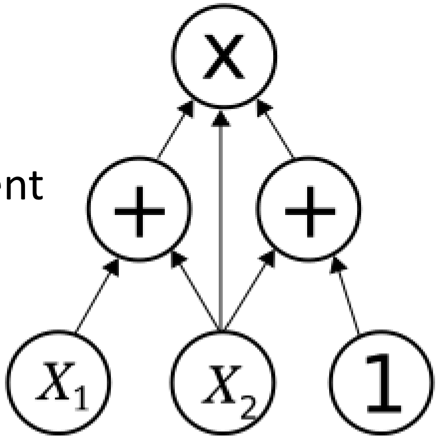
Arya:
Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution

Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, Mary Maller

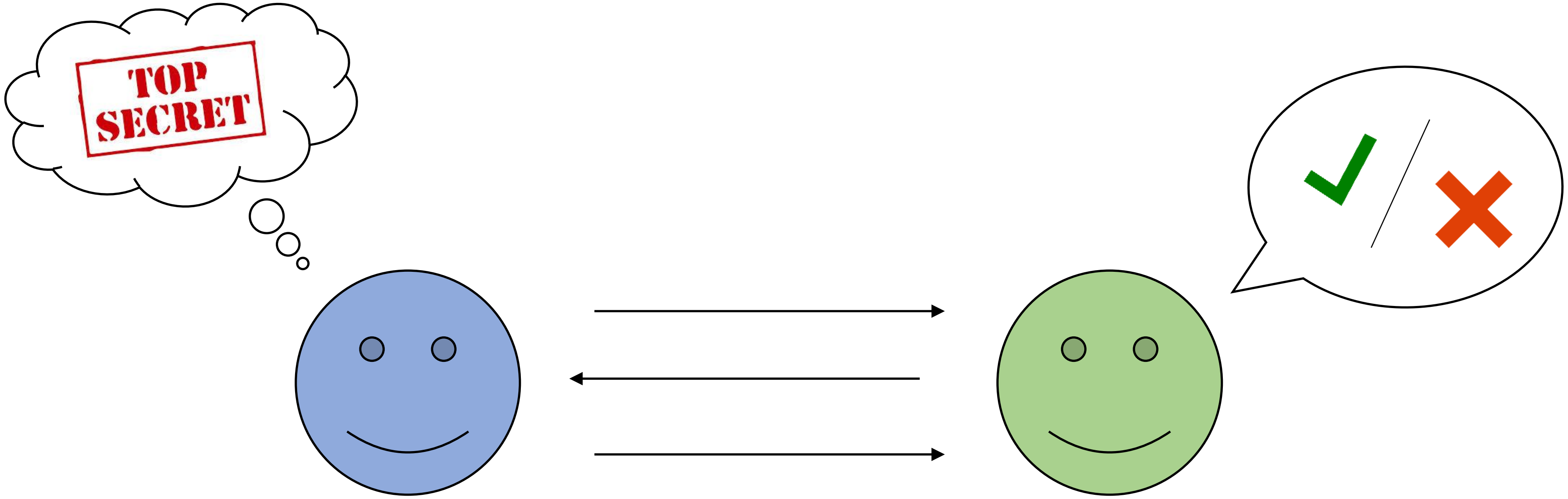
IBM Research | DFINITY | UCL

Zero-Knowledge Proofs for Correct Program Execution

Statement



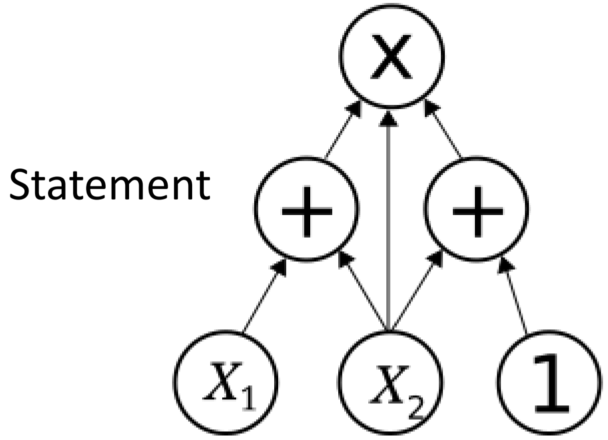
Witness

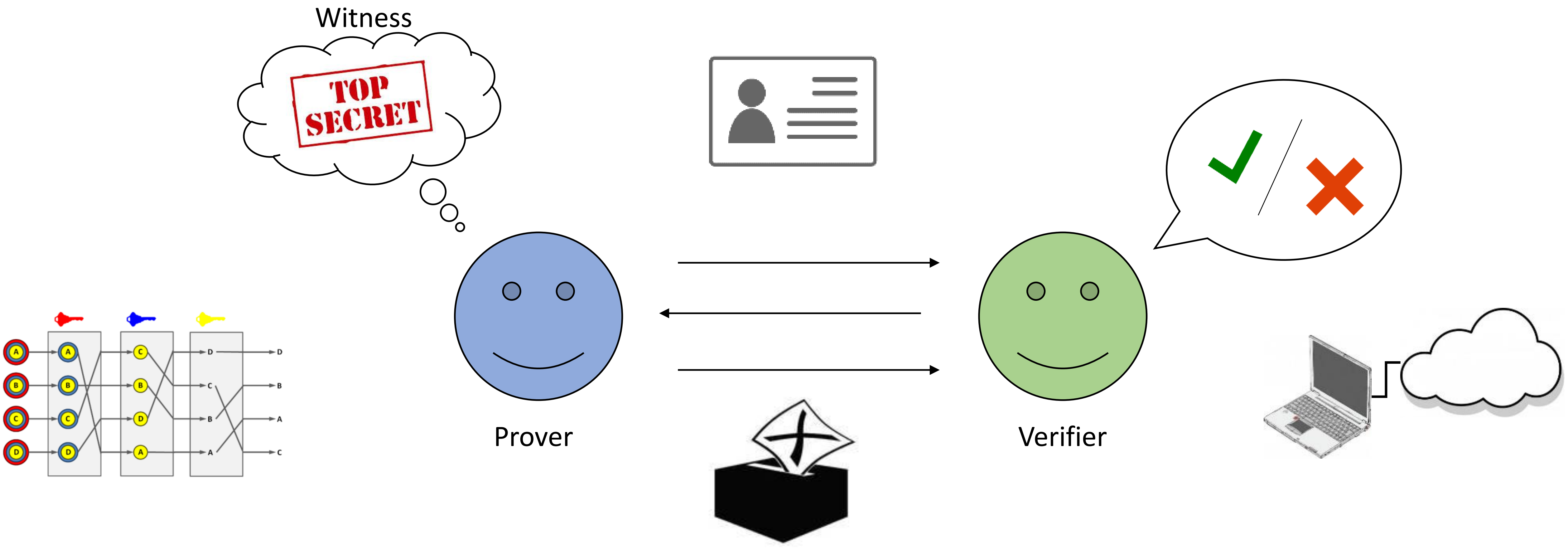


Prover

Verifier

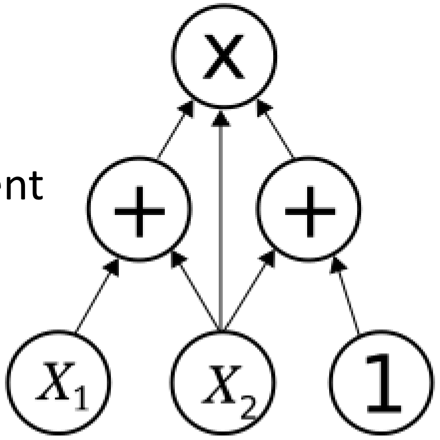
Zero-Knowledge Proofs for Correct Program Execution

