MuxProofs

Succinct Arguments for Machine Computation from Vector Lookups

Zijing Di   Lucas Xia   Wilson Nguyen   Nirvan Tyagi

ASIACRYPT 2024

Goal: Proofs for Machine Computation

Zero-knowledge, succinct proofs of knowledge of correct output of a program executed on specified input

Program Description

```
Prog:
  LOAD r1 0
  MOV r1 r2
  ADD r2 1
  JMPIF r2 rx
  Prog Instruction 5
  Prog Instruction 6
  ...
  Prog Instruction 14
  Prog Instruction 15
  Prog Instruction 16
```

Fixed Instruction Set

```
ADD
SUB
MOV
MUL
LOAD
JMPIF
Instruction 7
Instruction 8
```

Goal: Proofs for Machine Computation

Zero-knowledge, succinct proofs of knowledge of correct output of a program executed on specified input

Program Description

```
Prog:
  LOAD r1 0
  MOV r1 r2
  ADD r2 1
  JMPIF r2 rx
  Prog Instruction 5
  Prog Instruction 6
  ...
  Prog Instruction 14
  Prog Instruction 15
  Prog Instruction 16
```

Fixed Instruction Set

```
  ADD
  SUB
  MOV
  MUL
  LOAD
  JMPIF
  Instruction 7
  Instruction 8
```

Program Execution Trace

```
Prog(input):
  LOAD r1 0 (r1 <- input)
  MOV r1 r2
  ADD r2 1
  JMPIF r2 rx
  ...
  Executed Instruction 30
  Executed Instruction 31
  Executed Instruction 32
Returns output
```

Goal: Proofs for Machine Computation

Zero-knowledge, succinct proofs of knowledge of correct output of a program executed on specified input

Program Description

```
Prog:
  LOAD r1 0
  MOV r1 r2
  ADD r2 1
  JMPIF r2 rx
  Prog Instruction 5
  Prog Instruction 6
  ...
  Prog Instruction 14
  Prog Instruction 15
  Prog Instruction 16
```

Fixed Instruction Set

```
  ADD
  SUB
  MOV
  MUL
  LOAD
  JMPIF
  Instruction 7
  Instruction 8
```

Program Execution Trace

```
Prog(input):
  LOAD r1 0 (r1 <- input)
  MOV r1 r2
  ADD r2 1
  JMPIF r2 rx
  ...
  Executed Instruction 30
  Executed Instruction 31
  Executed Instruction 32
Returns output
```

Statement `x = (Prog, input, output)`

Witness `w = Execution trace`

$(x,w) \in R$ iff

"Prog executed on input results in output"

Goal: Proofs for Machine Computation

Zero-knowledge, succinct proofs of knowledge of correct output of a program executed on specified input

Program Description

```
Prog:
  LOAD r1 0
  MOV r1 r2
  ADD r2 1
  JMPIF r2 rx
  Prog Instruction 5
  Prog Instruction 6
  ...
  Prog Instruction 15
```

Fixed Instruction Set

```
ADD
SUB
MOV
MUL
LOAD
JMPIF
Instruction 7
Instruction 8
```

Program Execution Trace

```
Prog(input):
  LOAD r1 0 (r1 <- input)
  MOV r1 r2
  ADD r2 1
  JMPIF r2 rx
  ...
  Executed Instruction 30
  Executed Instruction 31
  Executed Instruction 32
Returns output
```

Zero-knowledge: Verifier does not learn anything about w

Proof of knowledge (soundness): Prover must know (x,w)∈R to convince verifier

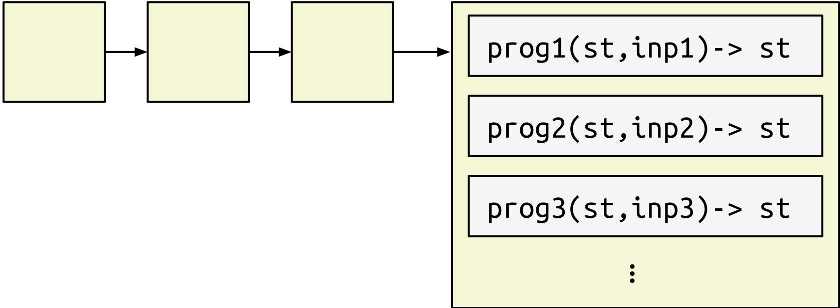
Succinctness: Proof/verification is small w.r.t. program, instruction set, and execution trace

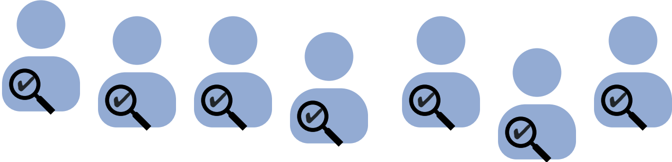
Statement `x = (Prog, input, output)`

Witness `w = Execution trace`

`(x,w)∈R iff`
`"Prog executed on input results in output"`

Application: Scalability for Blockchain Smart Contracts



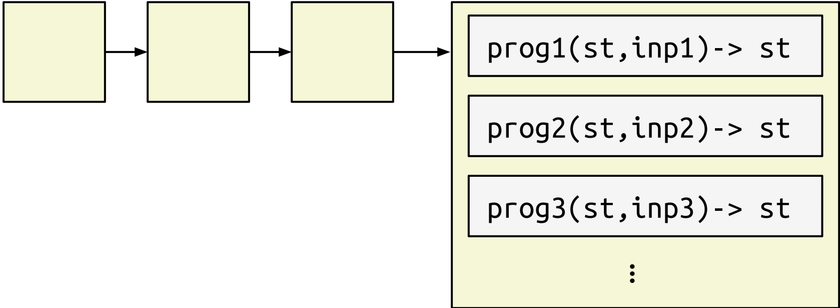


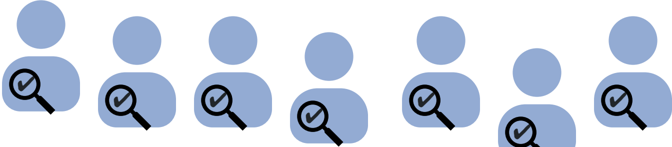
Blockchain is updated by applying smart contracts defined over supported instruction set, e.g., Ethereum Virtual Machine (EVM)

Contract data is dispersed across validators

Validators recompute contracts to ensure state has been updated correctly

Application: Scalability for Blockchain Smart Contracts





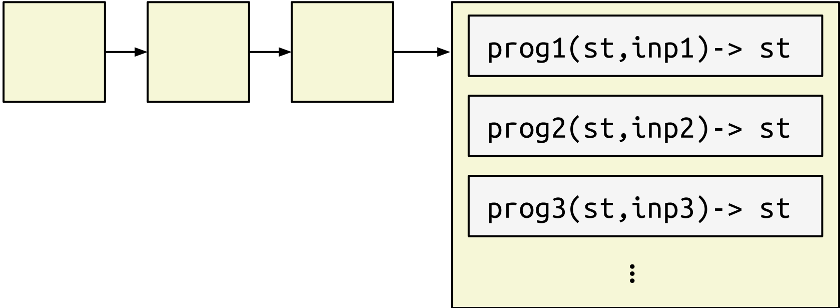
Blockchain is updated by applying smart contracts defined over supported instruction set, e.g., Ethereum Virtual Machine (EVM)

Contract data is dispersed across validators

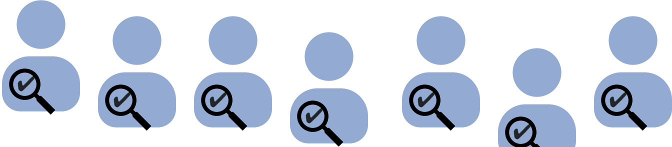
Validators recompute contracts to ensure state has been updated correctly

Succinct proofs of machine computation avoid redundancy in recomputation of contracts!

Application: Privacy for Blockchain Smart Contracts




```
prog1, inp1,
prog2, inp2,
prog3, inp3,
...
```



Validators

Blockchain is updated by applying smart contracts defined over supported instruction set, e.g., Ethereum Virtual Machine (EVM)

Contract data is dispersed across validators

Validators recompute contracts to ensure state has been updated correctly

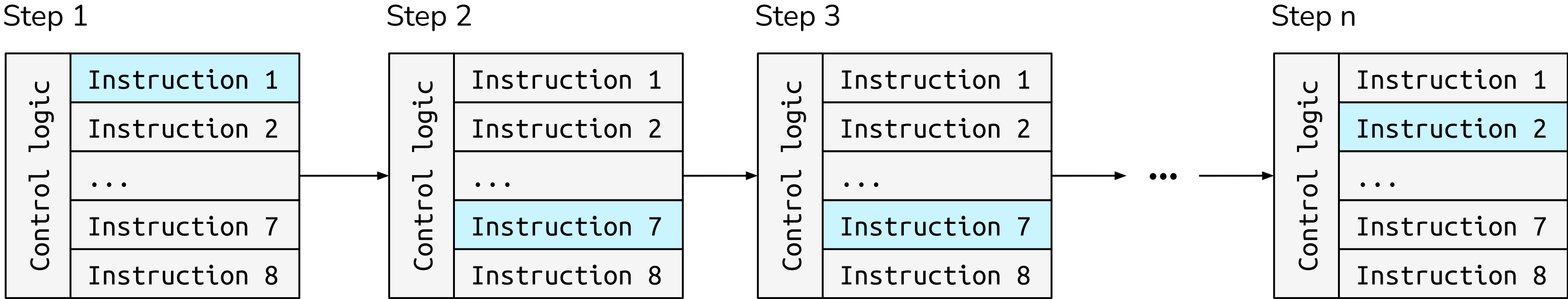
Succinct proofs of machine computation avoid redundancy in recomputation of contracts!

Zero-knowledge proofs of machine computation can hide contract and inputs allowing for privacy-preserving apps!

