Polymath: Groth16 Is Not The Limit

Helger Lipmaa, University of Tartu, Estonia

ZKProof 7, Sofia, 24.03.25 (paper appeared at Crypto 2024)

ZK-SNARKs

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

ZK-SNARKs

Computation: f
Public input (statement) x
Private input (witness) w

srs

srs

Computation: f
Public input (statement) x

ZK-SNARKs

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

srs

srs

Proof π that $f(\mathrm{x}, \mathrm{w}) = 1$

ZK-SNARKs

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

srs

srs

Proof π that $f(\mathrm{x}, \mathrm{w}) = 1$

- Completeness

ZK-SNARKs

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

srs

srs

Proof π that $f(\mathrm{x}, \mathrm{w}) = 1$

- Completeness
- Knowledge-soundness

ZK-SNARKs

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

srs

srs

Proof π that $f(\mathrm{x}, \mathrm{w}) = 1$

- Completeness
- Knowledge-soundness
- Zero-knowledge

ZK-SNARKs

Computation: f
Public input (statement) x
Private input (witness) w

srs

srs

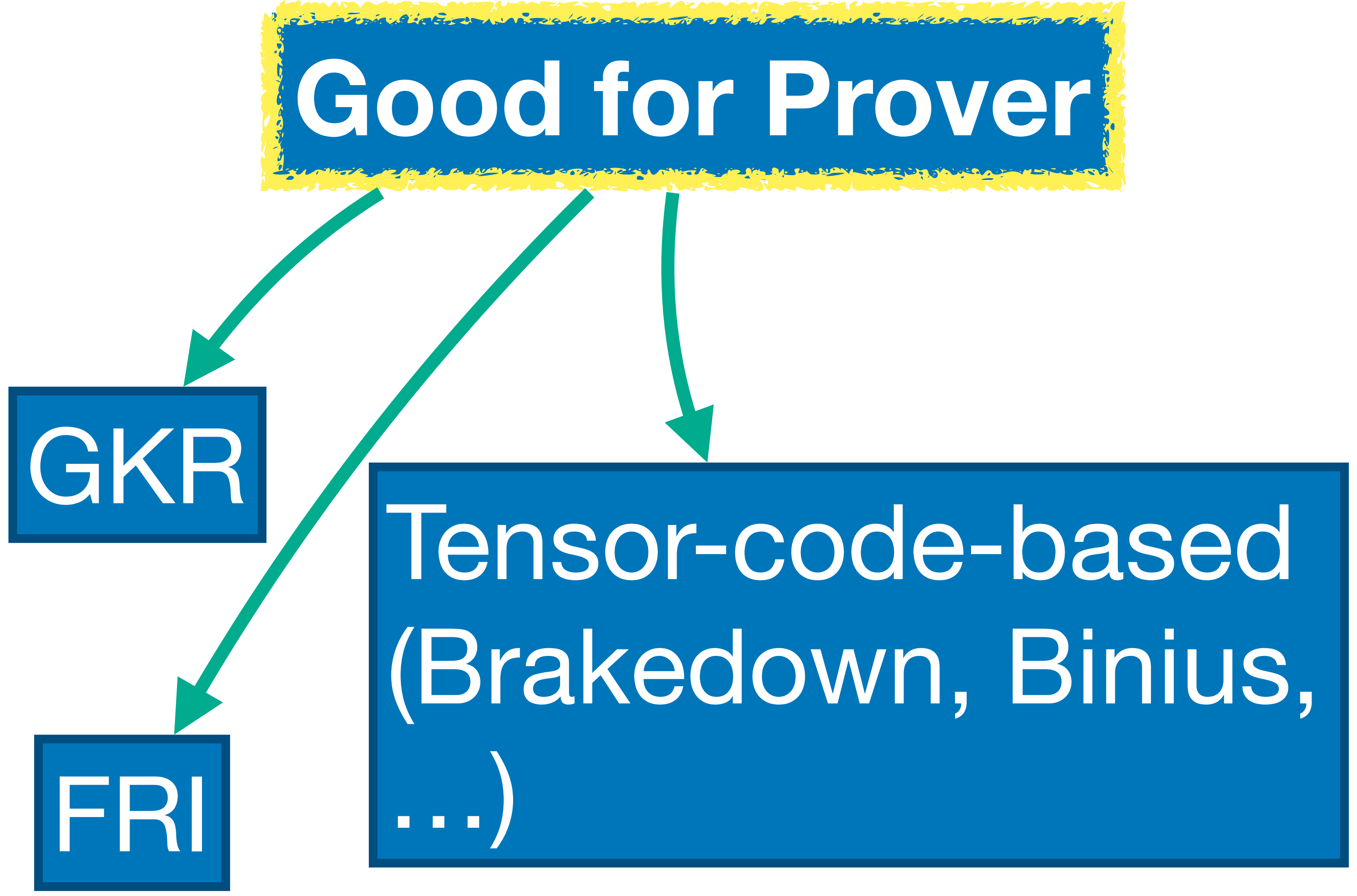
Computation: f
Public input (statement) x

Proof π that $f(\mathrm{x}, \mathrm{w}) = 1$

- Completeness
- Knowledge-soundness
- Zero-knowledge
- Succinct arguments

Landscape

Landscape



Landscape


Good for Prover
GKR
FRI
Tensor-code-based (Brakedown, Binius, …)
Good for Verifier
…
Plonk
Groth16