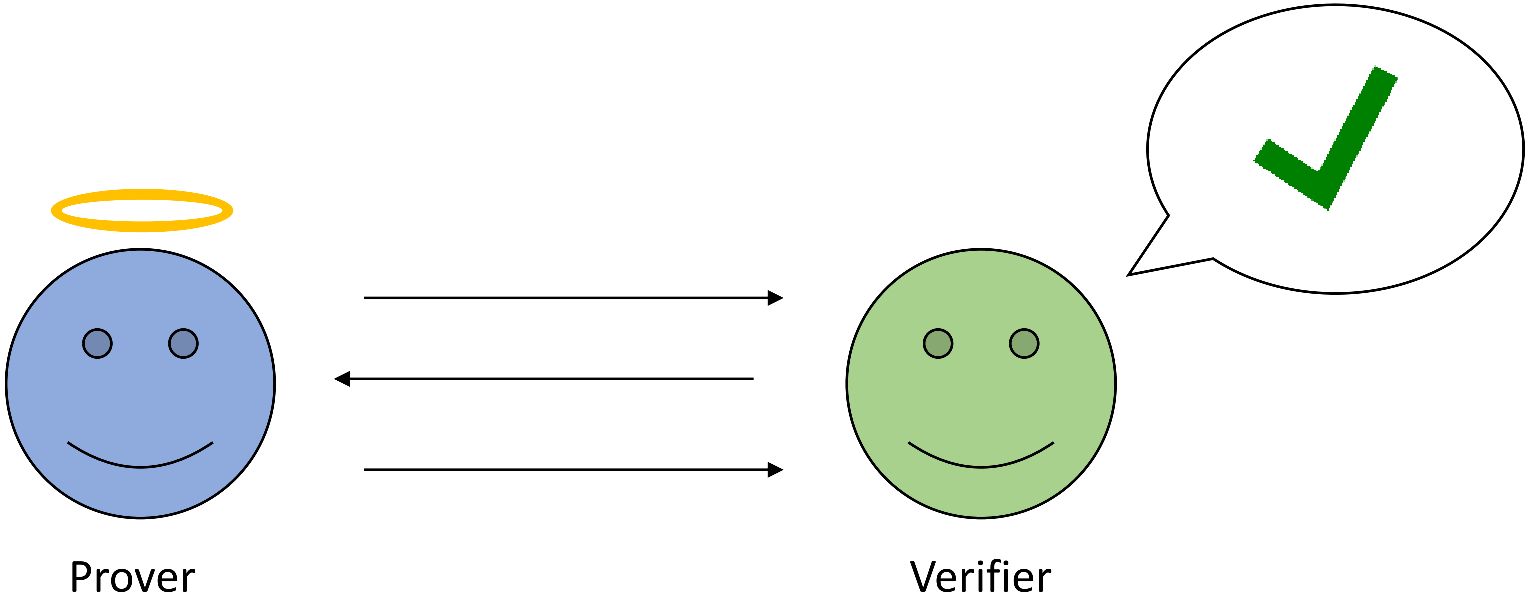




Zero-Knowledge Proofs for Correct Program Execution

Statement

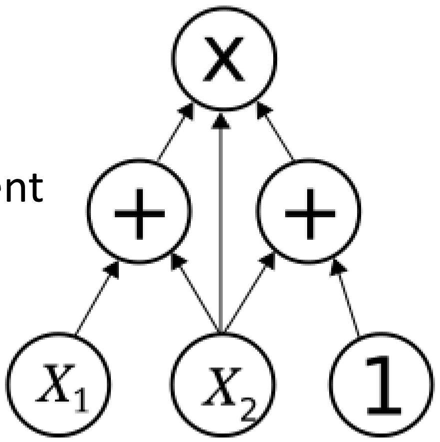


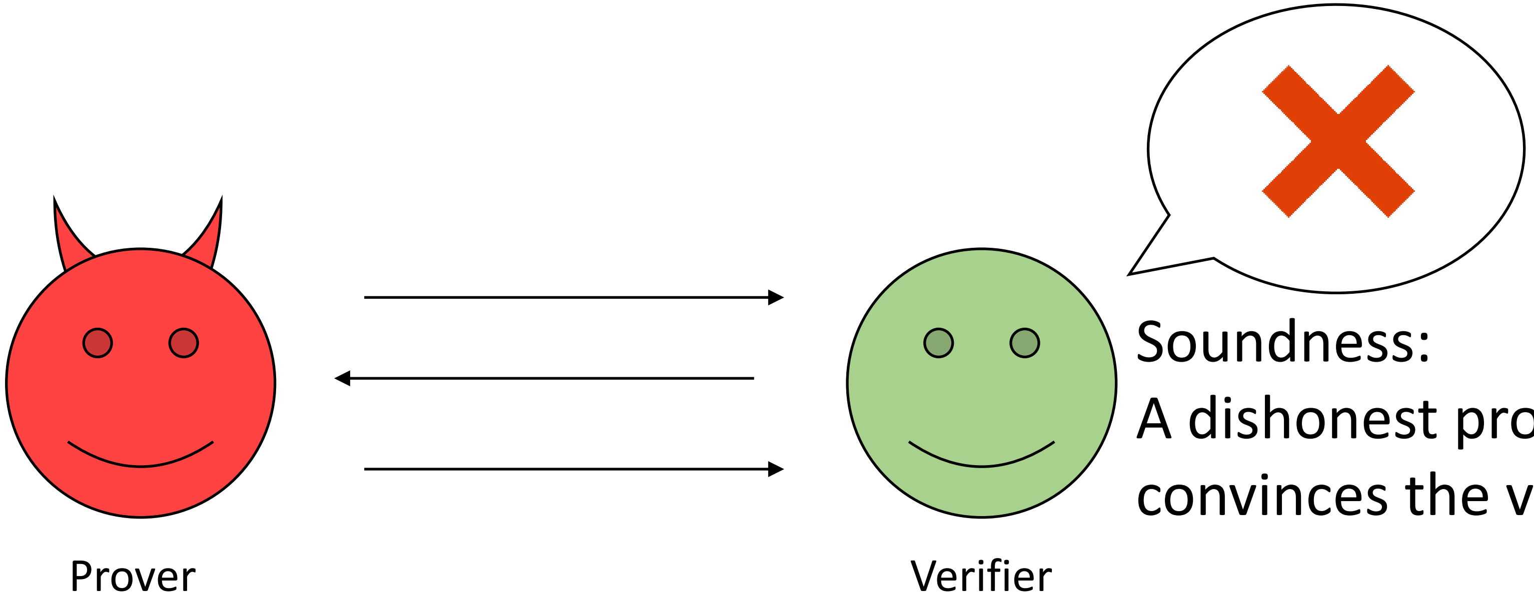


Completeness:
An honest prover convinces the verifier.

Zero-Knowledge Proofs for Correct Program Execution

Statement



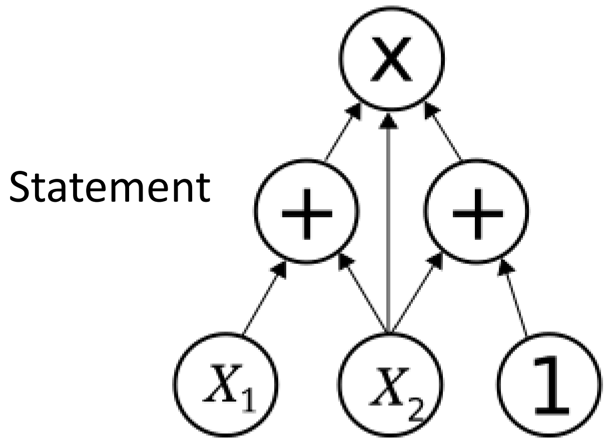


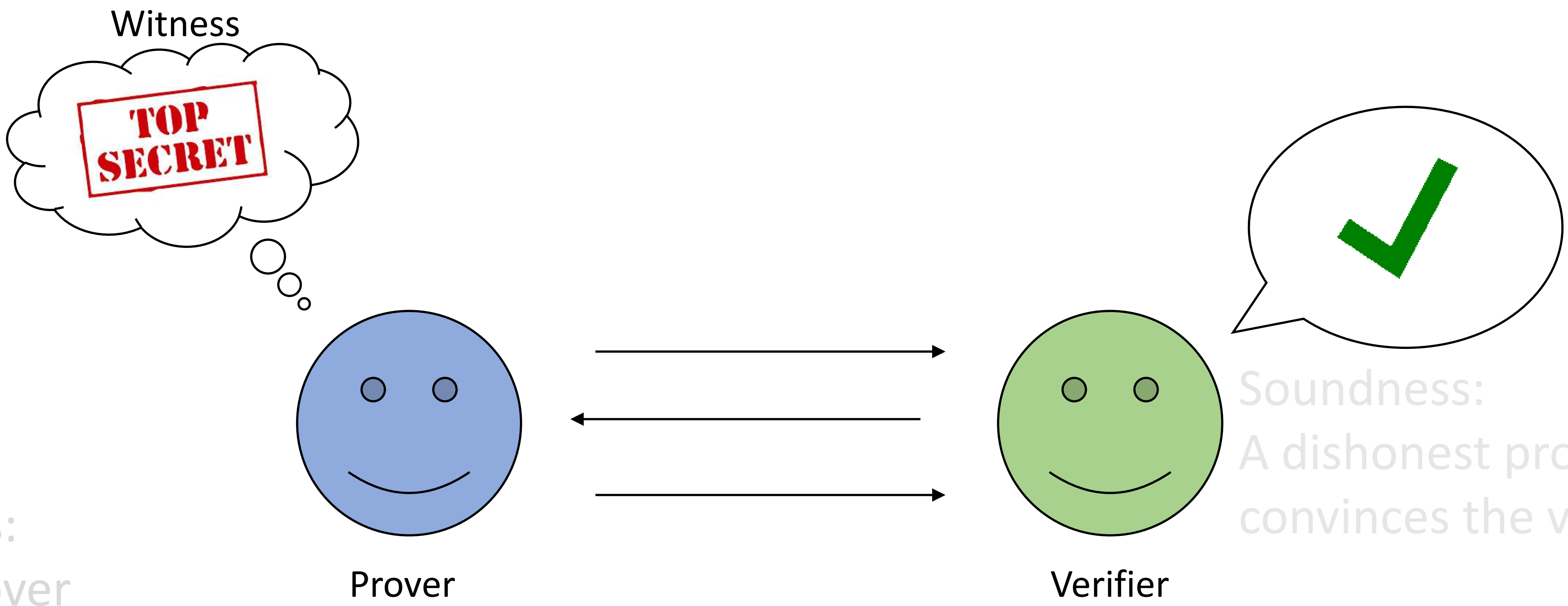
Completeness:
An honest prover convinces the verifier.

Soundness:
A dishonest prover never convinces the verifier.

Computational guarantee
-> argument

Zero-Knowledge Proofs for Correct Program Execution





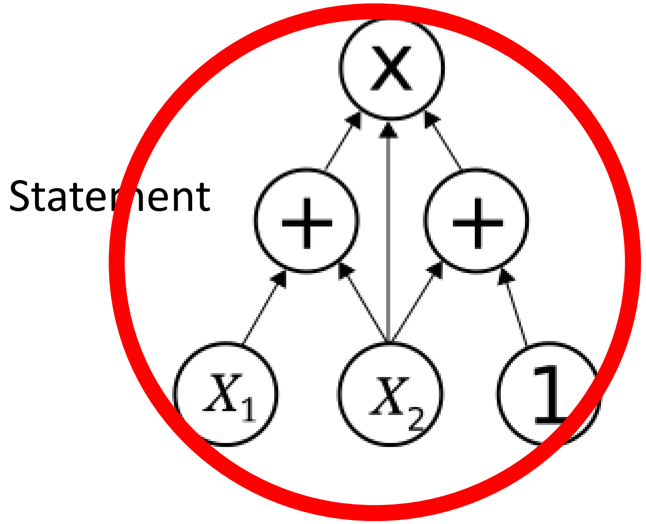
Soundness:
A dishonest prover never convinces the verifier.