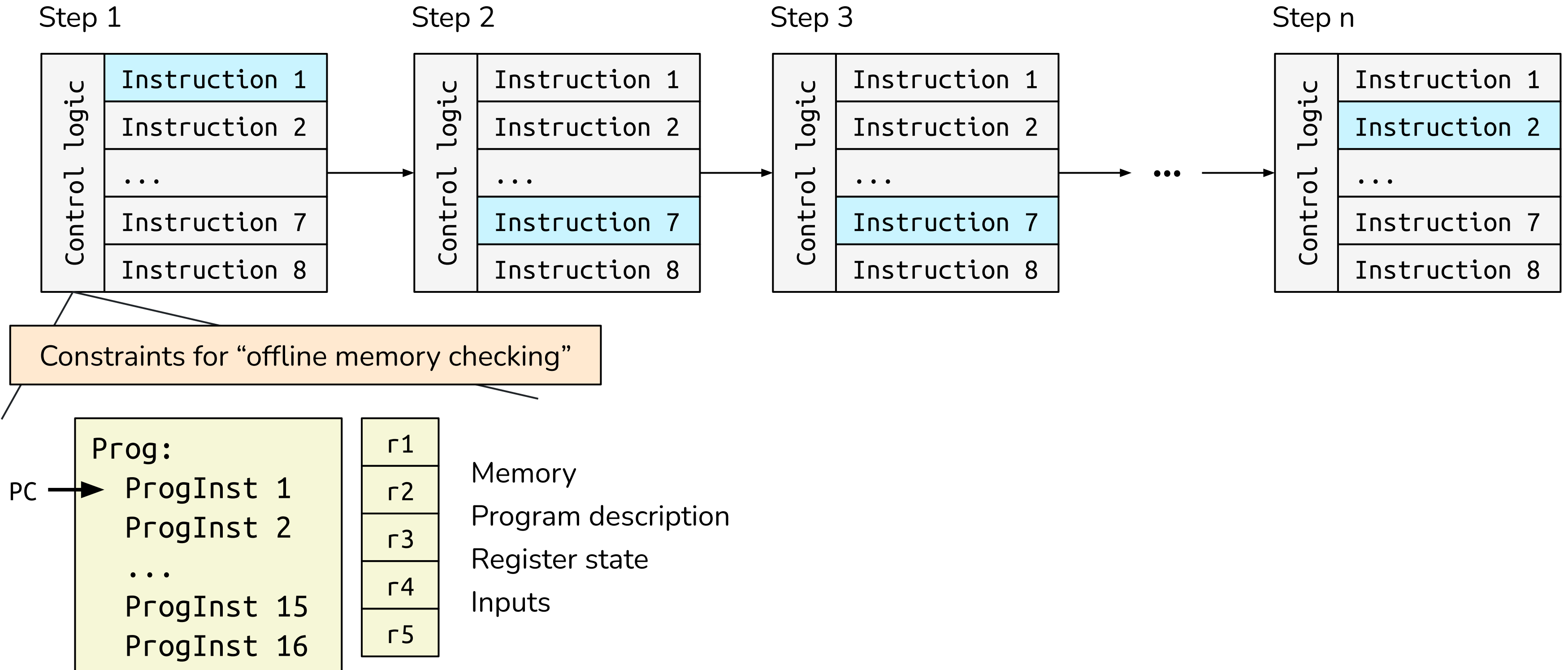
This talk: MuxProofs

- New approach to proofs for machine execution
  - Prover costs scale with the cost of executed instructions
  - Avoids proof recursion
- New core cryptographic primitive: Succinct vector lookups
- Strategy:
  - Machine computation from succinct vector lookups
  - Succinct vector lookups from polynomial encodings on structured evaluation subdomains

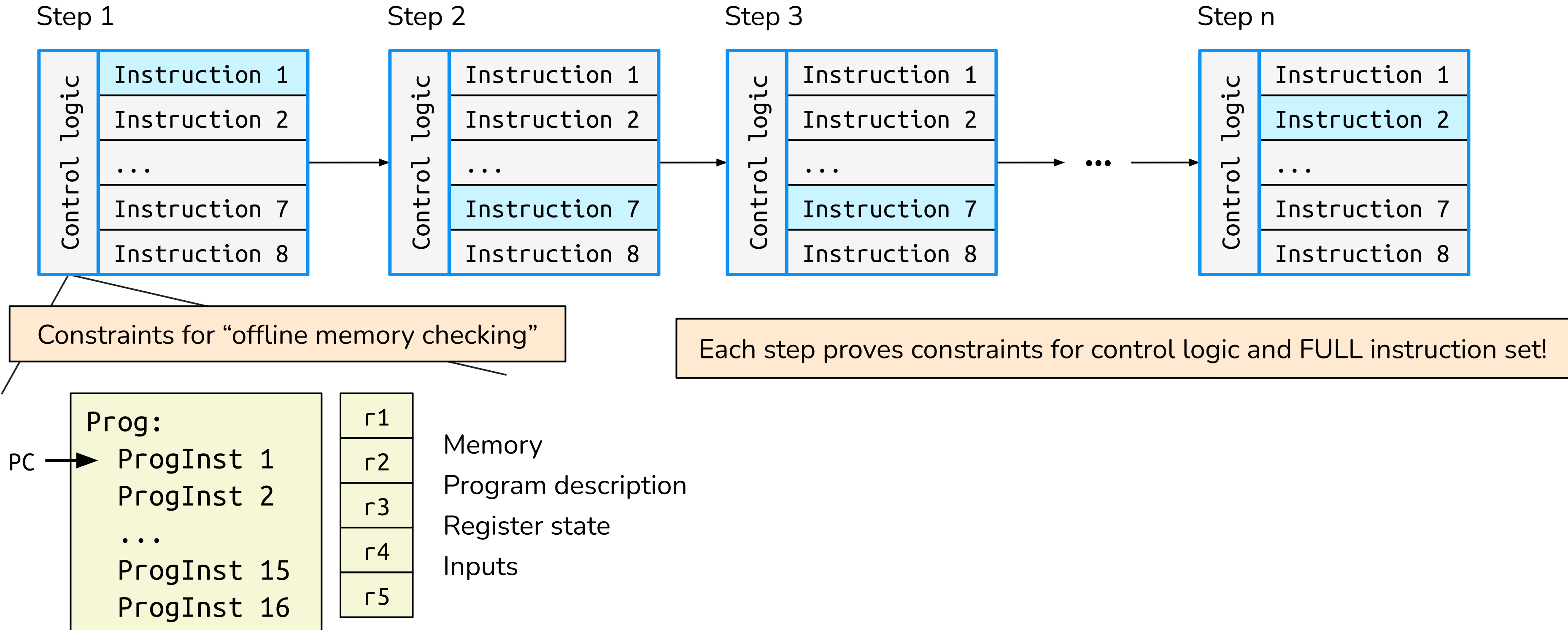
Prior work: Universal instruction set constraints



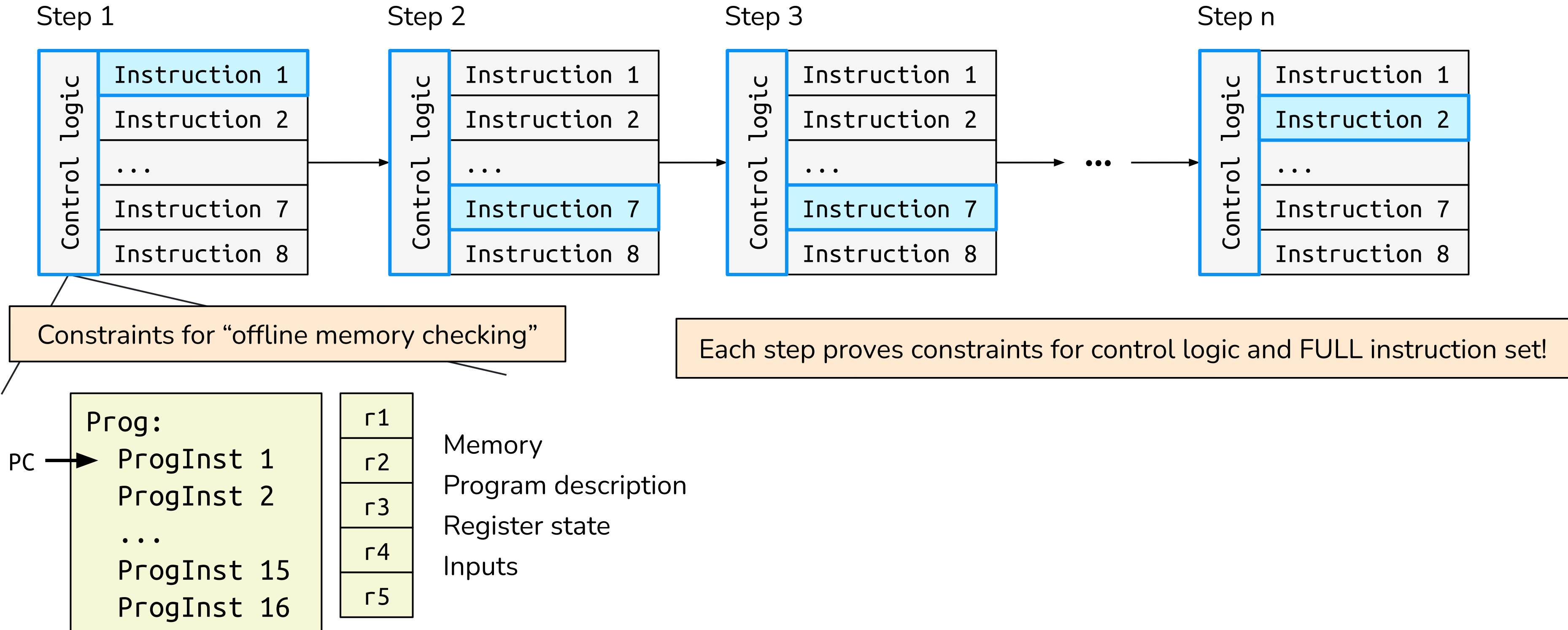
Prior work: Universal instruction set constraints