Landscape

- **Good for Prover**
  - GKR
  - FRI
  - Tensor-code-based (Brakedown, Binius, …)
- **Good for Both**
  - Composition:
    - GKR + **Groth16**
    - Brakedown + **Groth16**
    - FRI + **Groth16**
    - …
- **Good for Verifier**
  - …
  - Plonk
  - Groth16

Landscape

Huge progress in zk-SNARK land in last 5 years
In 2024,Groth16 still landed supreme after 8 years
- Shortest argument
- Fastest verifier

Good for Prover

GKR

FRI

Tensor-code-based (Brakedown, Binius, …)

Good for Both

Composition:
- GKR + **Groth16**
- Brakedown + **Groth16**
- FRI + **Groth16**
- …

Good for Verifier

…

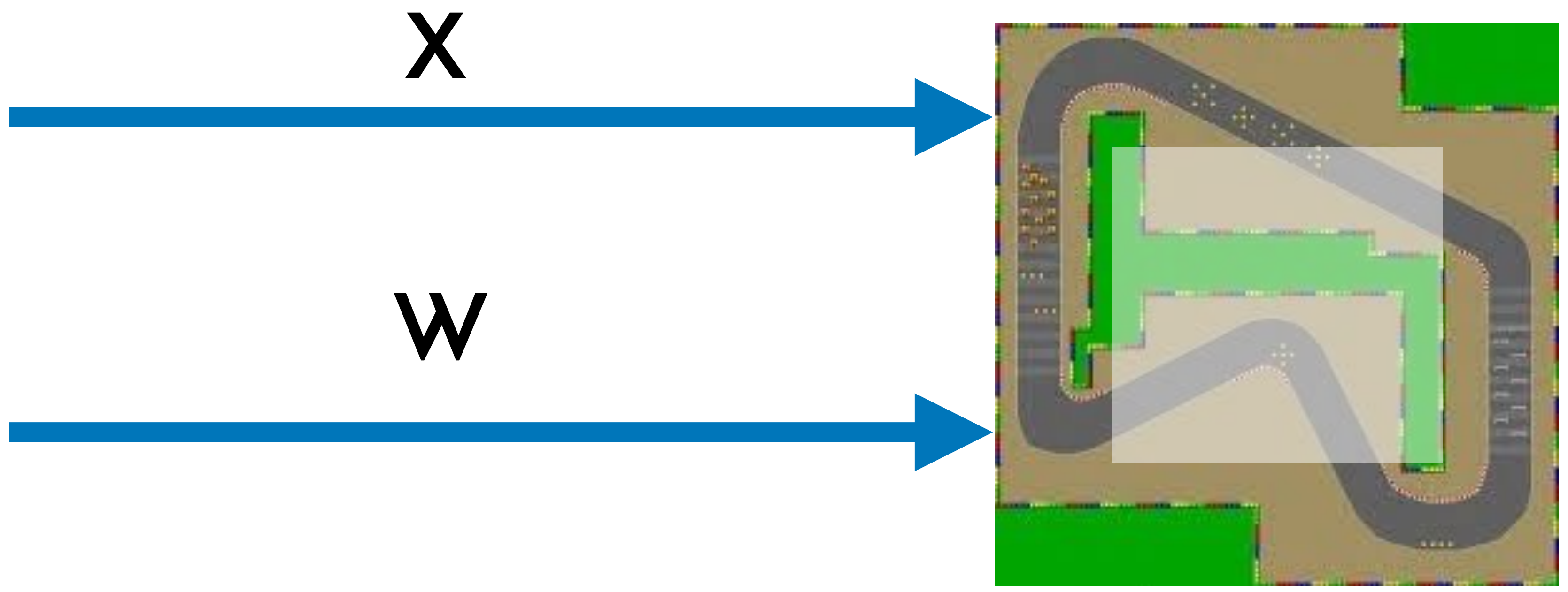
Plonk

Groth16

Composition (Intuition)

Long

$n = |\mathrm{x}| \approx 2^{26}$

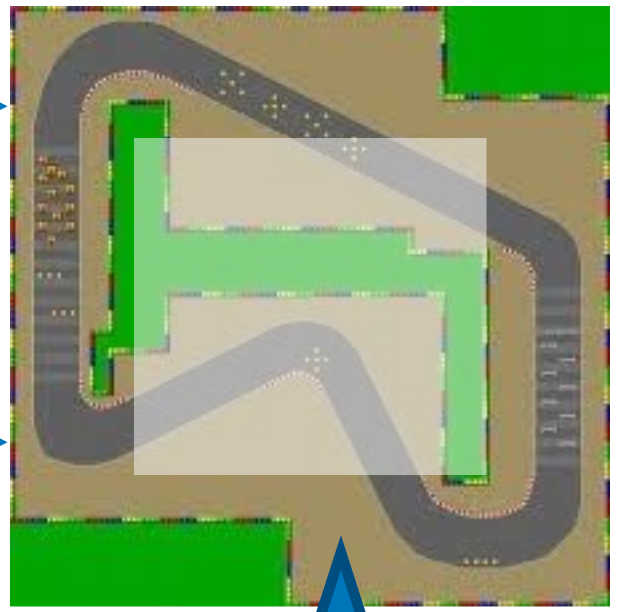


Composition (Intuition)