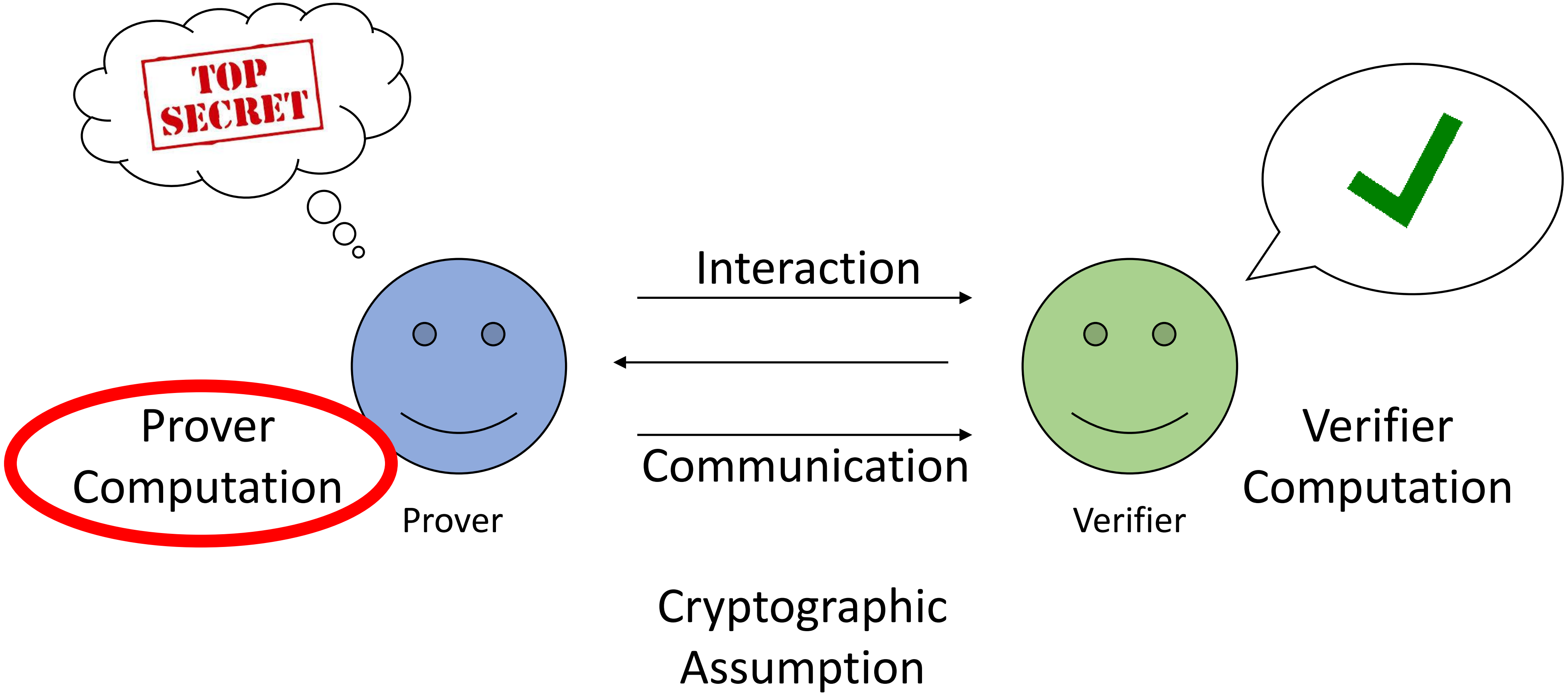
Completeness:
An honest prover convinces the verifier.

Computational guarantee -> argument

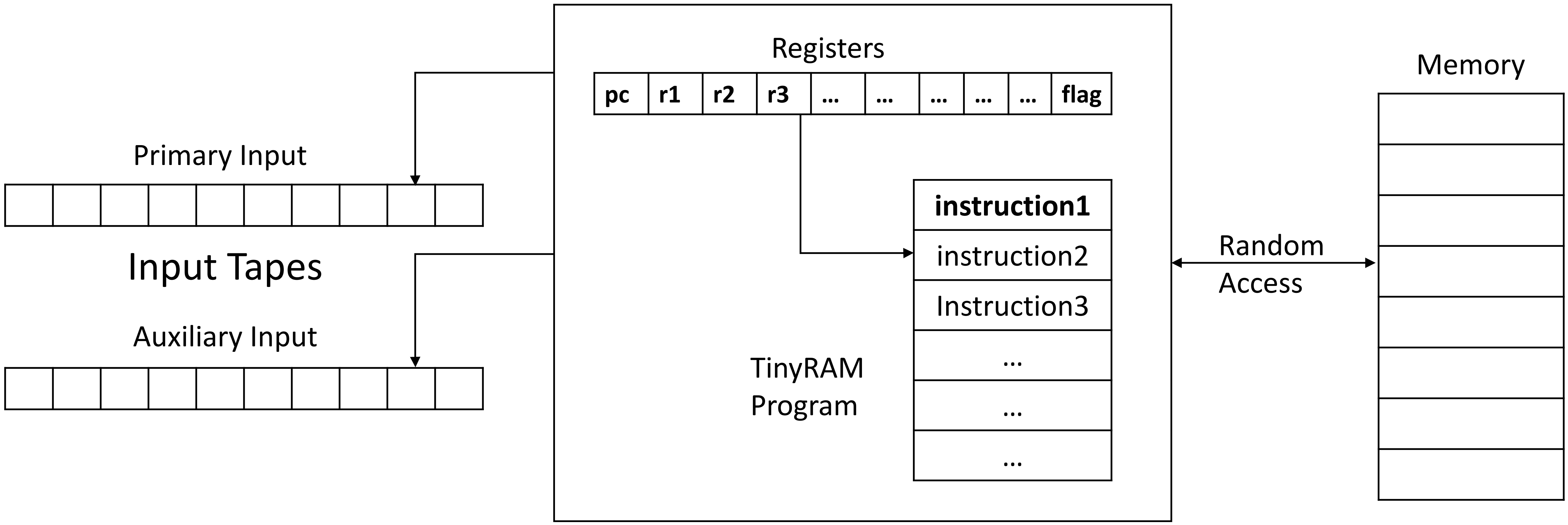
Zero-knowledge:
Nothing but the truth of the statement is revealed.

Zero-Knowledge Proofs for Correct Program Execution





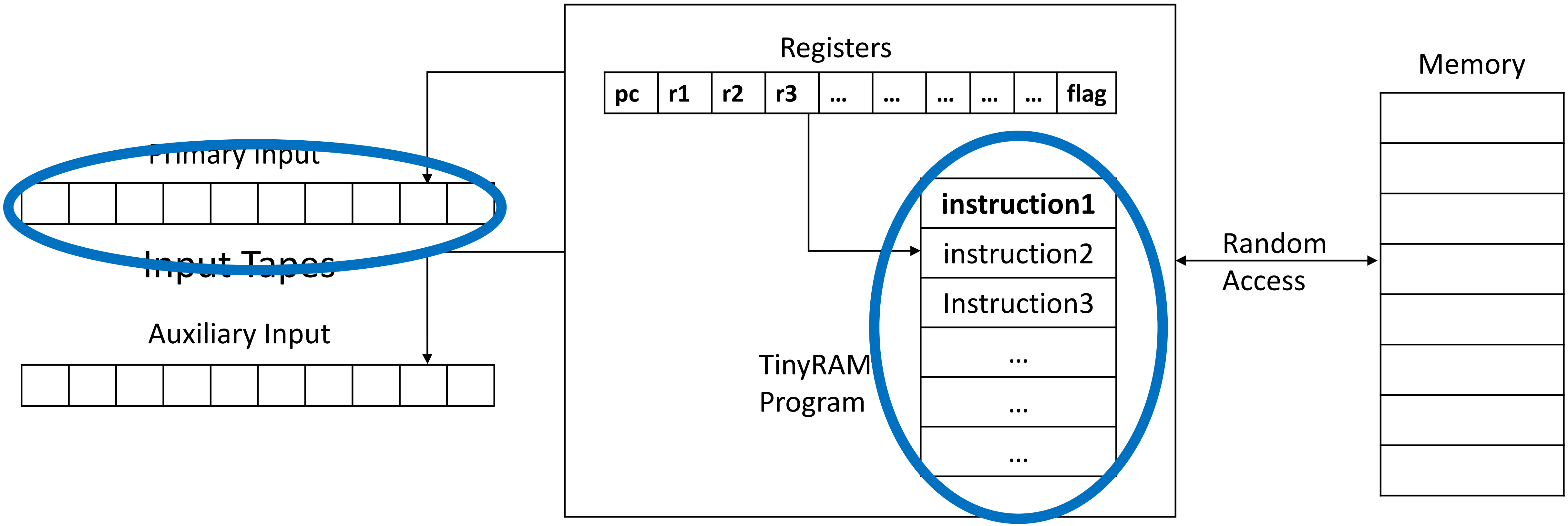
Zero-Knowledge Proofs for
Correct Program Execution