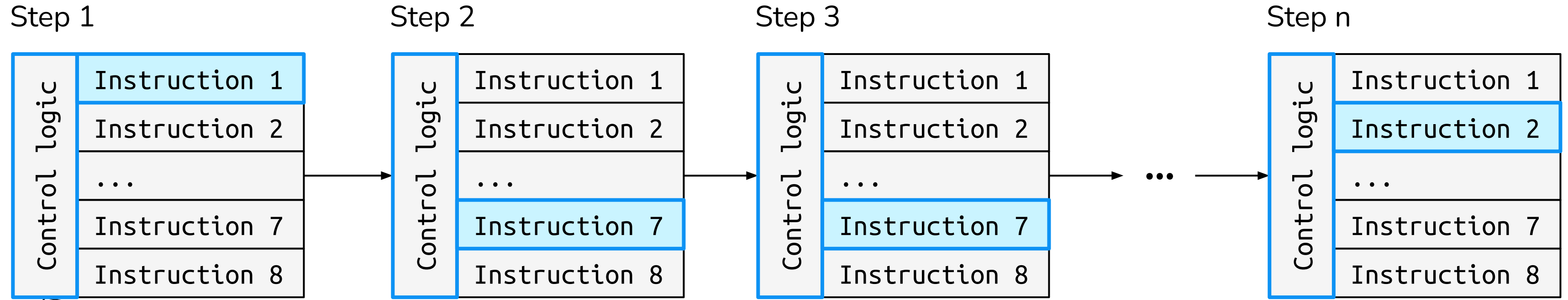
Prior work: Universal instruction set constraints



Goal: Executed instruction constraints

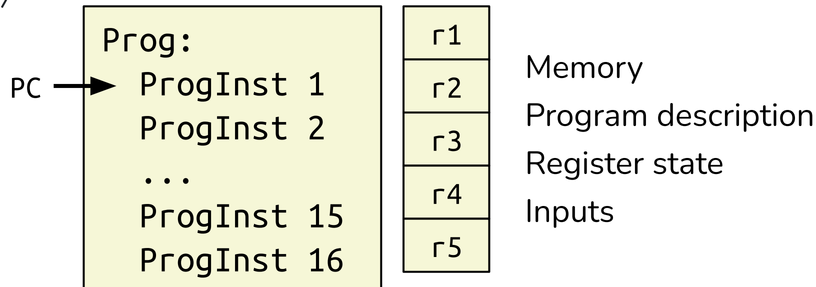


Goal: Executed instruction constraints



Constraints for "offline memory checking"

Each step proves constraints for control logic and FULL instruction set!

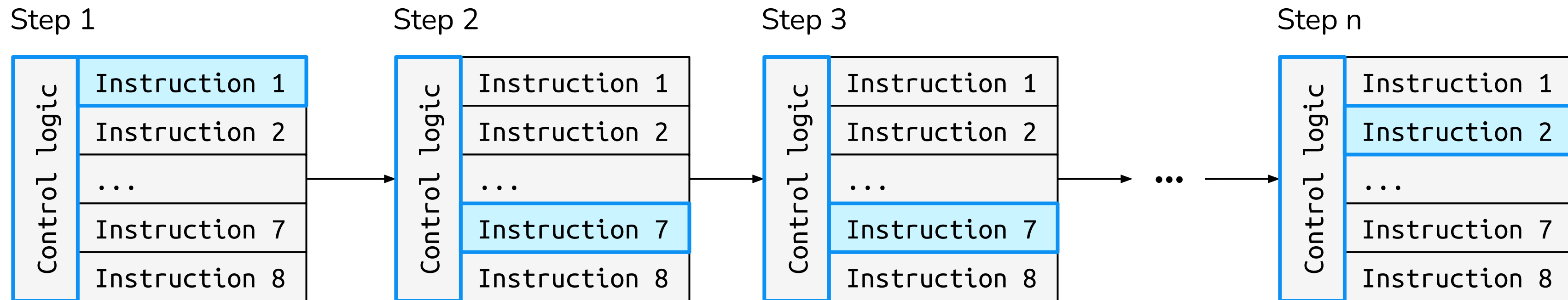


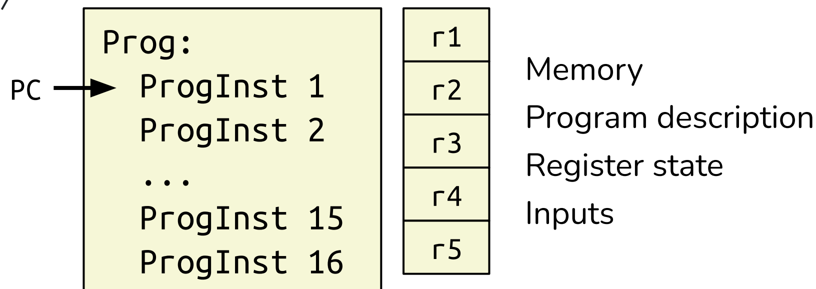
Hard to prove only constraints of executed instructions when:

- executed instructions depend on input
- want to provide zero-knowledge of executed instructions

Other work: Executed instruction constraints **w/ recursion**

SuperNova [Kothapalli & Setty] achieve proving costs on the order of executed constraints using proof recursion (folding)





Memory

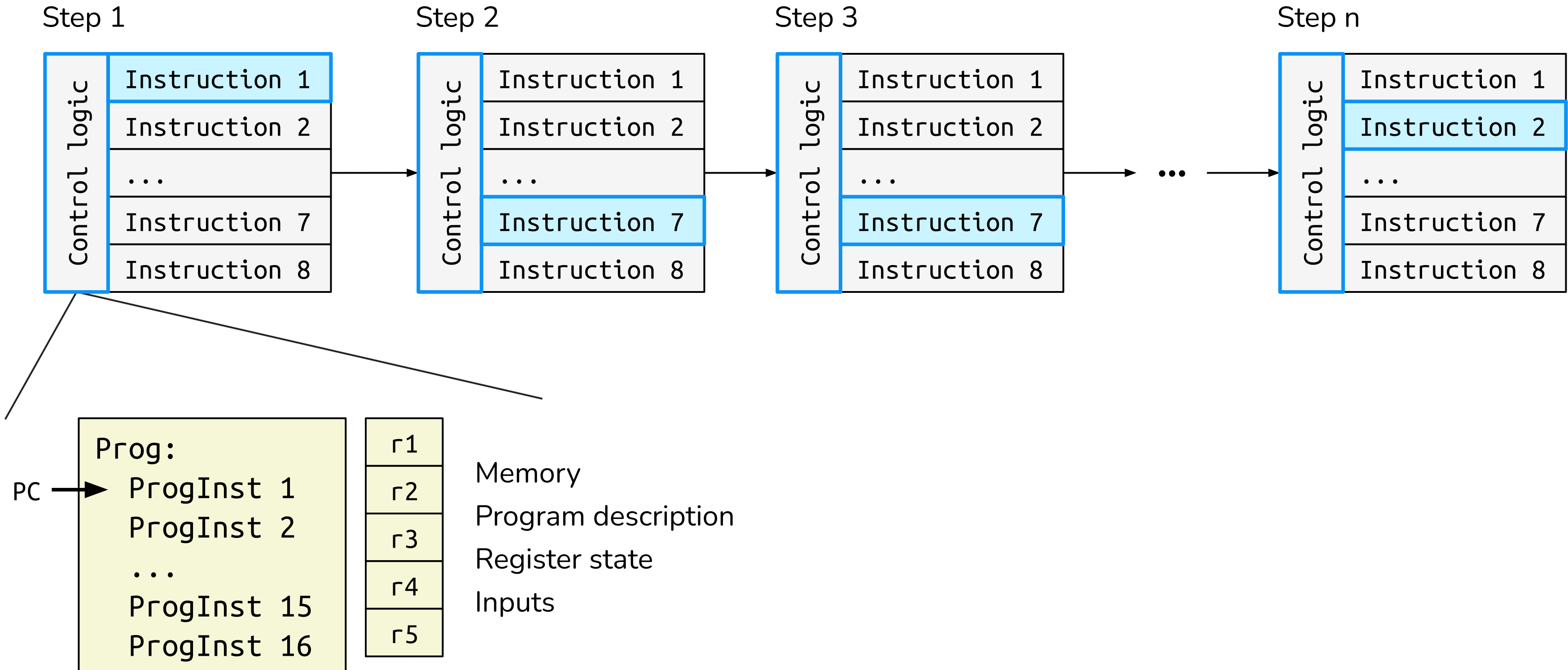
Program description

Register state

Inputs

This work: Executed instruction constraints w/o recursion

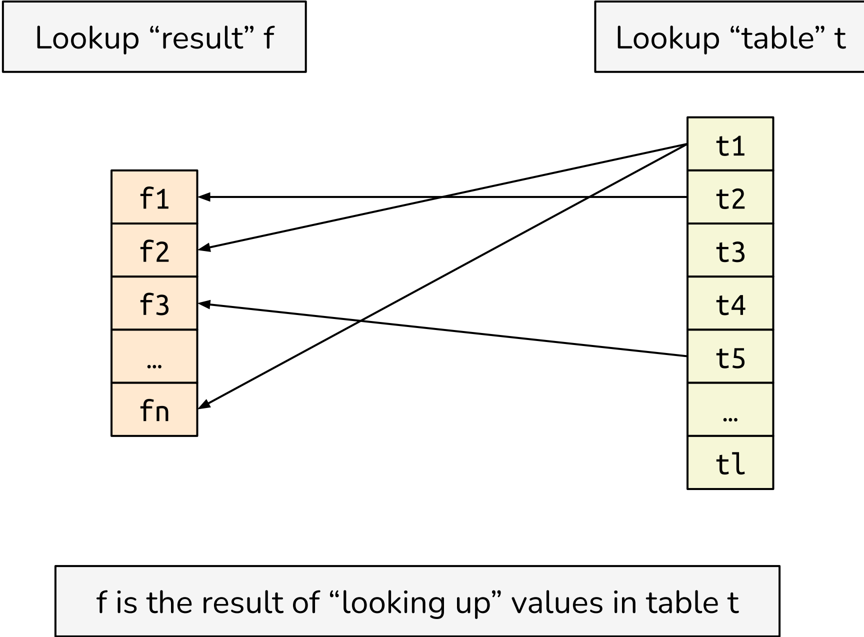
SuperNova [Kothapalli & Setty] achieve proving costs on the order of executed constraints using proof recursion (folding)