Long
$n = |\mathrm{x}| \approx 2^{26}$

x

w



Prove
$(\mathrm{x}, \mathrm{w}) \in \mathscr{R}$

Composition (Intuition)

Long

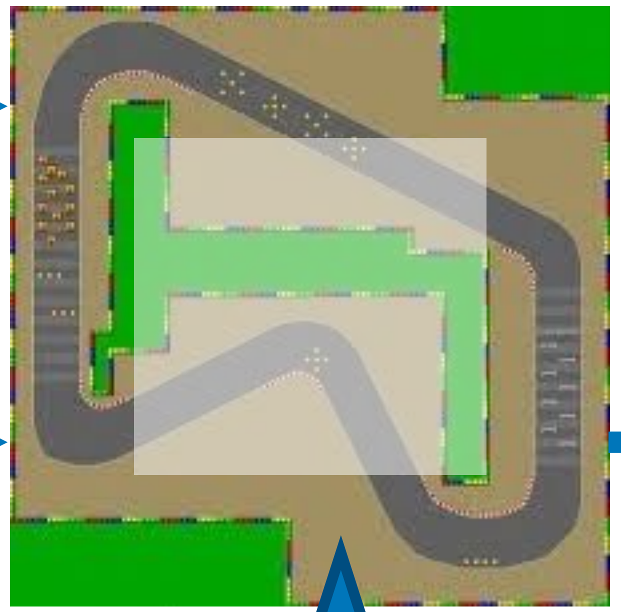
$n = |\mathrm{x}| \approx 2^{26}$

Semi-long

$\sqrt{n} = \sqrt{|\mathrm{x}|} \approx 2^{13}$?