TinyRAM

Instructions include ADD, MULT, XOR, AND,…

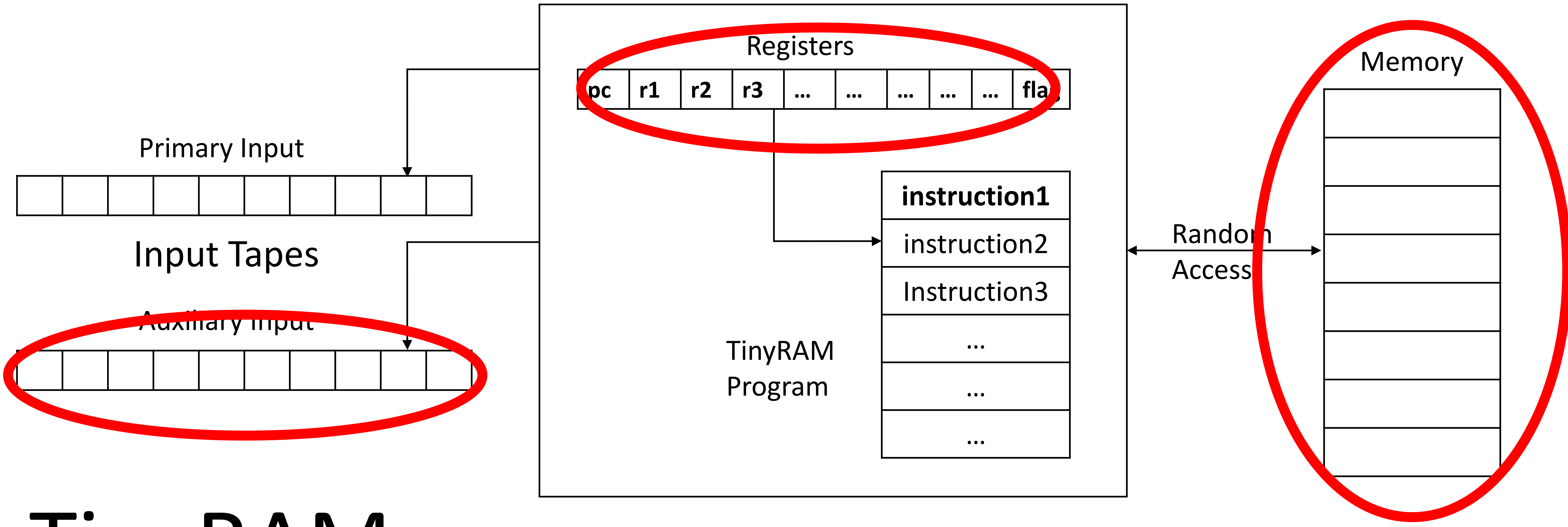
Zero-Knowledge Proofs for
Correct Program Execution



TinyRAM

Public Values <-> Statement

Zero-Knowledge Proofs for
Correct Program Execution



TinyRAM

Private Values <-> Prover's Witness

Zero-Knowledge Proofs for Correct Program Execution

Why TinyRAM?

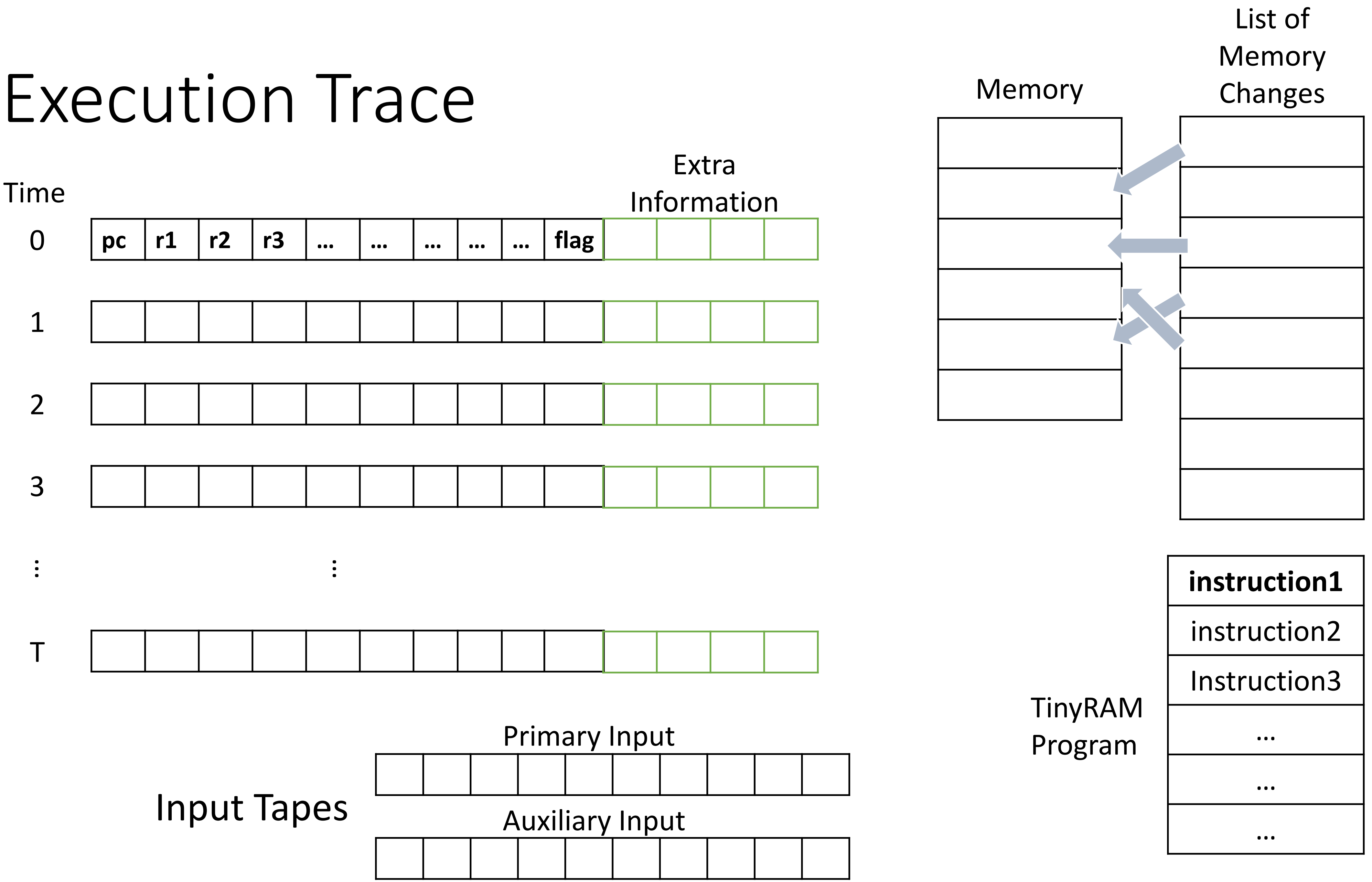
- Closer to real world statements
- Compilers from restricted C to TinyRAM

```c
#include <stdio.h>
#include <unistd.h>

int main ()
{
        printf("Running 'net join' with the following parameters: \n");
        char *domain="mydomain";
        char *user="domainjoinuser";
        char *pass="mypassword";
        char *vastool="/opt/quest/bin/vastool";
        char *ou="OU=test,DC=mtdomian,DC=local";
        char unjoin[512];

        sprintf(unjoin,"/opt/quest/bin/vastool -u %s -w '%s' unjoin -f",user,pass);

        printf("Domain: %s\n",domain);
        printf("User: %s\n",user);
        printf("-----------------\n");

        printf("\nUnjoin...........\n");
        system(unjoin);

        printf("\nJoin..............\n");
        execl("/opt/quest/bin/vastool", "vastool", "-u", user, "-w", pass, "join", "-c", ou,
"-f", domain, (char*)0);

}
```

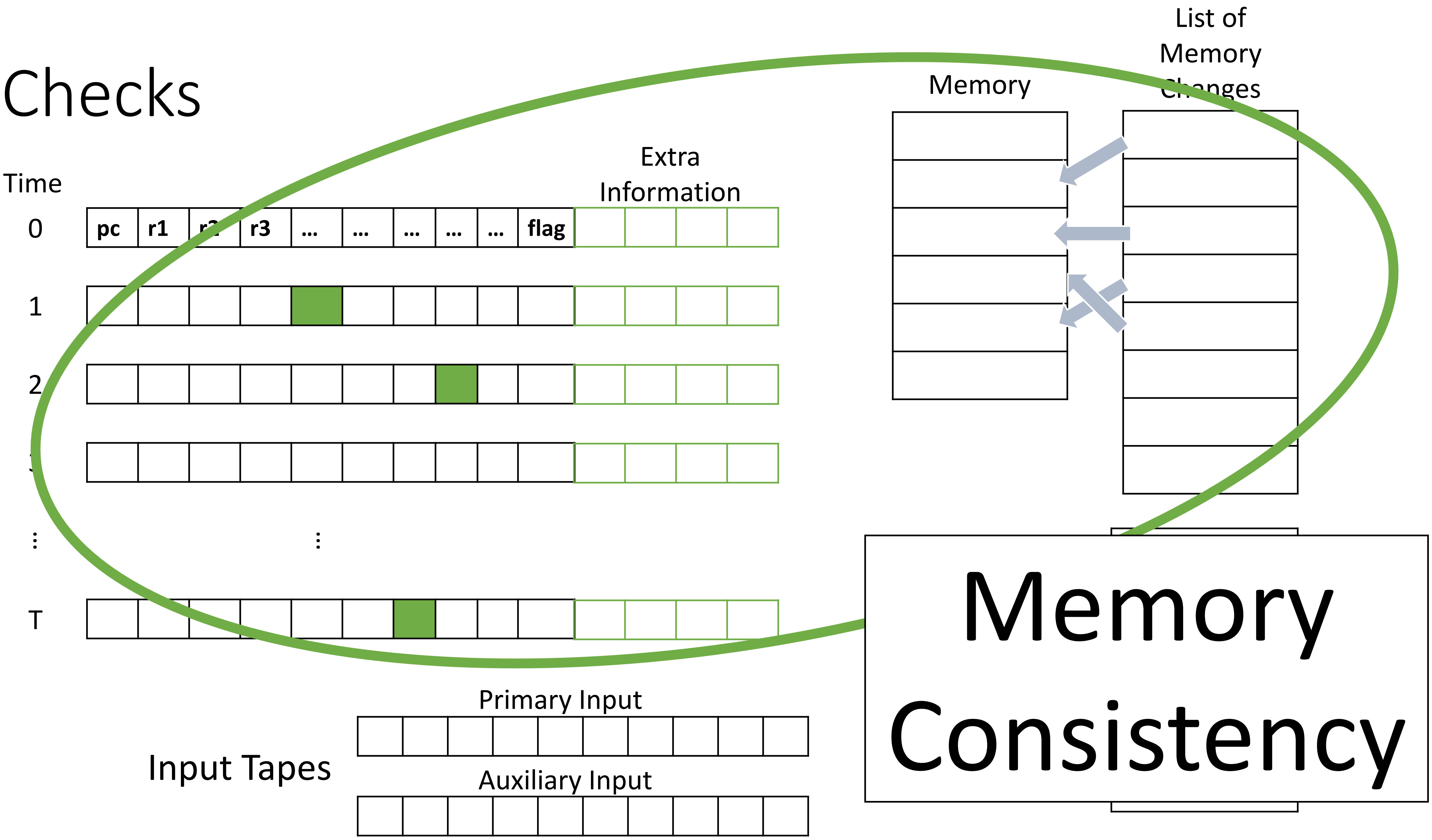
Zero-Knowledge Proofs for Correct Program Execution