This talk: MuxProofs

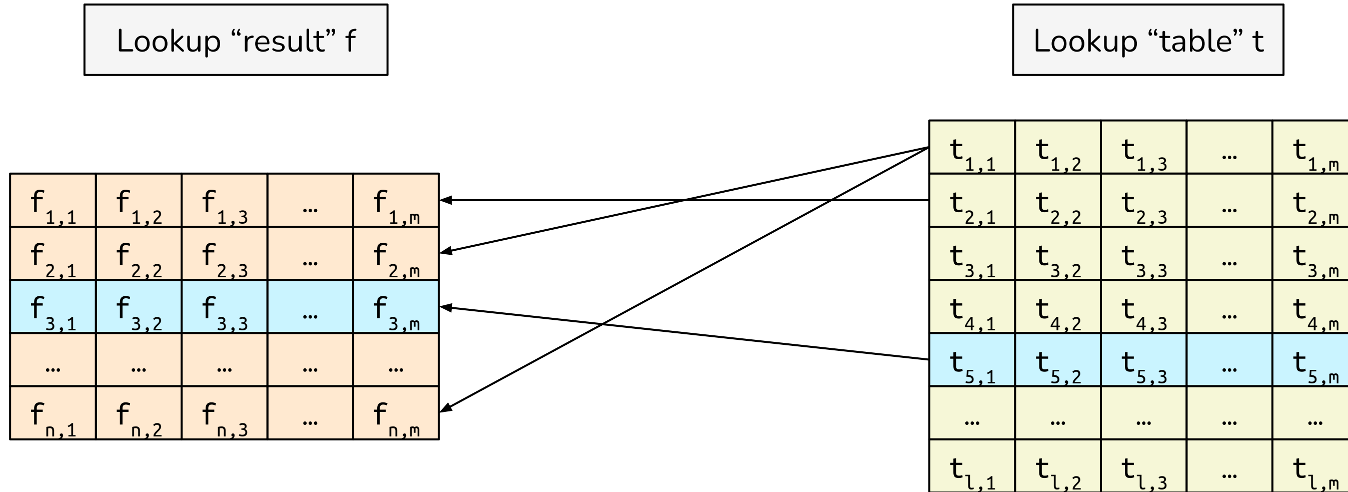
- New approach to proofs for machine execution
  - Prover costs scale with the cost of executed instructions
  - Avoids proof recursion
- New core cryptographic primitive: Succinct vector lookups
- Strategy:
  - Machine computation from succinct vector lookups
  - Succinct vector lookups from polynomial encodings on structured evaluation subdomains

Key cryptographic primitive: Lookup arguments



$$\{fi\}_{i \in [1,n]} \subseteq \{tk\}_{k \in [1,l]}$$

New cryptographic primitive: Succinct vector lookups

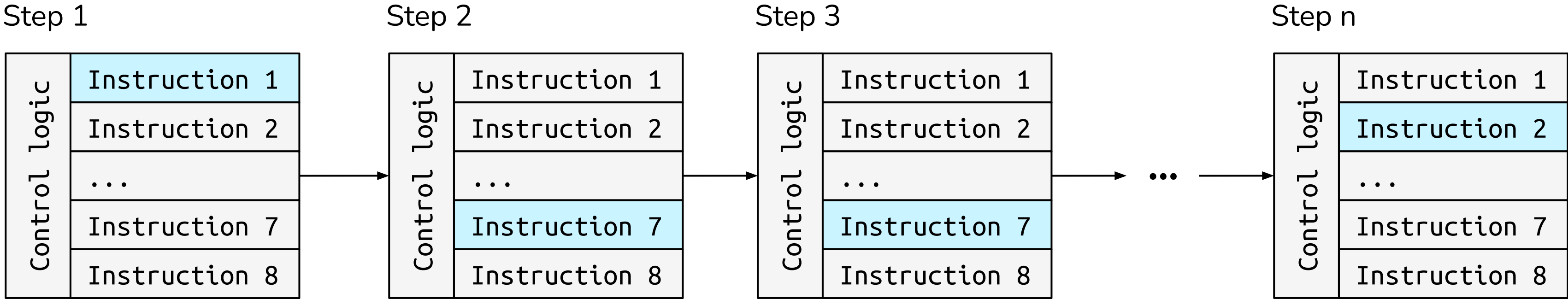


f is the result of "looking up" **vectors** in table t

$$\{[f_{i,1},\dots,f_{i,m}]\}_{i\in[1,n]} \subseteq \{[t_{k,1},\dots,t_{k,m}]\}_{k\in[1,l]}$$

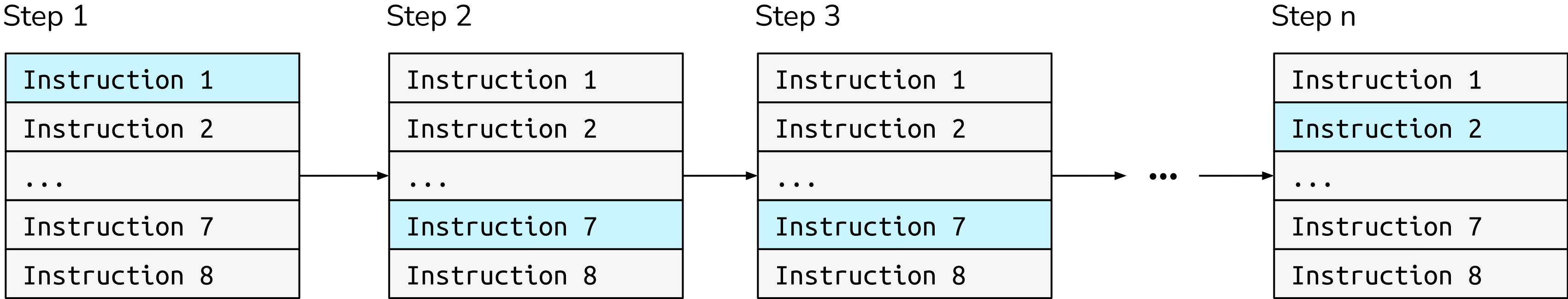
Strategy: Vector lookups for machine computation

Detour: Preprocessing SNARKs based on polynomial encodings of vectors



Strategy: Vector lookups for machine computation

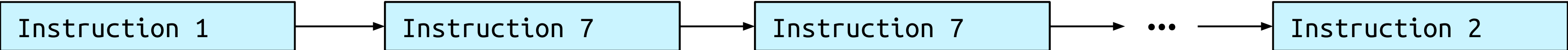
Detour: Preprocessing SNARKs based on polynomial encodings of vectors



Strategy: Vector lookups for machine computation

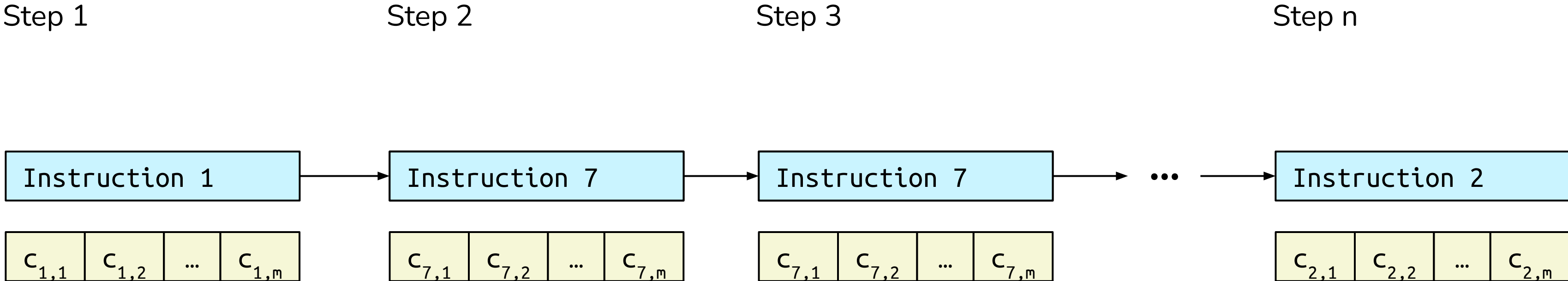
Detour: Preprocessing SNARKs based on polynomial encodings of vectors

Step 1 Step 2 Step 3 Step n



Strategy: Vector lookups for machine computation

Detour: Preprocessing SNARKs based on polynomial encodings of vectors

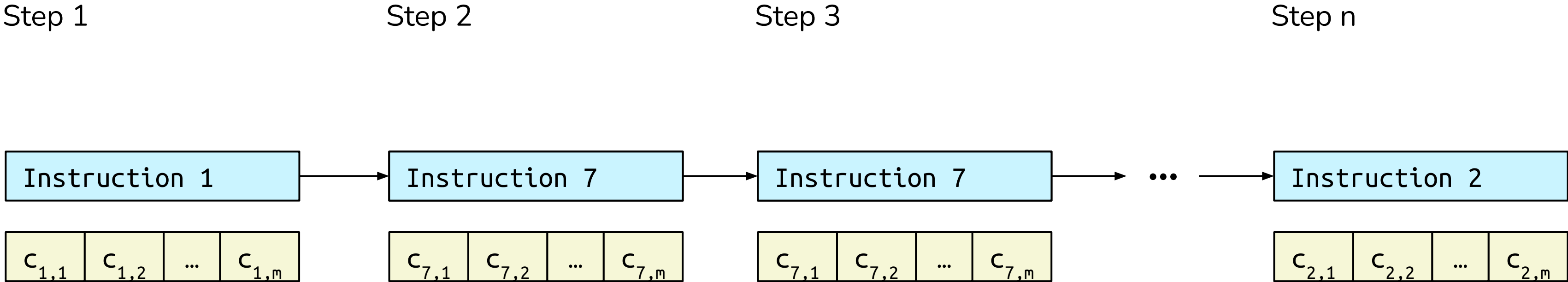


Constraints for an instruction are represented as a vector of field elements

Verifier preprocesses full vector of constraints as polynomial commitment for use in SNARK

Strategy: Vector lookups for machine computation

Detour: Preprocessing SNARKs based on polynomial encodings of vectors

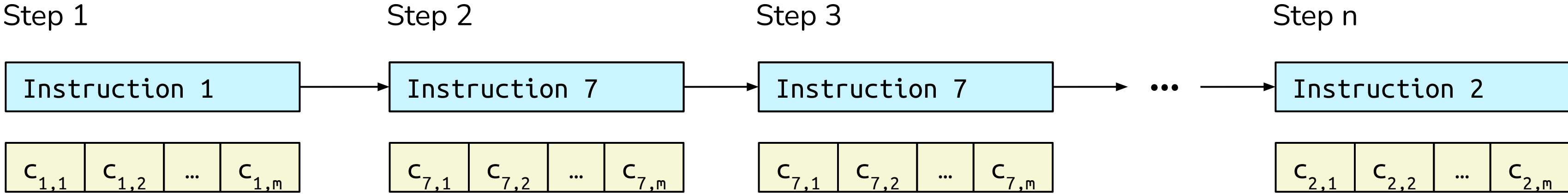


Constraints for an instruction are represented as a vector of field elements

Verifier preprocesses full vector of constraints as polynomial commitment for use in SNARK

Verifier cannot preprocess in machine computation setting since executed instruction order is not known ahead of time!