x

w



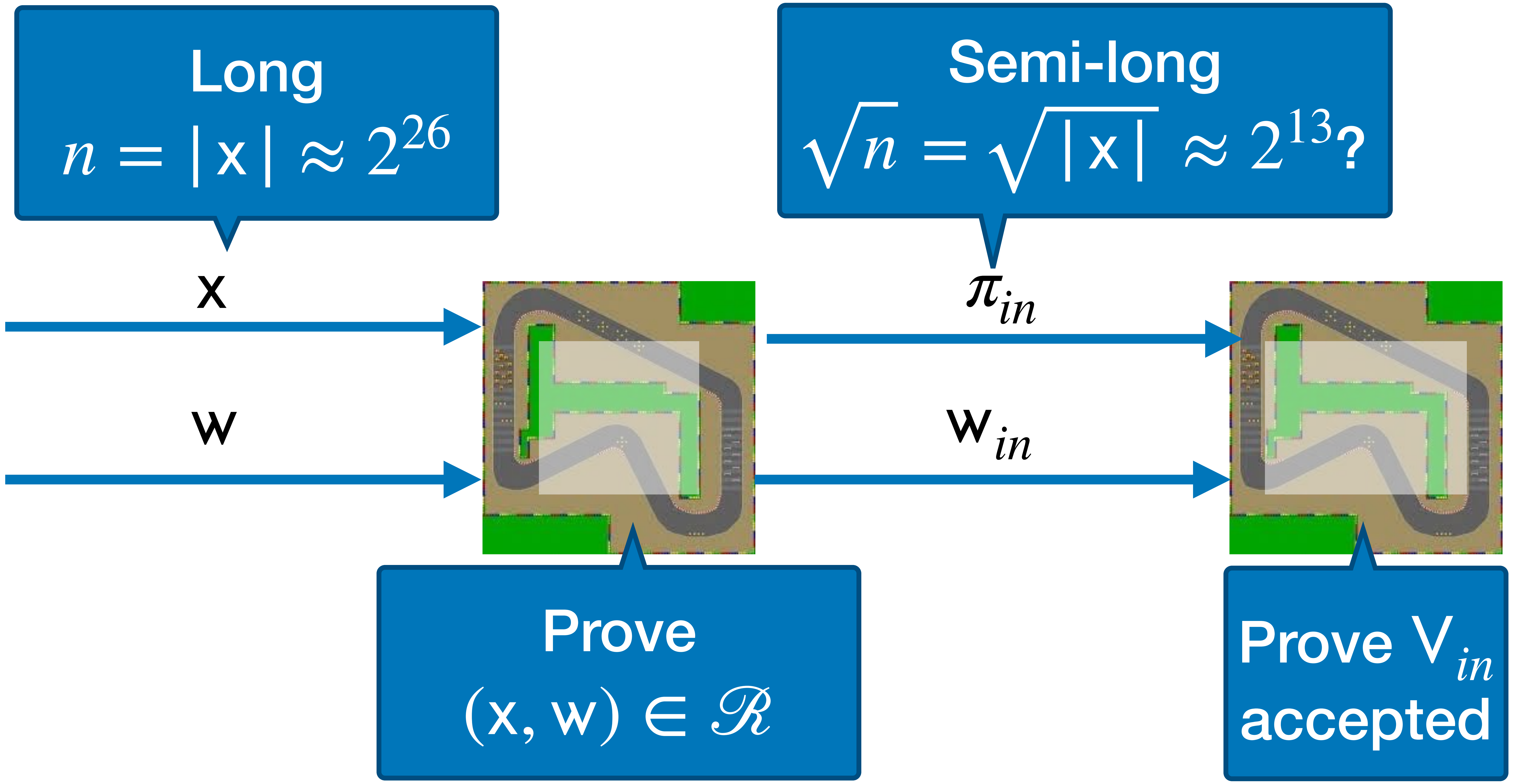
π_{in}

w_{in}

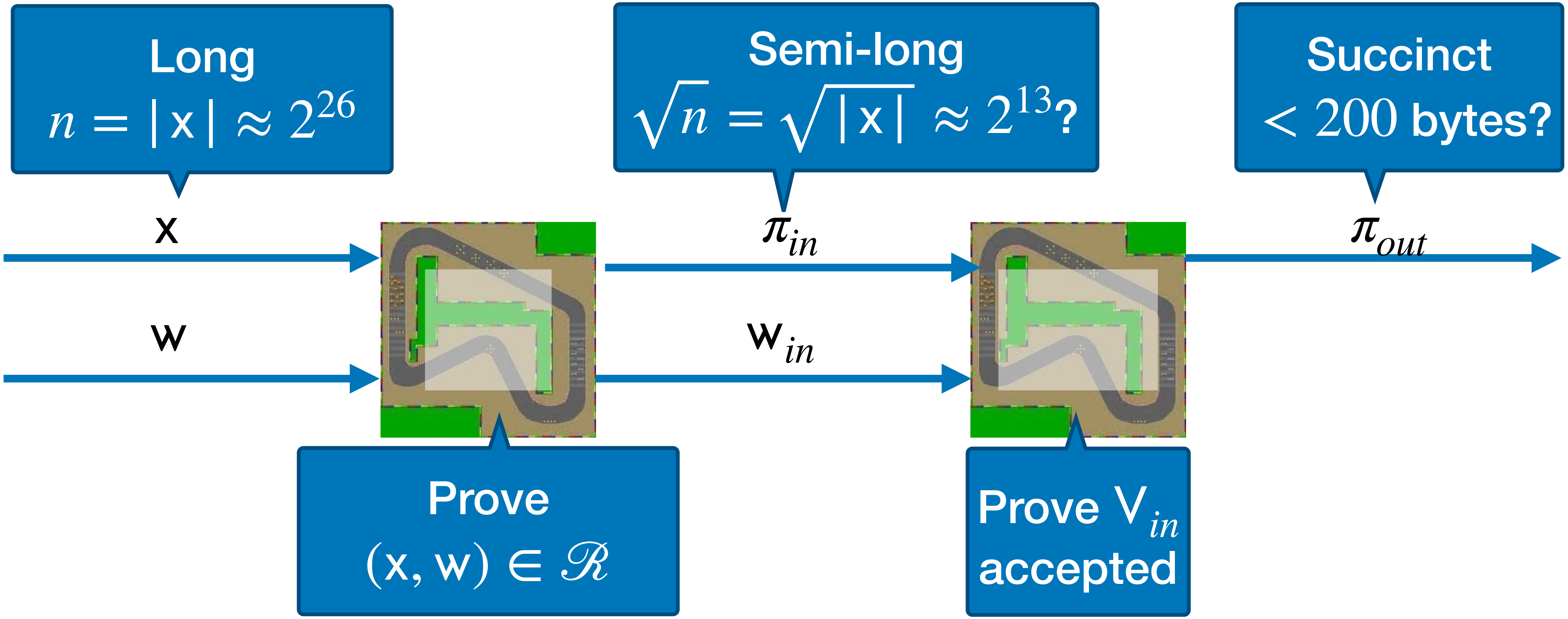
Prove

$(\mathrm{x}, \mathrm{w}) \in \mathscr{R}$

Composition (Intuition)



Composition (Intuition)



Composition (Intuition)

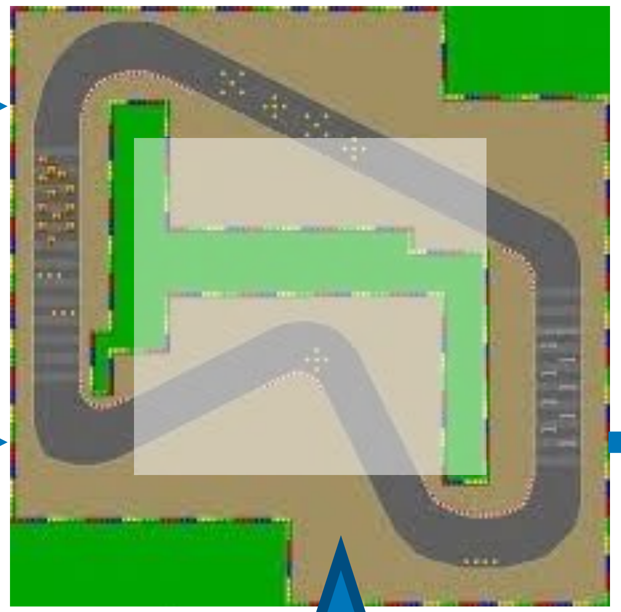
Long
$n = |\mathrm{x}| \approx 2^{26}$

Semi-long
$\sqrt{n} = \sqrt{|\mathrm{x}|} \approx 2^{13}$?

Succinct
< 200 bytes?