Goal:
Zero-knowledge proof for correct TinyRAM execution with low prover overhead

Execution Trace

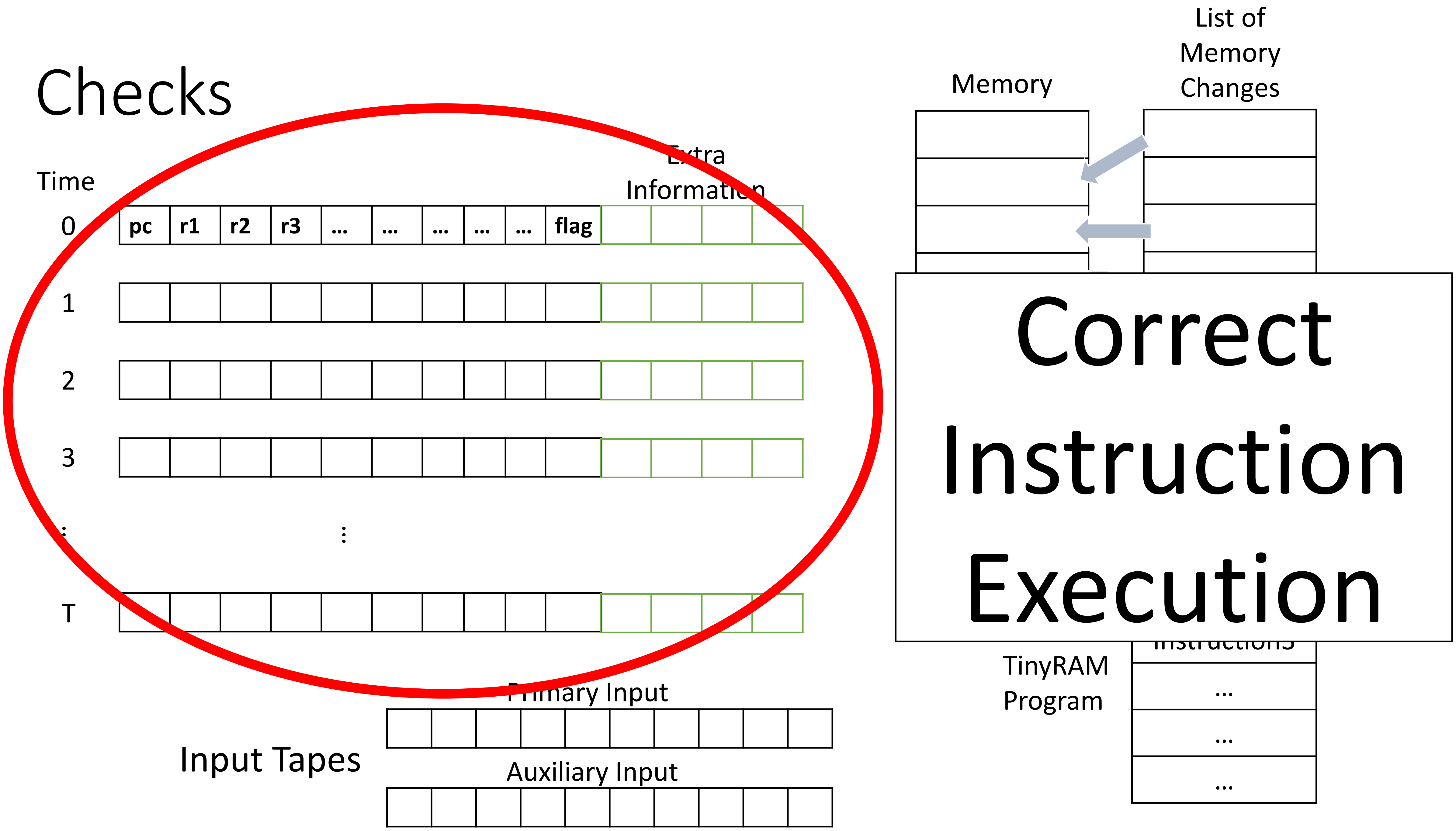


Checks



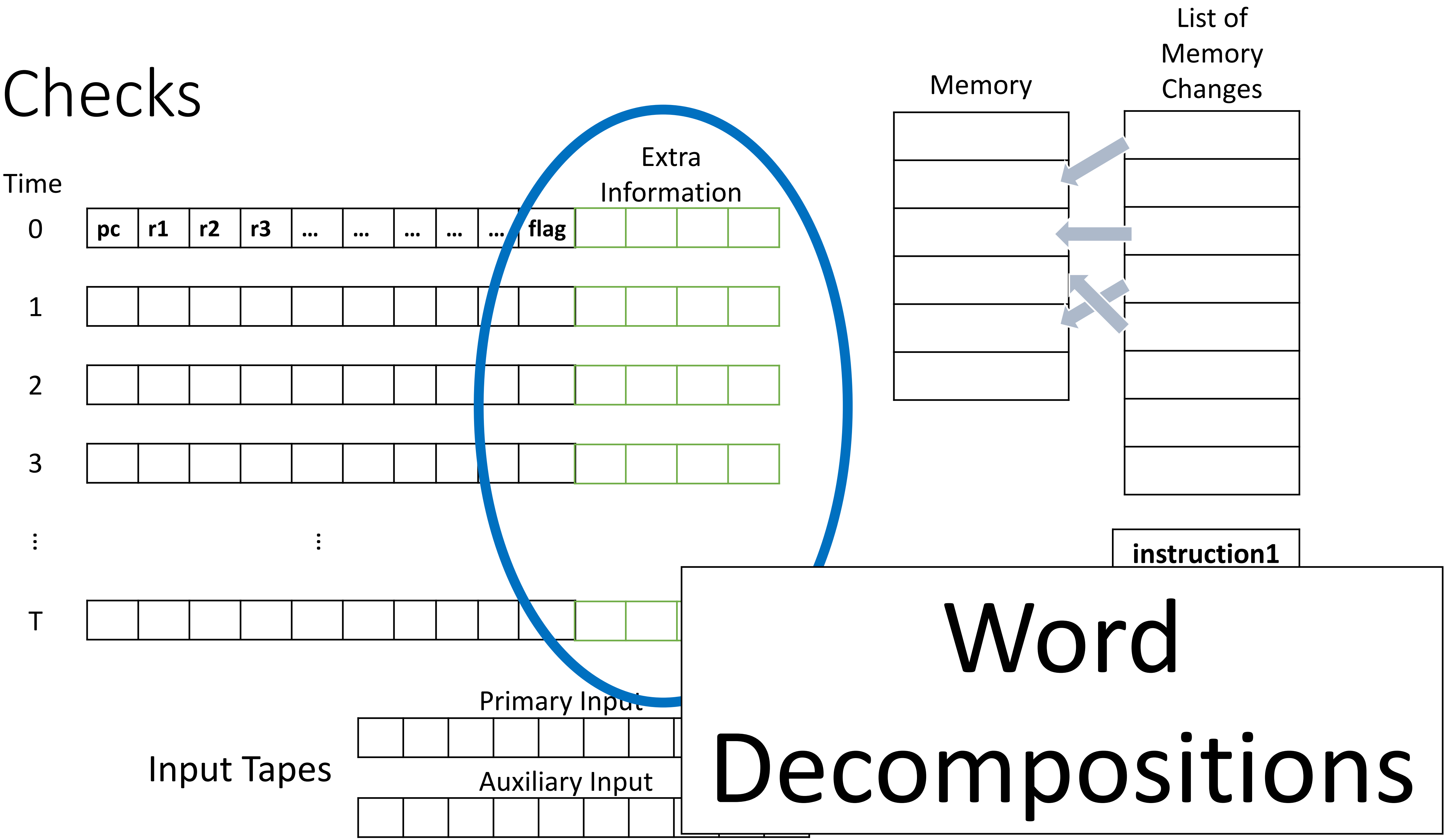
Time
0
1
2
T
pc
r1
r3
...
flag
Extra Information
Memory
List of Memory Changes
Primary Input
Auxiliary Input
Input Tapes
Memory Consistency

Checks



Time
0
1
2
3
T
pc
r1
r2
r3
...
flag
Extra Information
Primary Input
Auxiliary Input
Input Tapes
Memory
List of Memory Changes
TinyRAM Program
...
Correct Instruction Execution

Checks



Word Decompositions

Proving Correct Program Execution

Sources of Overhead

- Large fields and large cyclic groups
- Permutation networks for checking memory
- Large circuits for bitwise operations

Our Solutions

- Use hash-based proof system over any field
- Alternative approach to checking permutations
- Word decomposition technique gives constant-size circuits

Results

Work	Prover Complexity	Verifier Complexity	Communication	Rounds	Assumption
BCTV14	$\Omega(T(\log T)^2)$	$\omega(L + \lvert v \rvert)$	$\omega(1)$	1	KoE
This Work	$O(\alpha T)$	$poly(\lambda)(\sqrt{T} + L + \lvert v \rvert)$	$poly(\lambda)(\sqrt{T} + L)$	$O(\log\log T)$	LT-CRHF

