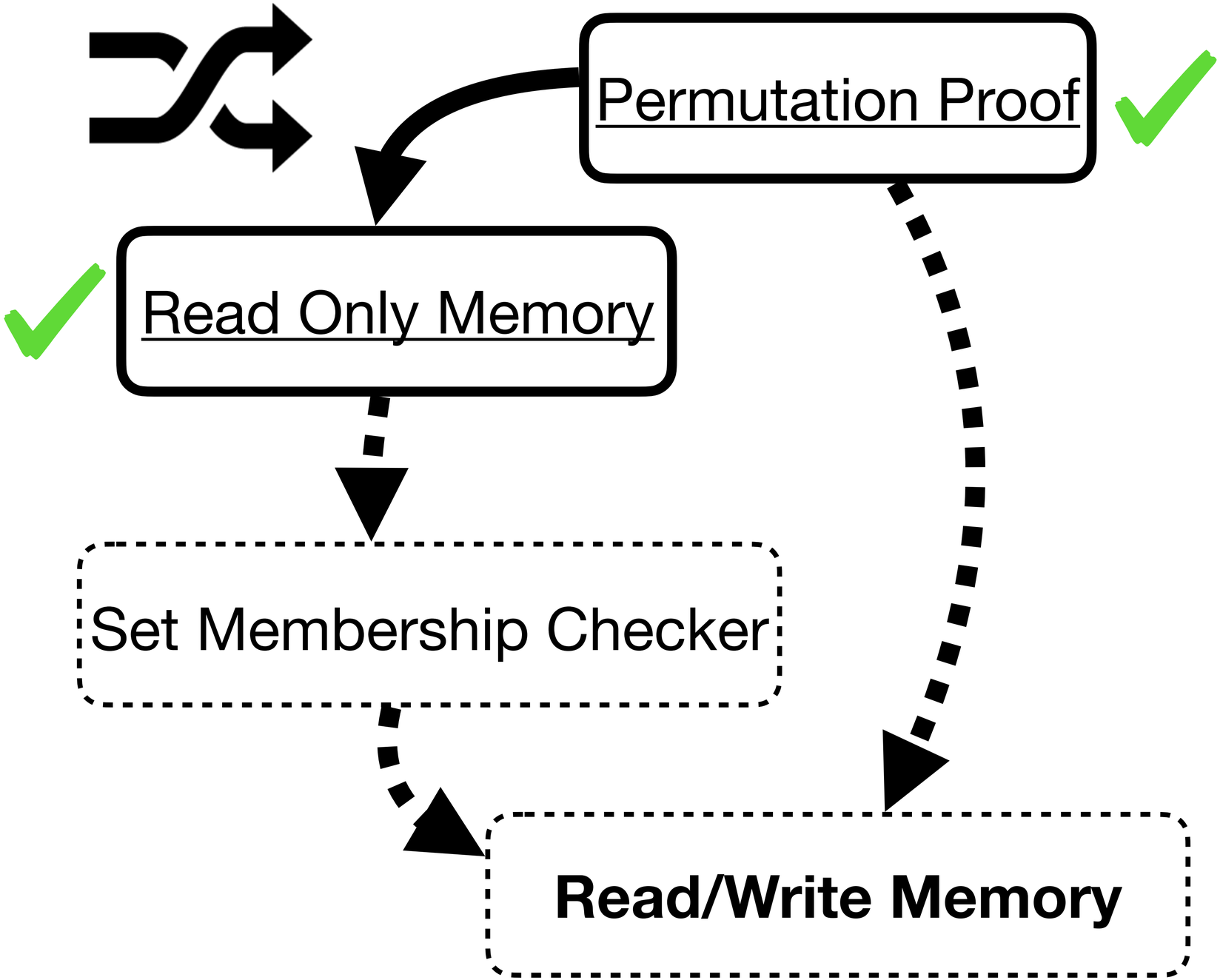
Read-Only Memory

Key	Value
0	x[0]
1	x[1]
2	x[2]

On an access, P gives two inputs: value and version

After T accesses, permutation check on length n+T vectors

2 INPUTs, 2 MULs per access


Permutation Proof
Read Only Memory
Set Membership Checker
Read/Write Memory

Evaluation

We implemented in the VOLE-based ZK setting [DIO21, YSWW21]

~600K random accesses per second on a 1 Gbps LAN

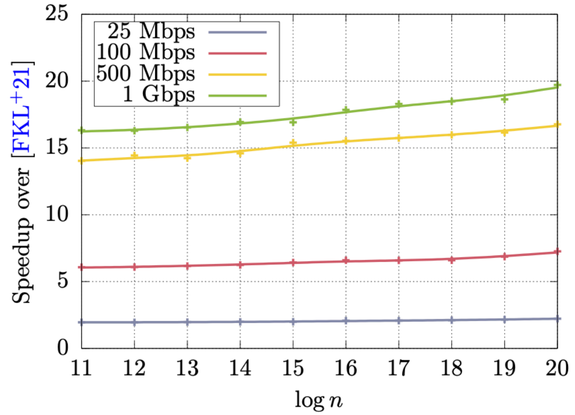


Figure 12: Speedup of our RAM over [FKL+21]'s RAM.

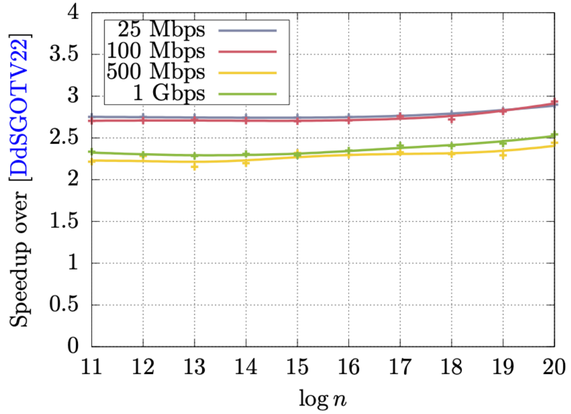


Figure 15: Our RAM's speedup over [DdSGOTV22]'s RAM (we optimize [DdSGOTV22]'s RAM).

Thank you!
Q&A

yyang811@gatech.edu

- ePrint: https://eprint.iacr.org/2023/1115
- GitHub: https://github.com/gconeice/improved-zk-ram