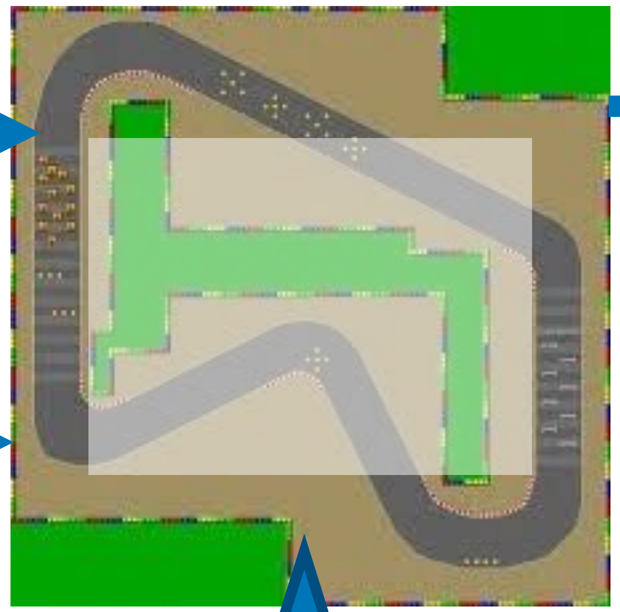
x

w



π_{in}

w_{in}



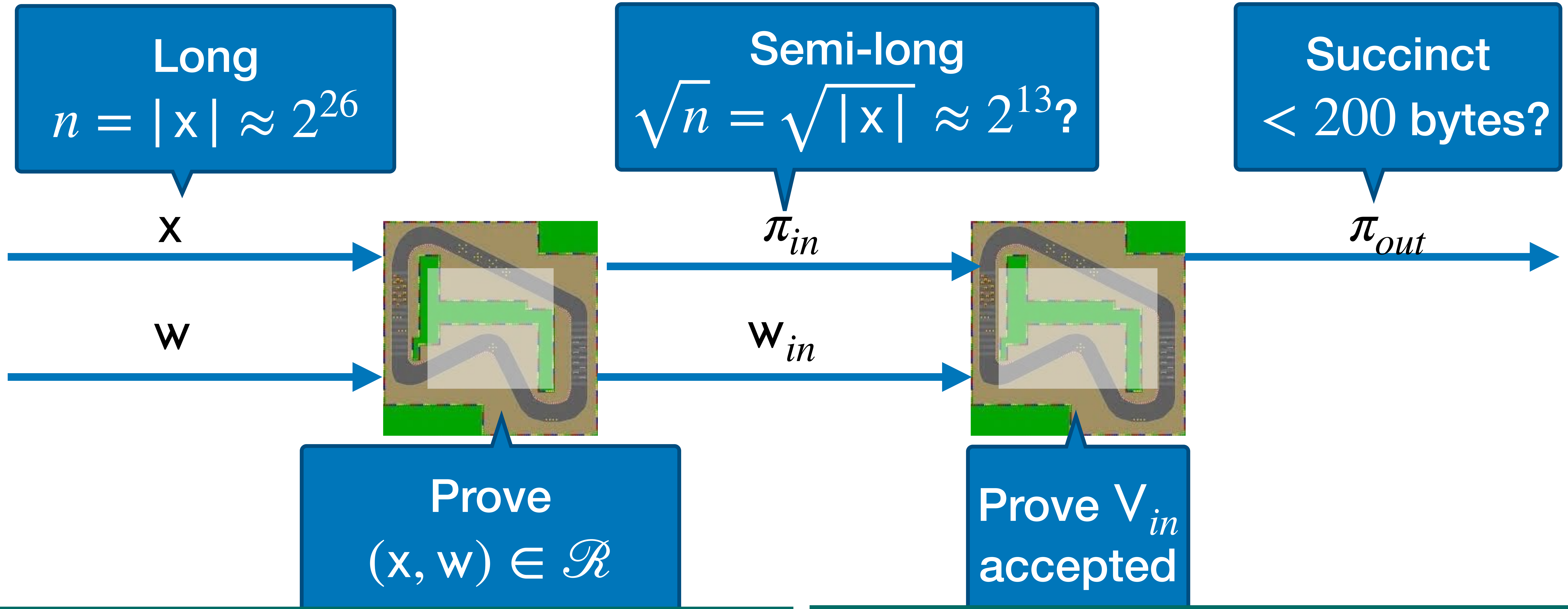
π_{out}

Prove
$(\mathrm{x}, \mathrm{w}) \in \mathscr{R}$

Prove V_{in} accepted

- Long input => need fast prover
- Proof has to be “short enough”
- Verifier’s circuit should be simple
- GKR, FRI, Brakedown, Bulletproofs, …

Composition (Intuition)



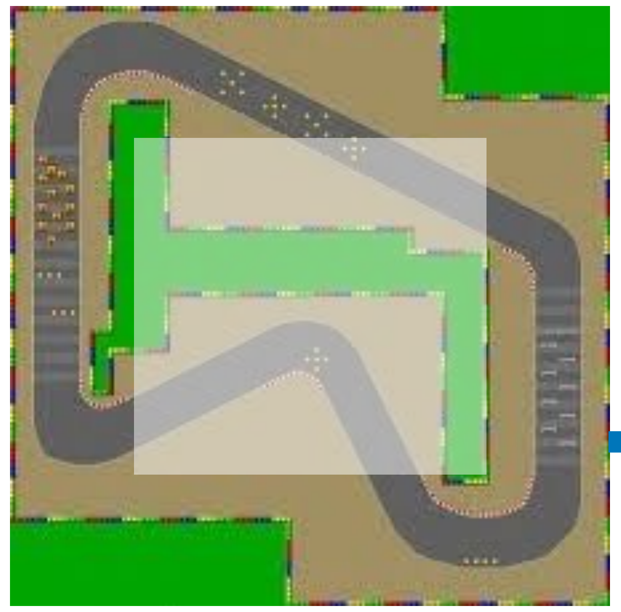
- Long input => need fast prover
- Proof has to be “short enough”
- Verifier’s circuit should be simple
- GKR, FRI, Brakedown, Bulletproofs, …

- Semi-long input => need decent prover
- Proof has to be “super succinct”
- Groth16!