Security $2^{-\omega(\log \lambda)}$

Program Length L

Runtime Bound T

Public Input V

Overview

TinyRAM Execution Trace

Algebraic Constraints

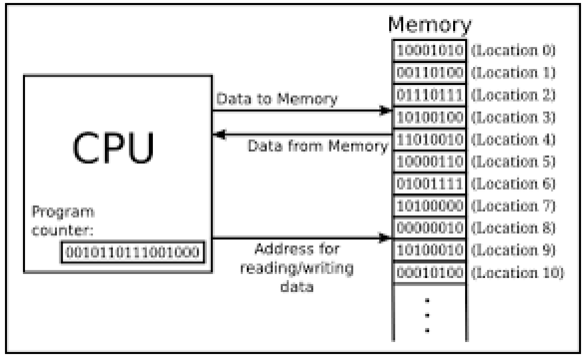
Polynomials

Ideal Protocols

Standard Protocols

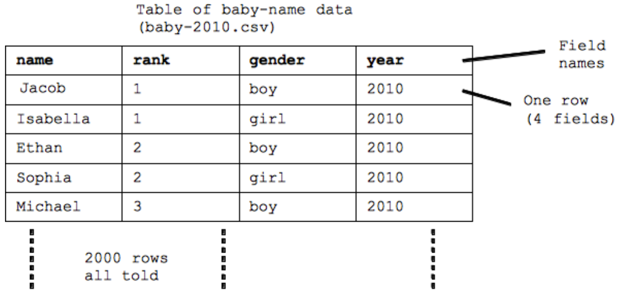
Prior work: Linear-Time Zero-Knowledge Proofs