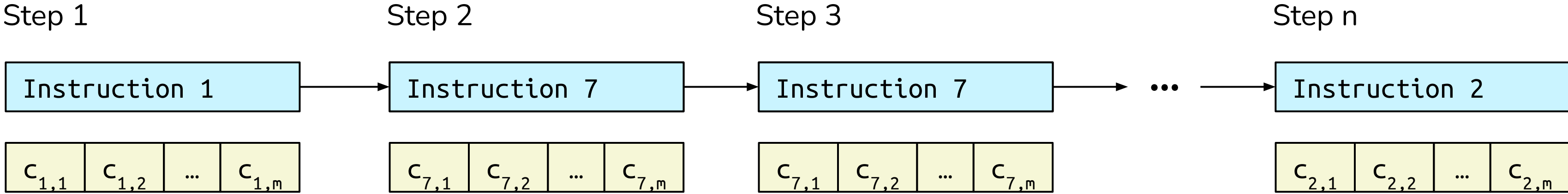
Strategy: Vector lookups for machine computation

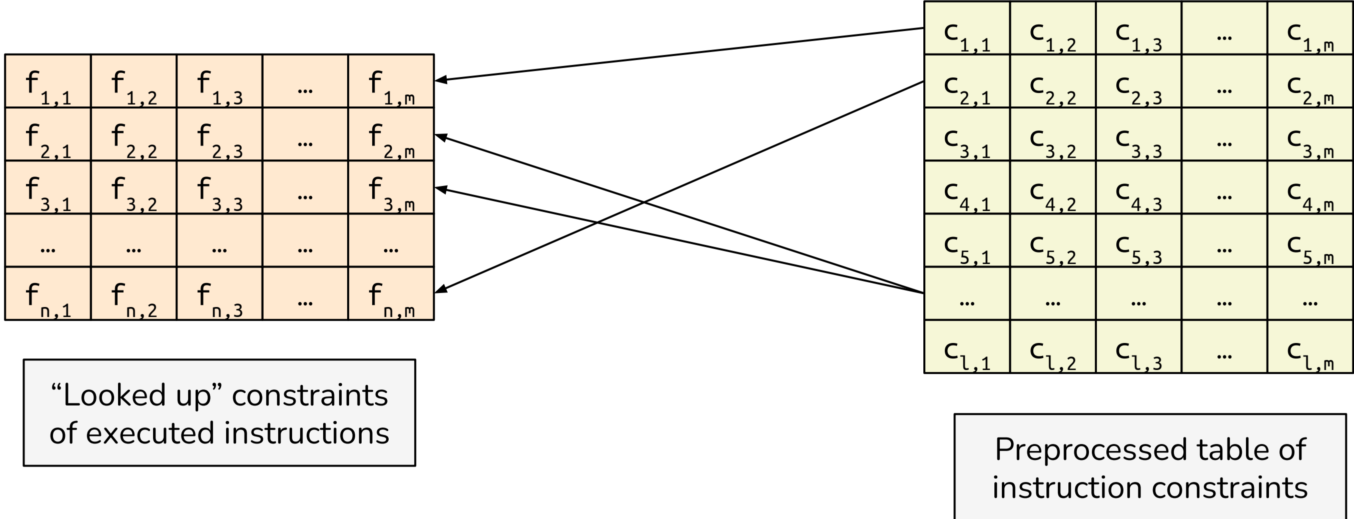


$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	…	$c_{1,m}$
$c_{2,1}$	$c_{2,2}$	$c_{2,3}$	…	$c_{2,m}$
$c_{3,1}$	$c_{3,2}$	$c_{3,3}$	…	$c_{3,m}$
$c_{4,1}$	$c_{4,2}$	$c_{4,3}$	…	$c_{4,m}$
$c_{5,1}$	$c_{5,2}$	$c_{5,3}$	…	$c_{5,m}$
…	…	…	…	…
$c_{l,1}$	$c_{l,2}$	$c_{l,3}$	…	$c_{l,m}$

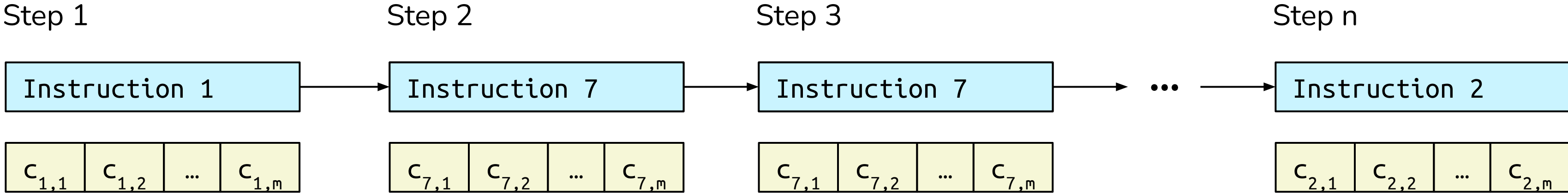
Preprocessed table of instruction constraints

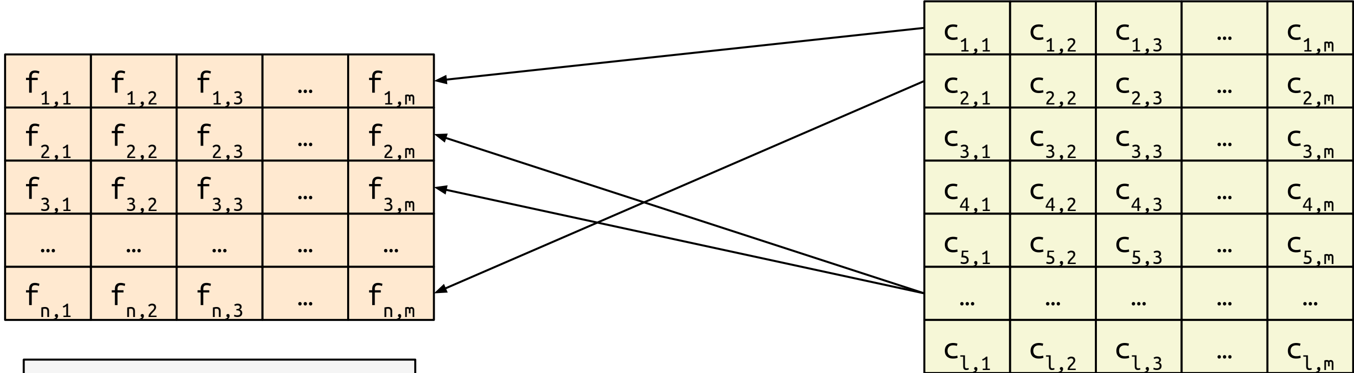
Strategy: Vector lookups for machine computation





Strategy: Vector lookups for machine computation





"Looked up" constraints of executed instructions

Preprocessed table of instruction constraints

Run SNARK on "f" as if it was a preprocessed vector of constraints

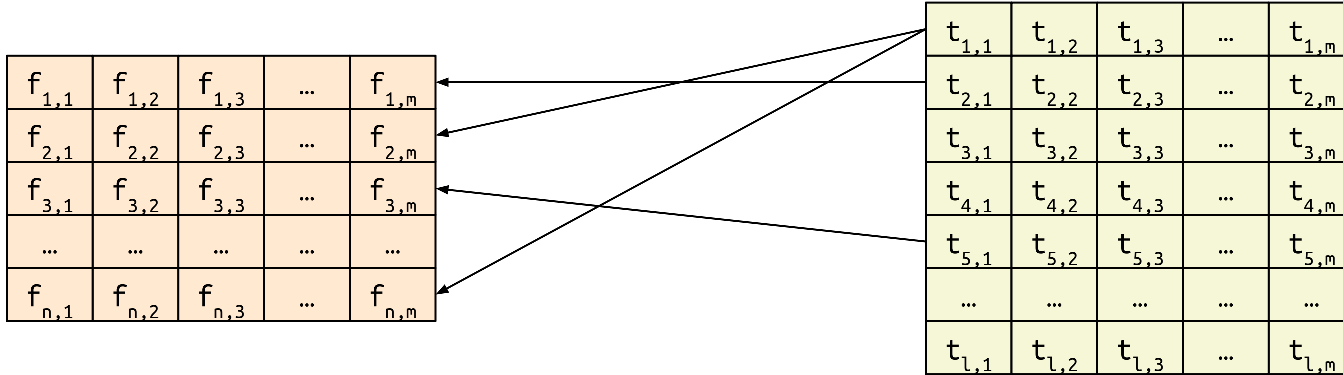
This talk: MuxProofs

- New approach to proofs for machine execution
  - Prover costs scale with the cost of executed instructions
  - Avoids proof recursion
- New core cryptographic primitive: Succinct vector lookups
- Strategy:
  - Machine computation from succinct vector lookups
  - Succinct vector lookups from polynomial encodings on structured evaluation subdomains

Succinct vector lookups via encodings over cosets

Important: Succinctness in “m” where “m” is the constraint length for an instruction.