Composition (Intuition)

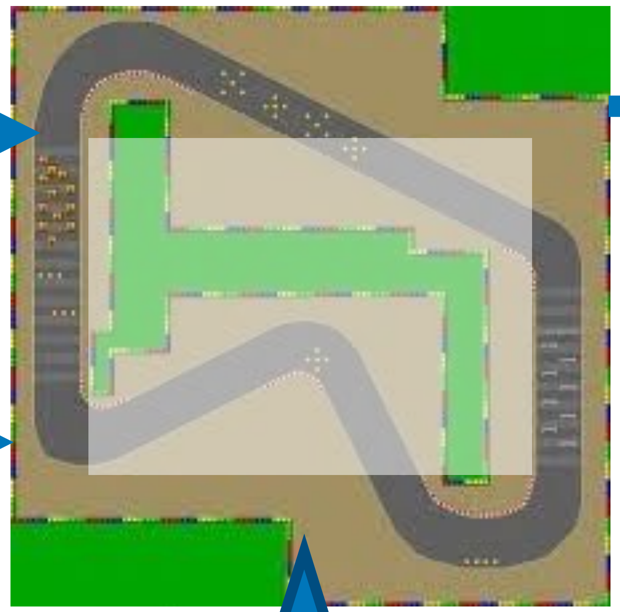
Semi-long
$\sqrt{n} = \sqrt{|x|} \approx 2^{13}$?

Succinct
< 200 bytes?



π_{in}

w_{in}



π_{out}

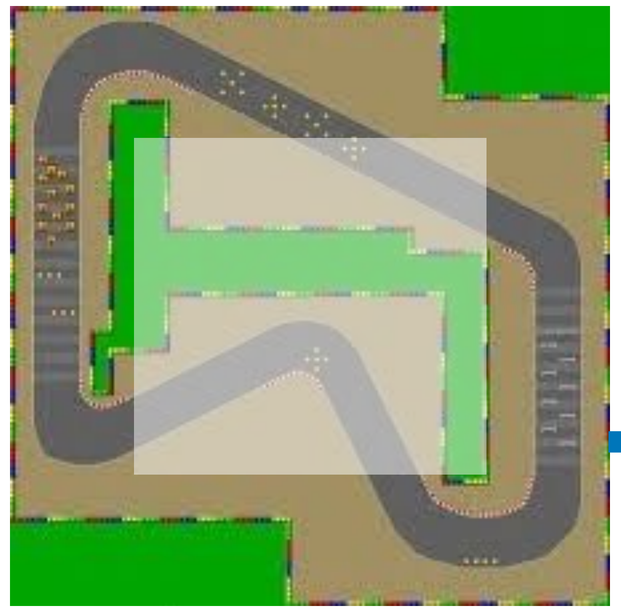
Prove V_{in} accepted

- Groth16 is non-universal?
  - "V_{in} accepts" is a fixed language
  - Non-universality is ok 😀
- Groth16 slow prover?
  - We apply Π_{out} to semi-long input 😀
- **But can we improve on $|\pi_{out}|$ and V speed?** 🤔

Composition (Intuition)

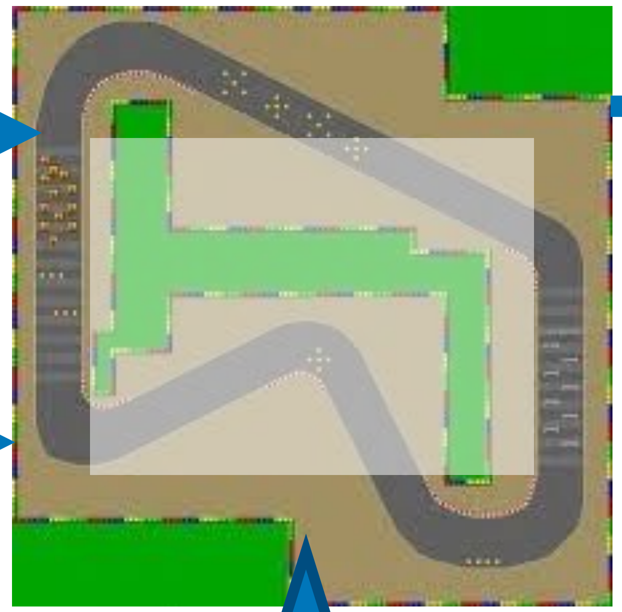
Semi-long
$\sqrt{n} = \sqrt{|x|} \approx 2^{13}$?

Succinct
< 200 bytes?