Arithmetising TinyRAM

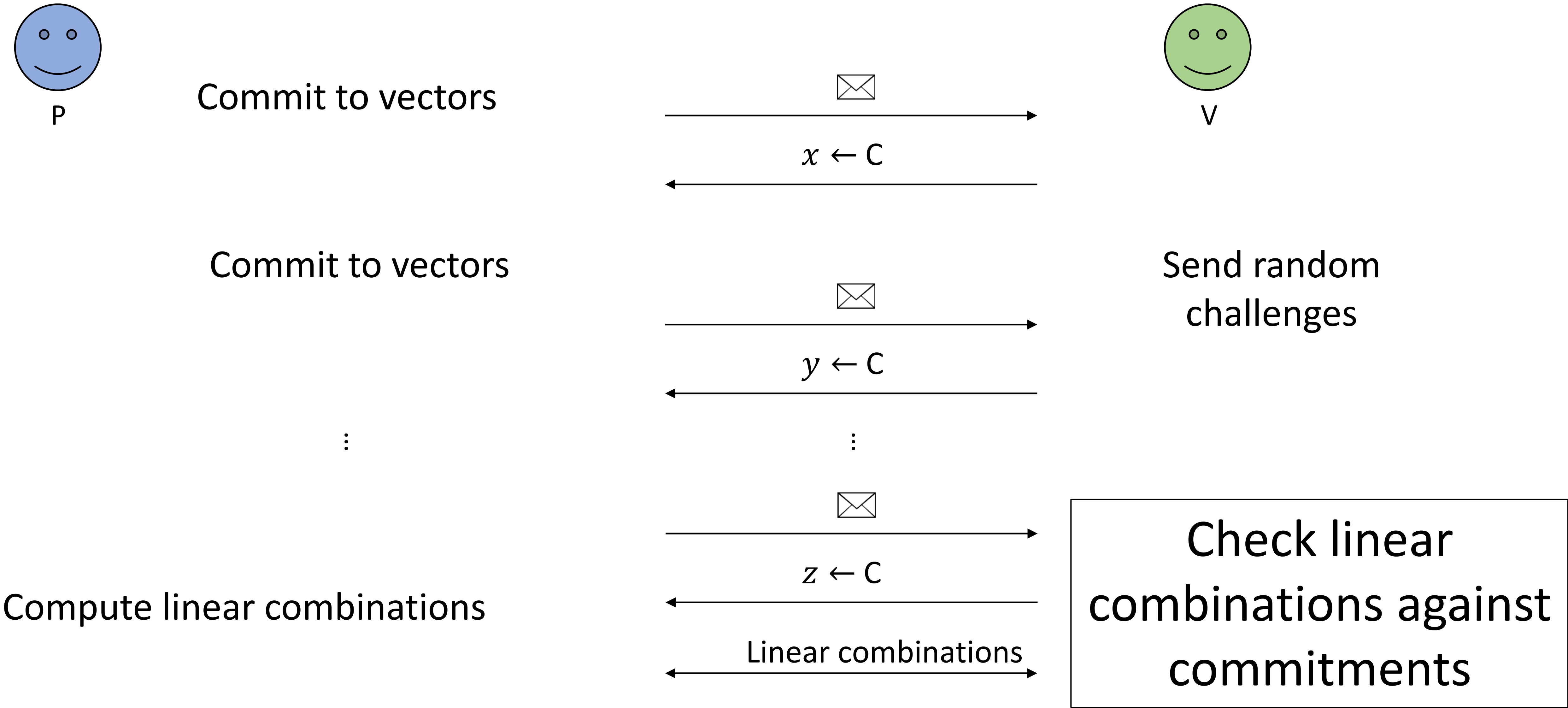


Main Contributions

Zero-Knowledge Look-Up Arguments and More



Ideal Linear Commitment Protocols



Ideal Linear Commitment Protocols

P

V

Commit to execution trace

$x \leftarrow C$

Commit to vectors

Send random challenges

$y \leftarrow C$

⋮

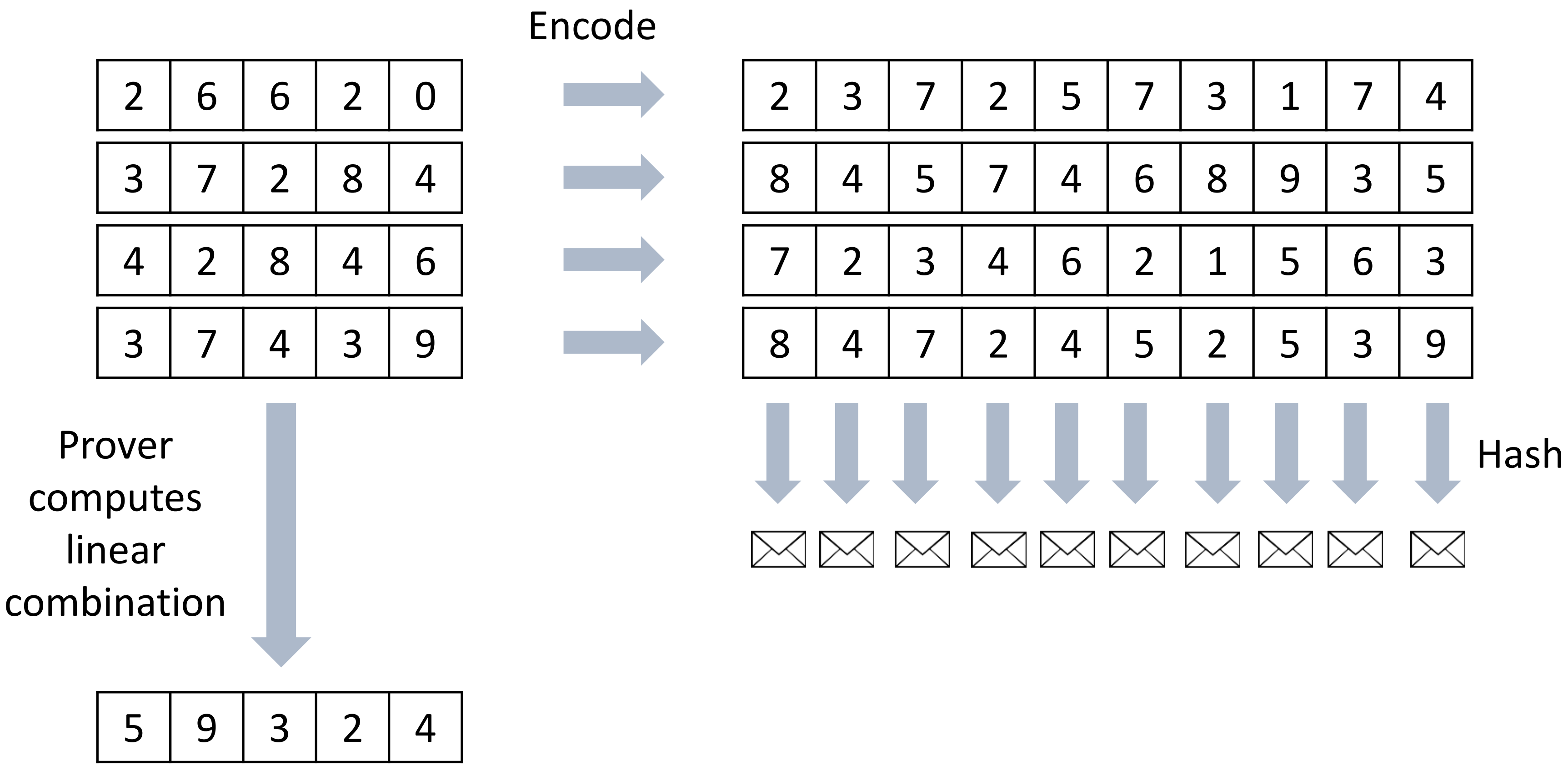
$z \leftarrow C$

Compute linear combinations

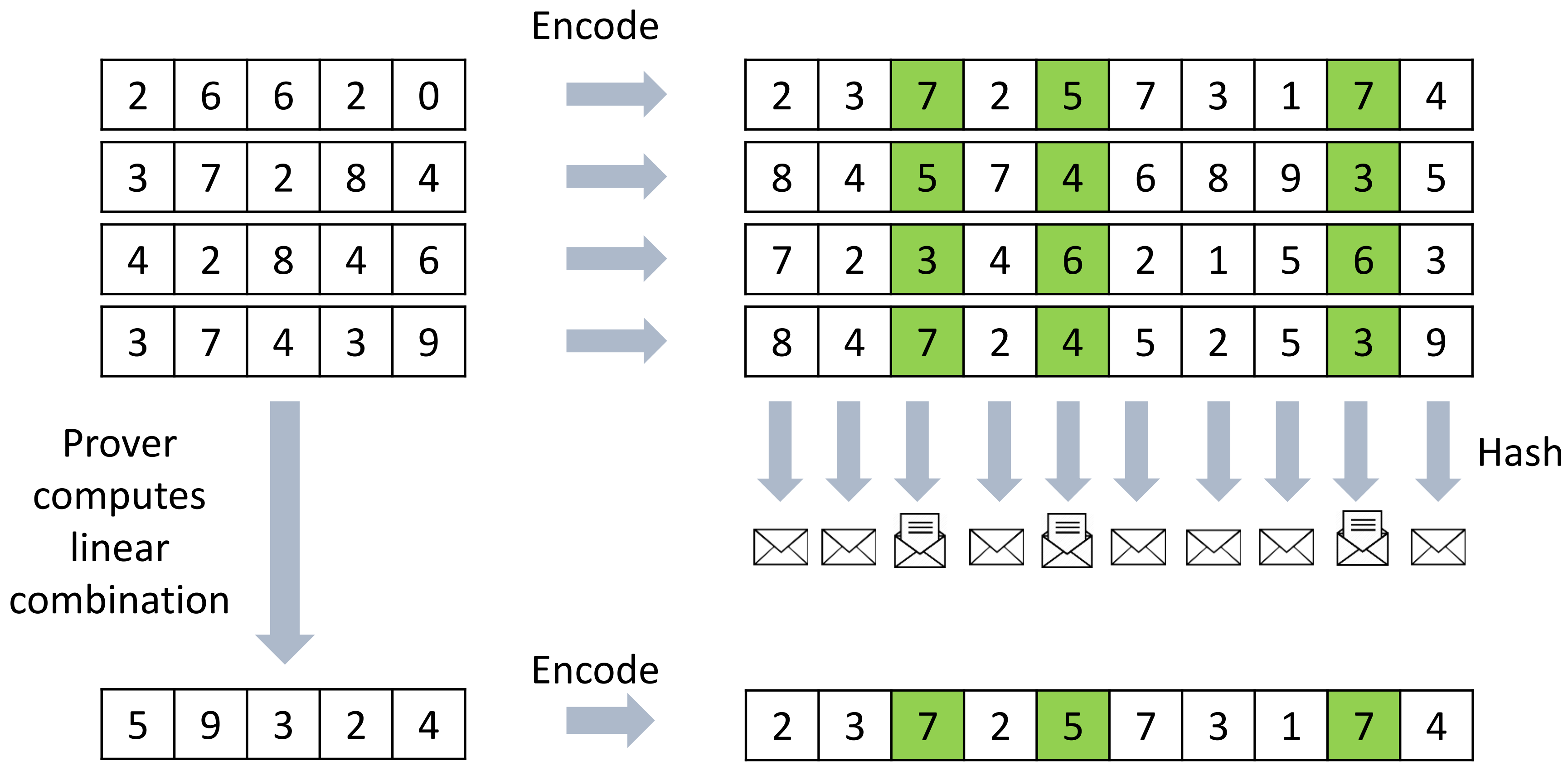
Linear combinations

Coefficients of linear combinations embed useful conditions

Committing


Encode
2 6 6 2 0
3 7 2 8 4
4 2 8 4 6
3 7 4 3 9
2 3 7 2 5 7 3 1 7 4
8 4 5 7 4 6 8 9 3 5
7 2 3 4 6 2 1 5 6 3
8 4 7 2 4 5 2 5 3 9
Hash
Prover computes linear combination
5 9 3 2 4

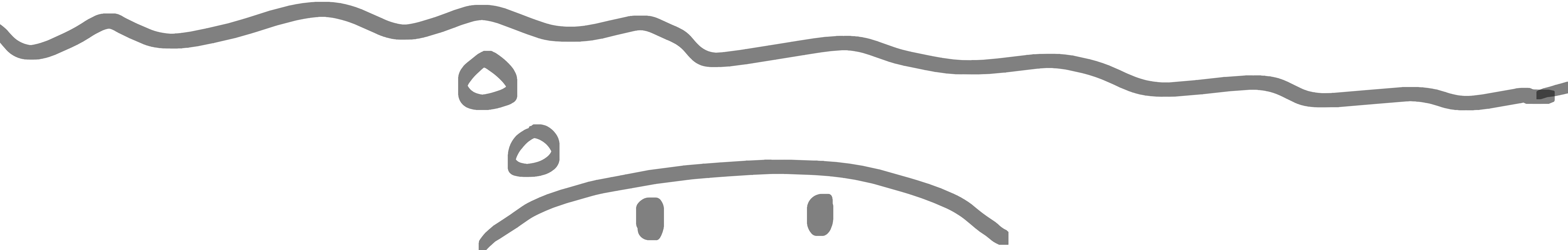
Checking Commitments



Verifier encodes and spot-checks columns
High minimum distance catches cheating

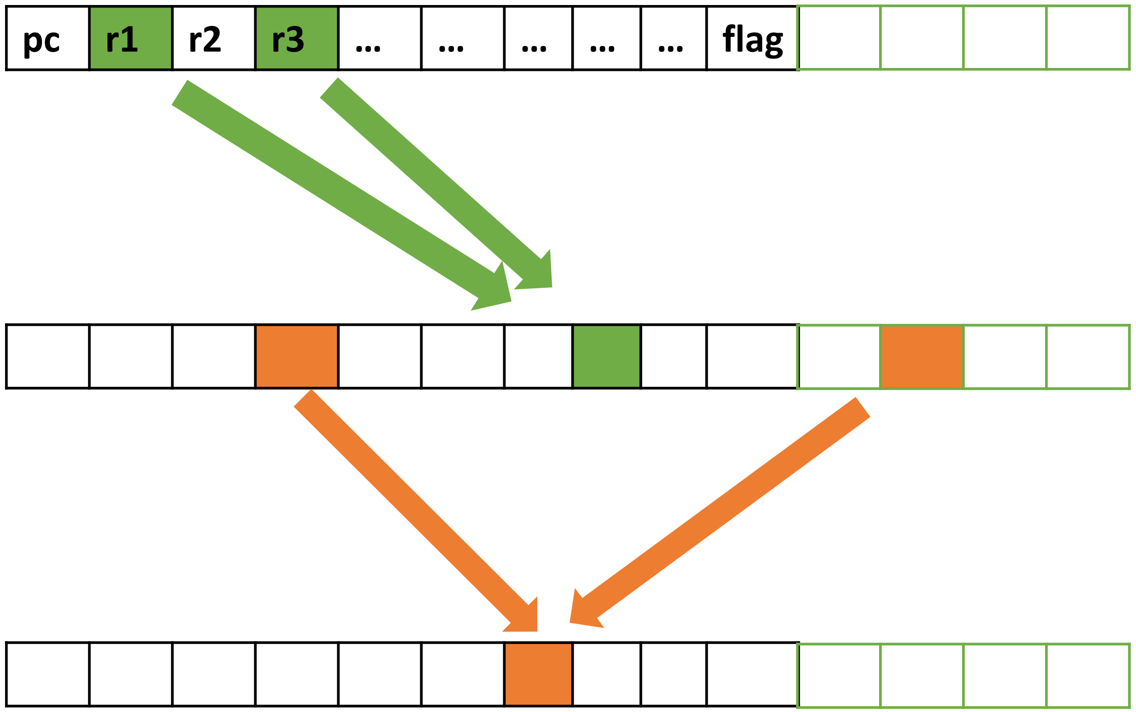
Linear-time Zero-Knowledge Arguments (2017)

= Efficient Ideal Protocols + Linear-Time Encodable Error-Correcting Codes + Linear-Time Computable Hash Functions