- RISC-V: 32-bit instructions have m ≈ 128
- EVM: Some expensive instructions like SHA3 hash or ECDSA verification (m ≈ 1.5 mil)



$$\{[f_{i,1},\dots,f_{i,m}]\}_{i\in[1,n]} \subseteq \{[t_{k,1},\dots,t_{k,m}]\}_{k\in[1,l]}$$

Prelims: Polynomial encoding of fields with cosets

$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	…	$f_{1,m}$
$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	…	$f_{2,m}$
$f_{3,1}$	$f_{3,2}$	$f_{3,3}$	…	$f_{3,m}$
…	…	…	…	…
$f_{n,1}$	$f_{n,2}$	$f_{n,3}$	…	$f_{n,m}$

f:

$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	…	$f_{1,m}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	…	$f_{2,m}$	…	$f_{n,1}$	$f_{n,2}$	$f_{n,3}$	…	$f_{n,m}$
1	ω	ω^2	…	ω^{m-1}	ω^m	…								…	ω^{mn-1}

Take subgroup of $\Omega \subseteq F^*$ of size $|\Omega| = mn$ generated by $\Omega = \langle\omega\rangle$

Prelims: Polynomial encoding of fields with cosets

$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	…	$f_{1,m}$
$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	…	$f_{2,m}$
$f_{3,1}$	$f_{3,2}$	$f_{3,3}$	…	$f_{3,m}$
…	…	…	…	…
$f_{n,1}$	$f_{n,2}$	$f_{n,3}$	…	$f_{n,m}$

f:

$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	…	$f_{1,m}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	…	$f_{2,m}$	…	$f_{n,1}$	$f_{n,2}$	$f_{n,3}$	…	$f_{n,m}$
1	ω	ω^2	…	ω^{m-1}	ω^m	…								…	ω^{mn-1}

Take subgroup of $\Omega \subseteq F^*$ of size $|\Omega| = mn$ generated by $\Omega = \langle\omega\rangle$

Does not provide algebraic structure to manipulate vectors

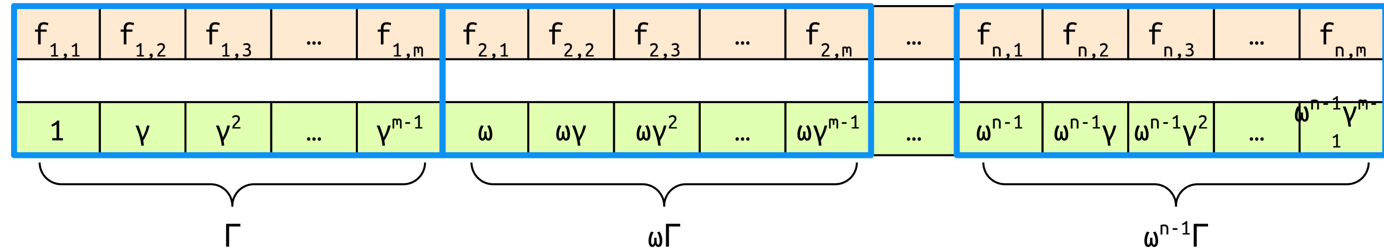
Prelims: Polynomial encoding of fields with cosets

$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	…	$f_{1,m}$
$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	…	$f_{2,m}$
$f_{3,1}$	$f_{3,2}$	$f_{3,3}$	…	$f_{3,m}$
…	…	…	…	…
$f_{n,1}$	$f_{n,2}$	$f_{n,3}$	…	$f_{n,m}$

f:

$\gamma = \omega^n$

n cosets of Γ in Ω:



Take subgroup of $\Omega \subseteq \mathbb{F}^*$ of size $|\Omega| = mn$ generated by $\Omega = \langle\omega\rangle$

Take subgroup of $\Gamma \subseteq \Omega$ of size $|\Gamma| = m$ generated by $\Gamma = \langle\gamma\rangle$

Prelims: Polynomial encoding of fields with cosets

$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	…	$f_{1,m}$
$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	…	$f_{2,m}$
$f_{3,1}$	$f_{3,2}$	$f_{3,3}$	…	$f_{3,m}$
…	…	…	…	…
$f_{n,1}$	$f_{n,2}$	$f_{n,3}$	…	$f_{n,m}$

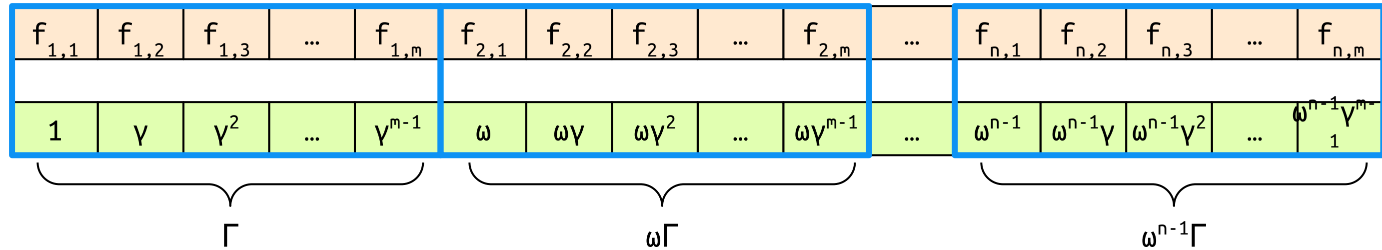
$t_{1,1}$	$t_{1,2}$	$t_{1,3}$	…	$t_{1,m}$
$t_{2,1}$	$t_{2,2}$	$t_{2,3}$	…	$t_{2,m}$
$t_{3,1}$	$t_{3,2}$	$t_{3,3}$	…	$t_{3,m}$
$t_{4,1}$	$t_{4,2}$	$t_{4,3}$	…	$t_{4,m}$
$t_{5,1}$	$t_{5,2}$	$t_{5,3}$	…	$t_{5,m}$
…	…	…	…	…
$t_{l,1}$	$t_{l,2}$	$t_{l,3}$	…	$t_{l,m}$

Need cosets of size `m`, `nm`, and `lm` fit to instruction set size and execution length!

f:

$\gamma=\omega^n$

n cosets of Γ in Ω:



Take subgroup of $\Omega \subseteq F^*$ of size $|\Omega| = mn$ generated by $\Omega = \langle\omega\rangle$

Take subgroup of $\Gamma \subseteq \Omega$ of size $|\Gamma| = m$ generated by $\Gamma = \langle\gamma\rangle$

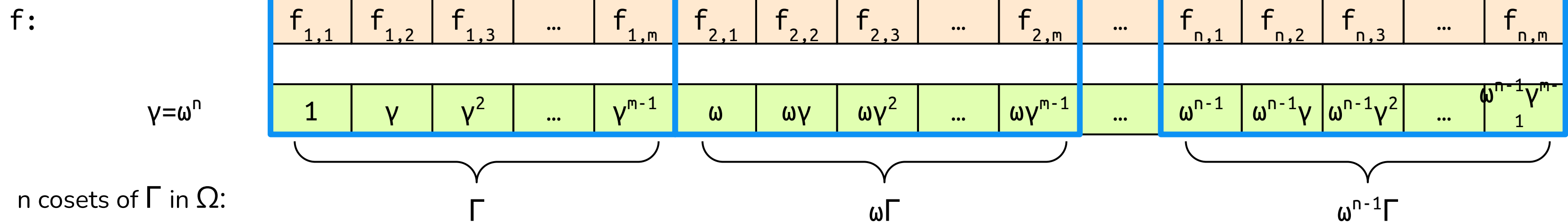
Prelims: Polynomial encoding of fields with cosets

$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	…	$f_{1,m}$
$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	…	$f_{2,m}$
$f_{3,1}$	$f_{3,2}$	$f_{3,3}$	…	$f_{3,m}$
…	…	…	…	…
$f_{n,1}$	$f_{n,2}$	$f_{n,3}$	…	$f_{n,m}$

$t_{1,1}$	$t_{1,2}$	$t_{1,3}$	…	$t_{1,m}$
$t_{2,1}$	$t_{2,2}$	$t_{2,3}$	…	$t_{2,m}$
$t_{3,1}$	$t_{3,2}$	$t_{3,3}$	…	$t_{3,m}$
$t_{4,1}$	$t_{4,2}$	$t_{4,3}$	…	$t_{4,m}$
$t_{5,1}$	$t_{5,2}$	$t_{5,3}$	…	$t_{5,m}$
…	…	…	…	…
$t_{l,1}$	$t_{l,2}$	$t_{l,3}$	…	$t_{l,m}$

Need cosets of size `m`, `nm`, and `lm` fit to instruction set size and execution length!

Not a problem! Popular “FFT-friendly” pairing curves like BLS12-381 have `F*` with subgroups of all powers-of-2 up to 2^{32}



Take subgroup of $\Omega \subseteq$ `F*` of size $|\Omega|$ = mn generated by $\Omega = \langle\omega\rangle$

Take subgroup of $\Gamma \subseteq \Omega$ of size $|\Gamma|$ = m generated by $\Gamma = \langle\gamma\rangle$

Construction preview: CosetLookup

Thm [Haböck '22]: If "f" and "t" are sequences of vectors the lookup relation holds

$$\{[f_{i,1},\dots,f_{i,m}]\}_{i\in[1,n]} \subseteq \{[t_{k,1},\dots,t_{k,m}]\}_{k\in[1,l]}$$

w.h.p. if there exists vector "c" that satisfies the following for random α, β (selected after c)

$$\sum_{i\in[1,n]} \frac{1}{(\alpha + \sum_{j\in[1,m]}\beta^{j-1} \cdot f_{i,j})} = \sum_{k\in[1,l]} \frac{c_k}{(\alpha + \sum_{j\in[1,m]}\beta^{j-1} \cdot t_{k,j})}$$