π_{in}

w_{in}



π_{out}

Prove V_{in} accepted

- Groth16 is non-universal?
  - "V_{in} accepts" is a fixed language
  - Non-universality is ok 😀
- Groth16 slow prover?
  - We apply Π_{out} to semi-long input 😀
- **But can we improve on $|\pi_{out}|$ and V speed?** 🤔

Groth16 => Polymath

Pros:

Groth16 => Polymath

Pros:

- Shorter argument

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x
- Batch-friendly

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x
- Batch-friendly
  - No adversarially output $\mathbb{G}_2$ elements

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x
- Batch-friendly
  - No adversarially output $\mathbb{G}_2$ elements

Cons:

Groth16 => Polymath

Pros:
- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x
- Batch-friendly
  - No adversarially output $\mathbb{G}_2$ elements

Cons:
- Slower prover

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x
- Batch-friendly
  - No adversarially output $\mathbb{G}_2$ elements

Cons:

- Slower prover
  - 4-8 times (no implementation**)

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x
- Batch-friendly
  - No adversarially output $\mathbb{G}_2$ elements

Cons:

- Slower prover
  - 4-8 times (no implementation**)
  - Should be ok in composition

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x
- Batch-friendly
  - No adversarially output $\mathbb{G}_2$ elements

Cons:

- Slower prover
  - 4-8 times (no implementation**)
  - Should be ok in composition
- Uses random oracle

Groth16 => Polymath

Pros:
- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x
- Batch-friendly
  - No adversarially output $\mathbb{G}_2$ elements

Cons:
- Slower prover
  - 4-8 times (no implementation**)
  - Should be ok in composition
- Uses random oracle
  - Mostly theoretical objection

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x
- Batch-friendly
  - No adversarially output $\mathbb{G}_2$ elements

Cons:

- Slower prover
  - 4-8 times (no implementation**)
  - Should be ok in composition
- Uses random oracle
  - Mostly theoretical objection
  - Initial SNARKs in composition use RO anyhow

Groth16 => Polymath

Pros:
- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x
- Batch-friendly
  - No adversarially output $\mathbb{G}_2$ elements

Cons:
- Slower prover
  - 4-8 times (no implementation**)
  - Should be ok in composition
- Uses random oracle
  - Mostly theoretical objection
  - Initial SNARKs in composition use RO anyhow
  - But RO makes certain apps difficult

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x
- Batch-friendly
  - No adversarially output $\mathbb{G}_2$ elements

Cons:

- Slower prover
  - 4-8 times (no implementation**)
  - Should be ok in composition
- Uses random oracle
  - Mostly theoretical objection
  - Initial SNARKs in composition use RO anyhow
  - But RO makes certain apps difficult
- No SnarkPack?

Groth16 => Polymath

Pros:

- Shorter argument
  - 1536 -> 1408 bits @ 128-bit
  - 3072 -> 1792 bits @ 192-bit
- Faster verifier
  - Potentially, for short x
  - Definitely, for long x
- Batch-friendly
  - No adversarially output $\mathbb{G}_2$ elements

Cons:

- Slower prover
  - 4-8 times (no implementation**)
  - Should be ok in composition
- Uses random oracle
  - Mostly theoretical objection
  - Initial SNARKs in composition use RO anyhow
  - But RO makes certain apps difficult
- No SnarkPack?
  - Not known how to batch pre-existing proofs

Pairings

For Muggles

- $\mathrm{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, [1]_1, [1]_2, \hat{e})$

Pairings

For Muggles

- $\mathrm{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, [1]_1, [1]_2, \hat{e})$
  - $\mathbb{G}_i$ are additive abelian groups of large prime order p

Pairings

For Muggles

- $\mathrm{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, [1]_1, [1]_2, \hat{e})$
  - $\mathbb{G}_i$ are additive abelian groups of large prime order p
  - $[1]_i$ is a generator of $\mathbb{G}_i$

Pairings

For Muggles

- $\mathrm{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, [1]_1, [1]_2, \hat{e})$
  - $\mathbb{G}_i$ are additive abelian groups of large prime order p
  - $[1]_i$ is a generator of $\mathbb{G}_i$
  - $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear map

Pairings

For Muggles

- $\mathrm{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, [1]_1, [1]_2, \hat{e})$
  - $\mathbb{G}_i$ are additive abelian groups of large prime order p
  - $[1]_i$ is a generator of $\mathbb{G}_i$
  - $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear map
  - $\hat{e}([a]_1, [b]_2) = [ab]_T$ for $a, b \in \mathbb{F}_p$

Pairings

For Muggles

- $\mathrm{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, [1]_1, [1]_2, \hat{e})$
  - $\mathbb{G}_i$ are additive abelian groups of large prime order p
  - $[1]_i$ is a generator of $\mathbb{G}_i$
  - $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear map
  - $\hat{e}([a]_1, [b]_2) = [ab]_T$ for $a, b \in \mathbb{F}_p$
- “Standard” curve for 128-bit security level: BLS12-381

Pairings

For Muggles

- $\mathrm{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, [1]_1, [1]_2, \hat{e})$
  - $\mathbb{G}_i$ are additive abelian groups of large prime order p
  - $[1]_i$ is a generator of $\mathbb{G}_i$
  - $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear map
  - $\hat{e}([a]_1, [b]_2) = [ab]_T$ for $a, b \in \mathbb{F}_p$
- "Standard" curve for 128-bit security level: BLS12-381
  - $\ell(\mathbb{F}_p) = 256$, $\ell(\mathbb{G}_1) = 384$, $\ell(\mathbb{G}_2) = 2\ell(\mathbb{G}_1) = 768$ (bits)