Correct Instruction Execution

Check consistency of values across each time step



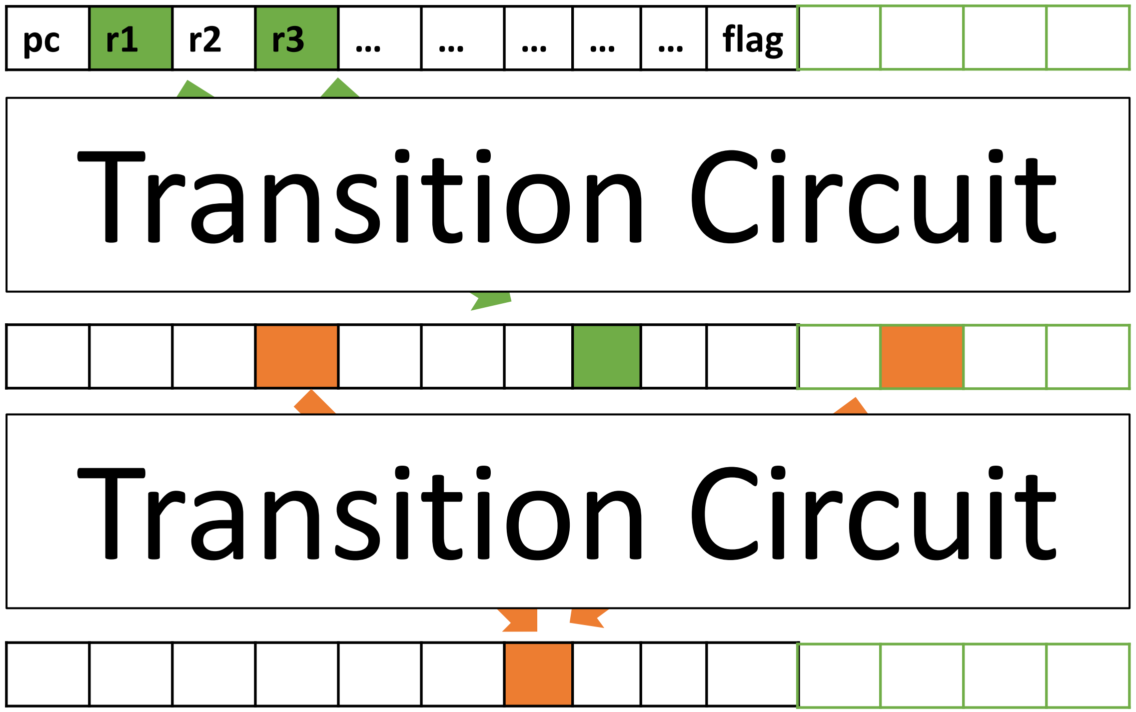
Covers all TinyRAM instructions

Constant size circuit

Give batch argument that each copy of circuit is satisfied

Correct Instruction Execution

Check consistency of values across each time step



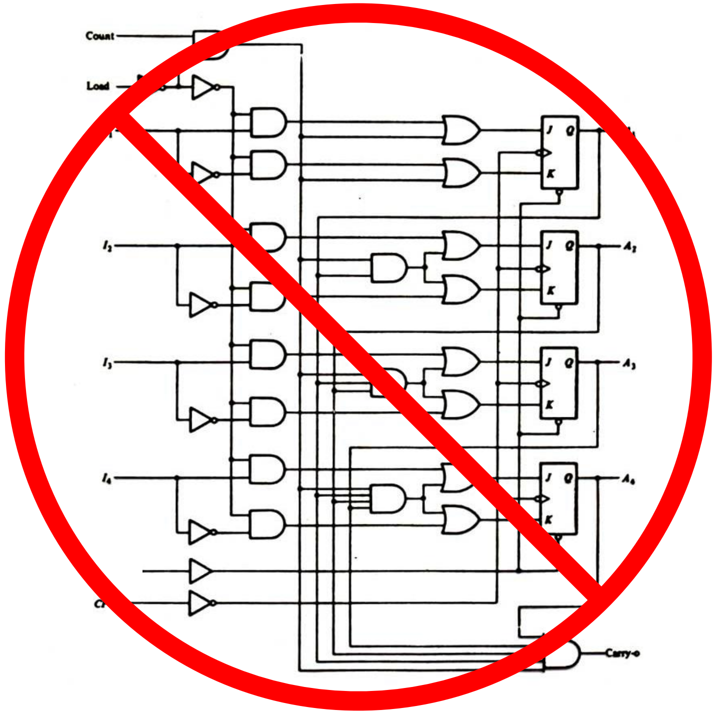
Covers all TinyRAM instructions

Constant size circuit

Give batch argument that each copy of circuit is satisfied

Word Decomposition

Avoid binary circuits when checking bitwise operations on non-binary field elements!



$$a, b \in \{0,1\}$$

$$a + b = 2(a \wedge b) + (a \oplus b)$$

Word Decomposition

Register value	Binary Decomposition										
a	a_0	a_1	a_2	a_3	…	…	…	…	a_{W-2}	a_{W-1}	
a_O	a_1	0	a_3	0	…	…	…	…	a_{W-1}	0	Odd bits
a_E	a_0	0	a_2	0	…	…	…	…	a_{W-2}	0	Even bits

$$a = 2a_O + a_E$$

Word Decomposition

a	a_0	a_1	a_2	a_3	…	…	…	…	a_{W-2}	a_{W-1}
a_O	a_1	0	a_3	0	…	…	…	…	a_{W-1}	0
a_E	a_0	0	a_2	0	…	…	…	…	a_{W-2}	0

$$a = 2a_O + a_E$$

b	a_0	a_1	a_2	a_3	…	…	…	…	a_{W-2}	a_{W-1}
b_O	a_1	0	a_3	0	…	…	…	…	a_{W-1}	0
b_E	a_0	0	a_2	0	…	…	…	…	a_{W-2}	0

$$b = 2b_O + b_E$$

XORs in even bits	$a_0 \oplus b_0$	$a_0 \wedge b_0$	$a_2 \oplus b_2$	$a_2 \wedge b_2$	…	…	…	…	$a_{W-2} \oplus b_{W-2}$	$a_{W-2} \wedge b_{W-2}$	ANDs in odd bits

$$a_E + b_E$$

Look-up Argument

Register Value	Even Bits	Odd Bits

Decomposition Look-Up Table

Register Values	Even Bits	Odd Bits

All possible register values

Use zero-knowledge look-up argument to show all decompositions correct

Look-up Argument

Values $a_1, a_2, \dots, a_m$ lie in table

$$\Leftrightarrow$$

$$\{a_1, a_2, \dots, a_m\} \subset \{b_1, b_2, \dots, b_n\}$$

$$\Leftrightarrow$$

$$\prod_{i=1}^{m}(X - a_i) = \prod_{j=1}^{n}(X - b_j)^{e_j}$$

for some $e_j \geq 0$

Look-Up Table

$\boldsymbol{b_1}$
$\boldsymbol{b_2}$
$\boldsymbol{b_3}$
…
…
…
…
$\boldsymbol{b_n}$

$b_1, b_2, \dots, b_n$ already public

Verify a 'square and multiply' algorithm in zero-knowledge

Think of $m \gg n$

Look-up Argument

Approach:

$b_1, b_2, \dots, b_n$ already public

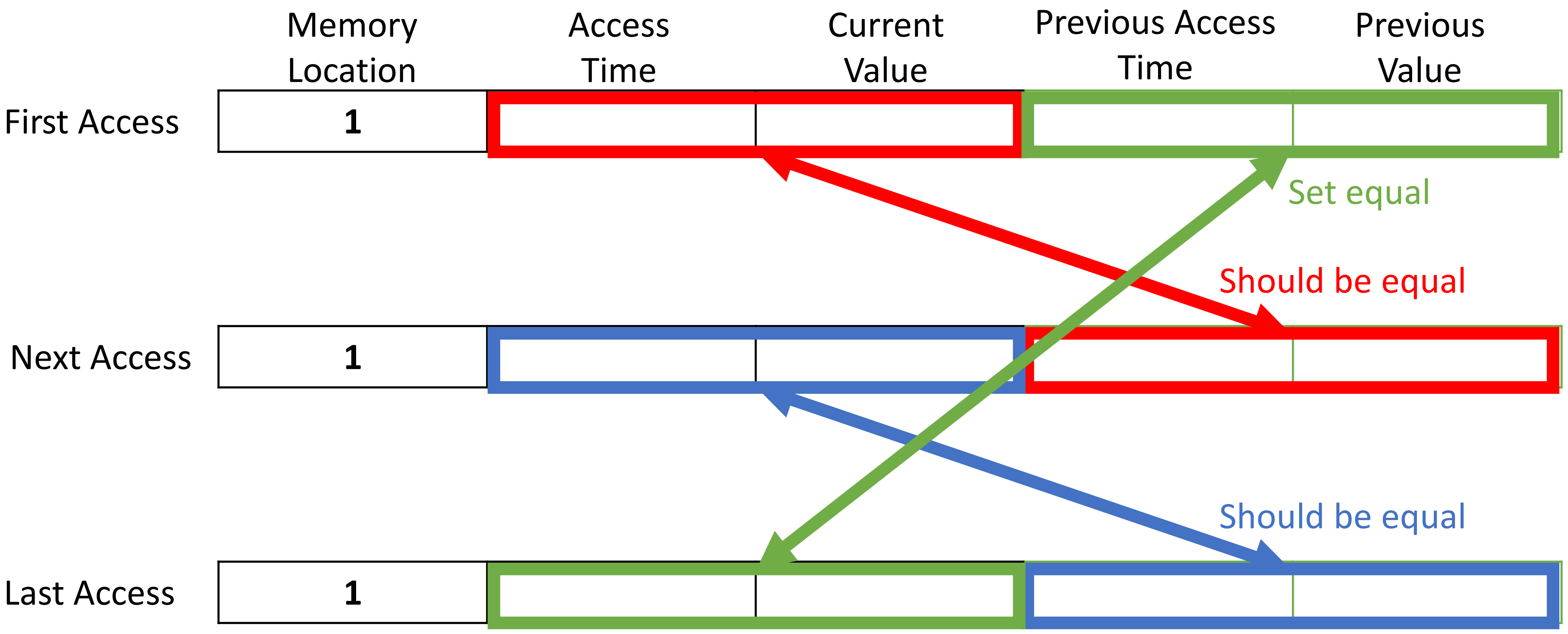
1. Commit to $a_1, a_2, \dots, a_m, e_1, e_2, \dots, e_n$

2. Prove in zero-knowledge that

Verify a 'square and multiply' algorithm in zero-knowledge

$$\prod_{i=1}^{m}(x - a_i) = \prod_{j=1}^{n}\left(x - b_j\right)^{e_j} \text{ for random } x$$

Memory Consistency



One memory location -> Cycle

All locations -> permutation

Memory Consistency

Approach:

$$a_1, a_2, \dots, a_m \text{ is a permutation of } b_1, b_2, \dots, b_m$$

$$\Leftrightarrow$$

$$\prod_{i=1}^{m}(X - a_i) = \prod_{i=1}^{m}(X - b_i)$$

Protocol similar to the look-up argument.

Summary

- Nearly-linear proving time
- Sublinear verification time
- New word decomposition technique for verifying binary operations over non-binary fields
- New look-up argument

Thanks!

Work	Prover Complexity	Verifier Complexity	Communication	Rounds	Assumption
BCTV14	$\Omega(T(\log T)^2)$	$\omega(L + \lvert v \rvert)$	$\omega(1)$	1	KoE
This Work	$O(\alpha T)$	$poly(\lambda)(\sqrt{T} + L + \lvert v \rvert)$	$poly(\lambda)(\sqrt{T} + L)$	$O(\log \log T)$	LT-CRHF

Security $2^{-\omega(\log \lambda)}$

Program Length L

Runtime Bound T

Public Input V