Construction preview: CosetLookup

Thm [Haböck '22]: If "f" and "t" are sequences of vectors the lookup relation holds

$$\{[f_{i,1},\dots,f_{i,m}]\}_{i\in[1,n]} \subseteq \{[t_{k,1},\dots,t_{k,m}]\}_{k\in[1,l]}$$

w.h.p. if there exists vector "c" that satisfies the following for random α, β (selected after c)

$$\sum_{i\in[1,n]} \frac{1}{(\alpha + \sum_{j\in[1,m]}\beta^{j-1}\cdot f_{i,j})} = \sum_{k\in[1,l]} \frac{c_k}{(\alpha + \sum_{j\in[1,m]}\beta^{j-1}\cdot t_{k,j})}$$

Construction preview: CosetLookup

$$\sum_{i\in[1,n]} \frac{1}{(\alpha + \sum_{j\in[1,m]} \beta^{j-1} \cdot f_{i,j})}$$

f:	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	…	$f_{1,m}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	…	$f_{2,m}$	…	$f_{n,1}$	$f_{n,2}$	$f_{n,3}$	…	$f_{n,m}$
$\gamma=\omega^n$	1	γ	γ^2	…	γ^{m-1}	ω	$\omega\gamma$	$\omega\gamma^2$	…	$\omega\gamma^{m-1}$	…	ω^{n-1}	$\omega^{n-1}\gamma$	$\omega^{n-1}\gamma^2$	…	$\omega^{n-1}\gamma^{m-1}$

n cosets of Γ in Ω: Γ, $\omega\Gamma$, …, $\omega^{n-1}\Gamma$

Construction preview: CosetLookup

$$\sum_{i\in[1,n]} \frac{1}{(\alpha + \sum_{j\in[1,m]}\beta^{j-1} \cdot f_{i,j})}$$

f:	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	…	$f_{1,m}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	…	$f_{2,m}$	…	$f_{n,1}$	$f_{n,2}$	$f_{n,3}$	…	$f_{n,m}$
$\gamma=\omega^n$	1	γ	γ^2	…	γ^{m-1}	ω	$\omega\gamma$	$\omega\gamma^2$	…	$\omega\gamma^{m-1}$	…	ω^{n-1}	$\omega^{n-1}\gamma$	$\omega^{n-1}\gamma^2$	…	$\omega^{n-1}\gamma^{m-1}$
n cosets of Γ in Ω:	Γ					$\omega\Gamma$						$\omega^{n-1}\Gamma$				
I:				…					…		…				…	

Construction preview: CosetLookup

$$\sum_{i\in[1,n]} \frac{1}{(\alpha + \sum_{j\in[1,m]} \beta^{j-1} \cdot f_{i,j})}$$

f:	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	…	$f_{1,m}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	…	$f_{2,m}$	…	$f_{n,1}$	$f_{n,2}$	$f_{n,3}$	…	$f_{n,m}$
$\gamma=\omega^n$	1	γ	γ^2	…	γ^{m-1}	ω	$\omega\gamma$	$\omega\gamma^2$	…	$\omega\gamma^{m-1}$	…	ω^{n-1}	$\omega^{n-1}\gamma$	$\omega^{n-1}\gamma^2$	…	$\omega^{n-1}\gamma^{m-1}$
n cosets of Γ in Ω:	Γ					$\omega\Gamma$						$\omega^{n-1}\Gamma$				
I:	1			…					…		…				…	

I(1) = 1

Construction preview: CosetLookup

$$\sum_{i\in[1,n]} \frac{1}{(\alpha + \sum_{j\in[1,m]}\beta^{j-1} \cdot f_{i,j})}$$

f:	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	…	$f_{1,m}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	…	$f_{2,m}$	…	$f_{n,1}$	$f_{n,2}$	$f_{n,3}$	…	$f_{n,m}$
$\gamma=\omega^n$	1	γ	γ^2	…	γ^{m-1}	ω	$\omega\gamma$	$\omega\gamma^2$	…	$\omega\gamma^{m-1}$	…	ω^{n-1}	$\omega^{n-1}\gamma$	$\omega^{n-1}\gamma^2$	…	$\omega^{n-1}\gamma^{m-1}$
n cosets of Γ in Ω:	Γ					$\omega\Gamma$						$\omega^{n-1}\Gamma$				
I:	1	β	β^2	…	β^{m-1}				…		…				…	

$I(1) = 1$

$(I(\gamma X) - \beta \cdot I(X))(X - \gamma^{m-1}) = 0$ over Γ

Construction preview: CosetLookup

$$\sum_{i\in[1,n]} \frac{1}{(\alpha + \sum_{j\in[1,m]} \beta^{j-1} \cdot f_{i,j})}$$

f:	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	…	$f_{1,m}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	…	$f_{2,m}$	…	$f_{n,1}$	$f_{n,2}$	$f_{n,3}$	…	$f_{n,m}$
$\gamma=\omega^n$	1	γ	γ^2	…	γ^{m-1}	ω	$\omega\gamma$	$\omega\gamma^2$	…	$\omega\gamma^{m-1}$	…	ω^{n-1}	$\omega^{n-1}\gamma$	$\omega^{n-1}\gamma^2$	…	$\omega^{n-1}\gamma^{m-1}$
n cosets of Γ in Ω:	Γ					$\omega\Gamma$						$\omega^{n-1}\Gamma$				
I:	1	β	β^2	…	β^{m-1}	1	β	β^2	…	β^{m-1}	…	1	β	β^2	…	β^{m-1}

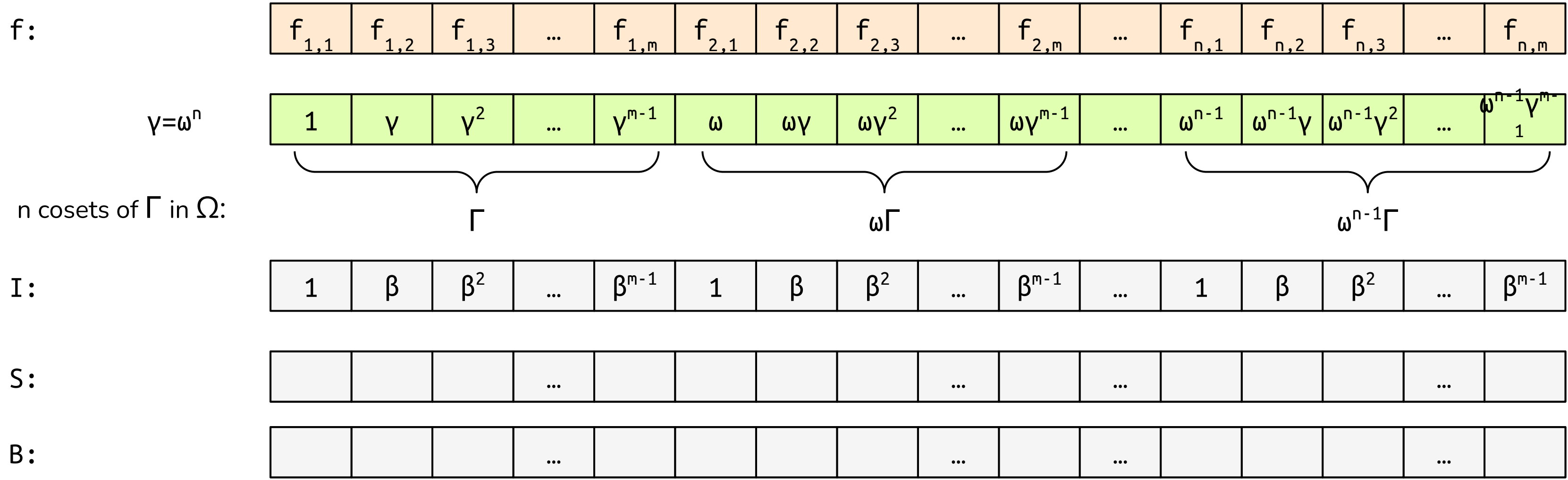
$I(1) = 1$

$(I(\gamma X) - \beta \cdot I(X))(X - \gamma^{m-1}) = 0$ over Γ

$(I(\omega X) - I(X))(Z[\omega^{n-1}\Gamma](X) - \gamma^{m-1}) = 0$ over Ω

Construction preview: CosetLookup

$$\sum_{i\in[1,n]} \frac{1}{(\alpha + \sum_{j\in[1,m]} \beta^{j-1} \cdot f_{i,j})}$$



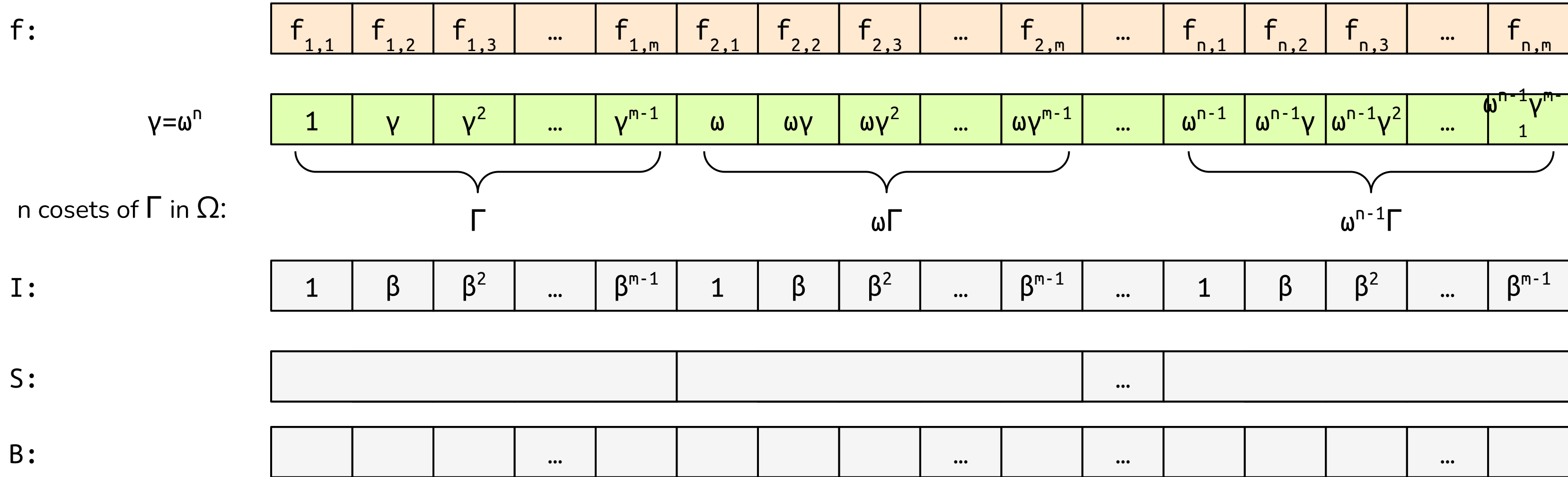
$I(1) = 1$

$(I(\gamma X) - \beta \cdot I(X))(X - \gamma^{m-1}) = 0$ over Γ

$(I(\omega X) - I(X))(Z[\omega^{n-1}\Gamma](X) - \gamma^{m-1}) = 0$ over Ω

Construction preview: CosetLookup

$$\sum_{i\in[1,n]} \frac{1}{(\alpha + \sum_{j\in[1,m]}\beta^{j-1} \cdot f_{i,j})}$$



$I(1) = 1$

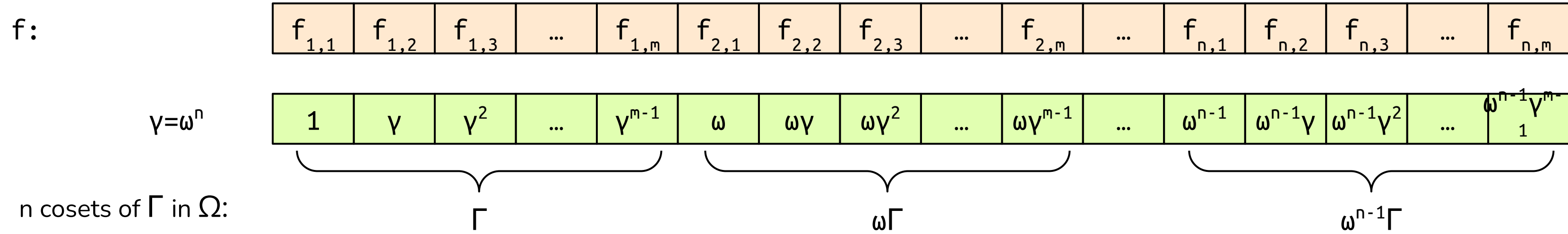
$(I(\gamma X) - \beta \cdot I(X))(X - \gamma^{m-1}) = 0$ over Γ

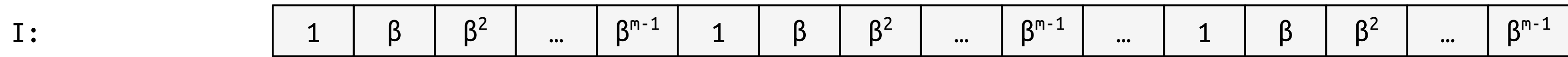
$(I(\omega X) - I(X))(Z[\omega^{n-1}\Gamma](X) - \gamma^{m-1}) = 0$ over Ω

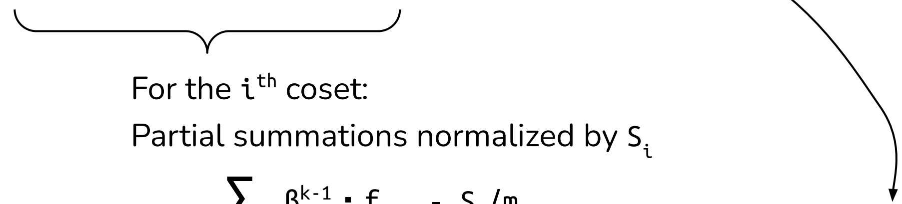
$S(\gamma X) = S(X)$ over Ω

Construction preview: CosetLookup

$$\sum_{i\in[1,n]} \frac{1}{(\alpha + \sum_{j\in[1,m]} \beta^{j-1} \cdot f_{i,j})}$$







$$\sum_{k\in[1,j]} \beta^{k-1} \cdot f_{i,k} - S_i/m$$

$$S_i = \sum_{j\in[1,m]} \beta^{j-1} \cdot f_{i,j}$$

$I(1) = 1$

$(I(\gamma X) - \beta \cdot I(X))(X - \gamma^{m-1}) = 0$ over Γ

$(I(\omega X) - I(X))(Z[\omega^{n-1}\Gamma](X) - \gamma^{m-1}) = 0$ over Ω

$S(\gamma X) = S(X)$ over Ω

$B(\gamma X) = B(X) + I(\gamma X) \cdot B(\gamma X) - S(X)/m$ over Ω

This talk: MuxProofs

- New approach to proofs for machine execution
  - Prover costs scale with the cost of executed instructions
  - Avoids proof recursion
- New core cryptographic primitive: Succinct vector lookups
  - Univariate setting: Coset encoding
  - Multivariate setting: Boolean subcube encoding
- Strategy:
  - Machine computation from succinct vector lookups
  - Succinct vector lookups from polynomial encodings on structured evaluation subdomains

`eprint.iacr.org/2023/974`

`github.com/lucasxia01/mux-proofs-impl`

Reserve Slides

Disclaimer: Why avoid recursion?

Cons of recursion	Pros of recursion
No security proof in random oracle model	Small memory usage

Disclaimer: Why avoid recursion?

Cons of recursion	Pros of recursion
No security proof in random oracle model	Small memory usage
Best encodings use cycles of elliptic curves	
Recursive overhead at each step	

Disclaimer: Why avoid recursion?

Cons of recursion		Pros of recursion
No security proof in random oracle model		Small memory usage
Best encodings use cycles of elliptic curves	$<$	Modern recursion techniques (folding) don't "prove" all constraints, instead "folds" them together into something of smaller size
Recursive overhead at each step		

For proving speed, *probably* state-of-the-art recursive approaches [SuperNova] would outperform MuxProofs

Disclaimer: Why avoid recursion?

Cons of recursion

- No security proof in random oracle model
- Best encodings use cycles of elliptic curves
- Recursive overhead at each step

<

Pros of recursion

- Small memory usage
- Modern recursion techniques (folding) don't "prove" all constraints, instead "folds" them together into something of smaller size

For proving speed, *probably* state-of-the-art recursive approaches [SuperNova] would outperform MuxProofs

We use this observation in Mangrove [CRYPTO '24] where we apply recursion to general computation

Example: Lookups used in proofs for machine computation

Decrease number of constraints in complex instructions with contained "lookups" like SHA2

SHA2 Arithmetic Circuit Constraints > SHA2 Arithmetic Circuit Constraints without lookups + SHA2 Lookup Constraints

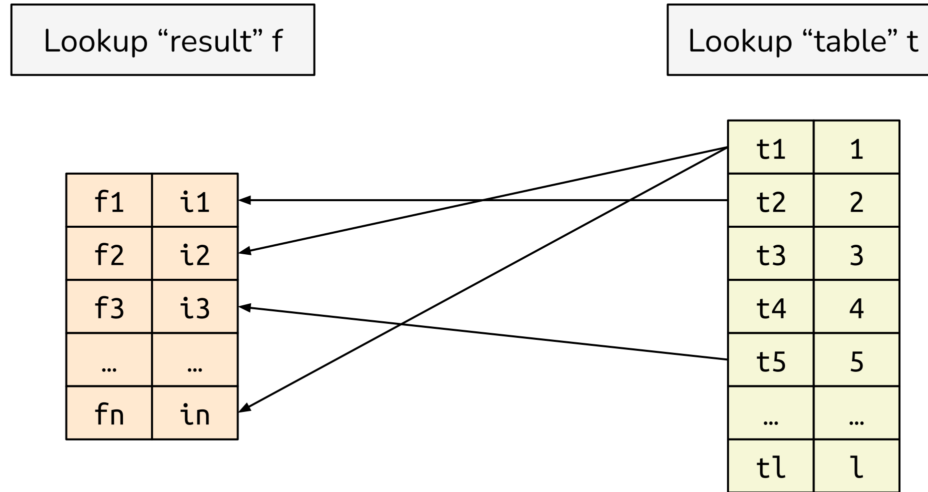
Directly "lookup" instruction result if instruction has certain algebraic structure, e.g., ADD in RISC-V

[Lasso, Jolt EUROCRYPT '24]

2^{32}-bit ADD Arithmetic Circuit Constraints > 2^{32}-bit ADD Lookup Constraints (2^{64}-size table)

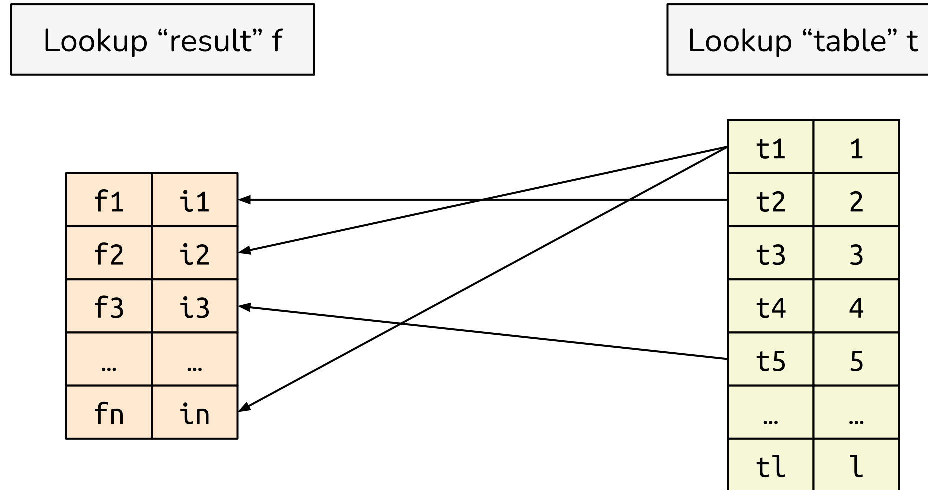
We use lookups differently! Our ideas are orthogonal and can likely be composed with other strategies

Example: Vector lookup arguments specifying index



Vectors of length 2 can be used to enforce lookups for a specified index!

Example: Vector lookup arguments specifying index



Vectors of length 2 can be used to enforce lookups for a specified index!

Can build vector lookups from element lookups [Plookup ‘20], but not succinct in vector length