Pairings

For Muggles

- $\mathrm{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, [1]_1, [1]_2, \hat{e})$
  - $\mathbb{G}_i$ are additive abelian groups of large prime order p
  - $[1]_i$ is a generator of $\mathbb{G}_i$
  - $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear map
  - $\hat{e}([a]_1, [b]_2) = [ab]_T$ for $a, b \in \mathbb{F}_p$
- "Standard" curve for 128-bit security level: BLS12-381
  - $\ell(\mathbb{F}_p) = 256$, $\ell(\mathbb{G}_1) = 384$, $\ell(\mathbb{G}_2) = 2\ell(\mathbb{G}_1) = 768$ (bits)
- Curves for 192-bit security level:

Pairings

For Muggles

- $\mathrm{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, [1]_1, [1]_2, \hat{e})$
  - $\mathbb{G}_i$ are additive abelian groups of large prime order p
  - $[1]_i$ is a generator of $\mathbb{G}_i$
  - $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear map
  - $\hat{e}([a]_1, [b]_2) = [ab]_T$ for $a, b \in \mathbb{F}_p$
- "Standard" curve for 128-bit security level: BLS12-381
  - $\ell(\mathbb{F}_p) = 256, \ell(\mathbb{G}_1) = 384, \ell(\mathbb{G}_2) = 2\ell(\mathbb{G}_1) = 768$ (bits)
- Curves for 192-bit security level:
  - $\ell(\mathbb{F}_p) = 256, \ell(\mathbb{G}_1) = 512, \ell(\mathbb{G}_2) = 4\ell(\mathbb{G}_1) = 2048$ (bits)

Groth16: Bird's-Eye View

Computation: f
Public input (statement) x
Private input (witness) w

$\mathrm{srs}(f)$

$\mathrm{srs}(f)$

Computation: f
Public input (statement) x

- SRS depends on the circuit

Groth16: Bird's-Eye View

Computation: f

Public input (statement) x

Private input (witness) w

$\mathrm{srs}(f)$

$\mathrm{srs}(f)$

Computation: f

Public input (statement) x

$\pi = ([a]_1, [b]_2, [c]_1)$

- SRS depends on the circuit
- Argument length: only 3 group elements

Groth16: Bird's-Eye View

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

$\mathrm{srs}(f)$

$\mathrm{srs}(f)$

$\pi = ([a]_1, [b]_2, [c]_1)$

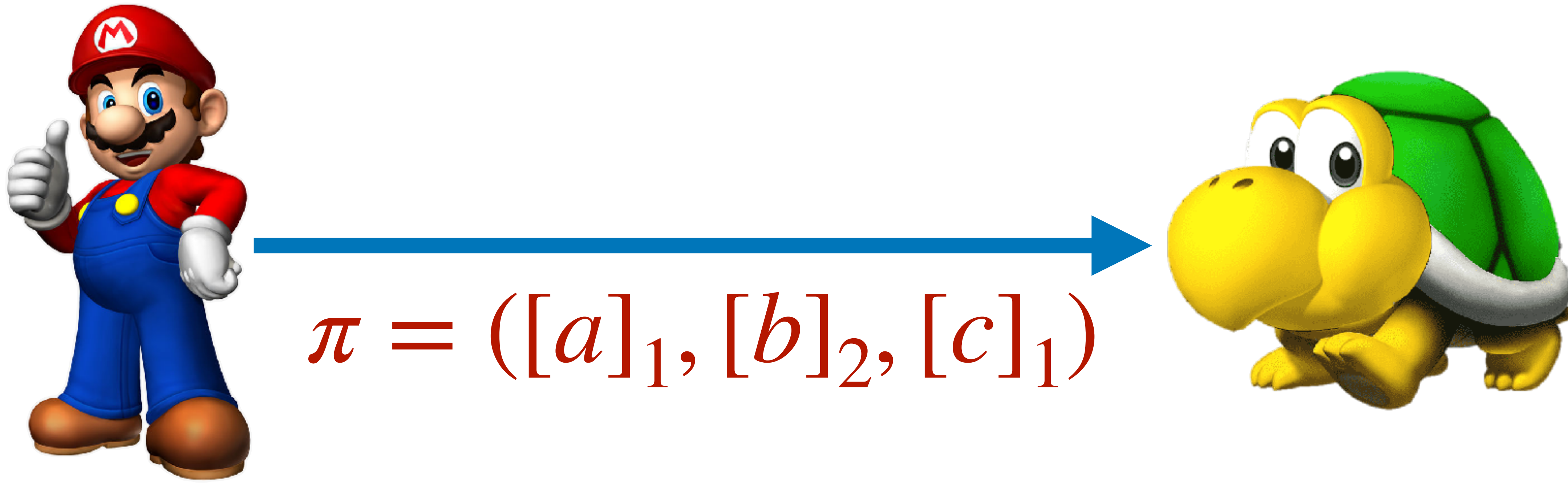
- SRS depends on the circuit
- Argument length: only 3 group elements
- Verifier executes three pairings and |x| group ops

On Optimality

$\pi = ([a]_1, [b]_2, [c]_1)$

- Groth16 has **three** group elements

On Optimality



- Groth16 has **three** group elements
- Lower bound [Groth, EC16]:

On Optimality

$\pi = ([a]_1, [b]_2, [c]_1)$

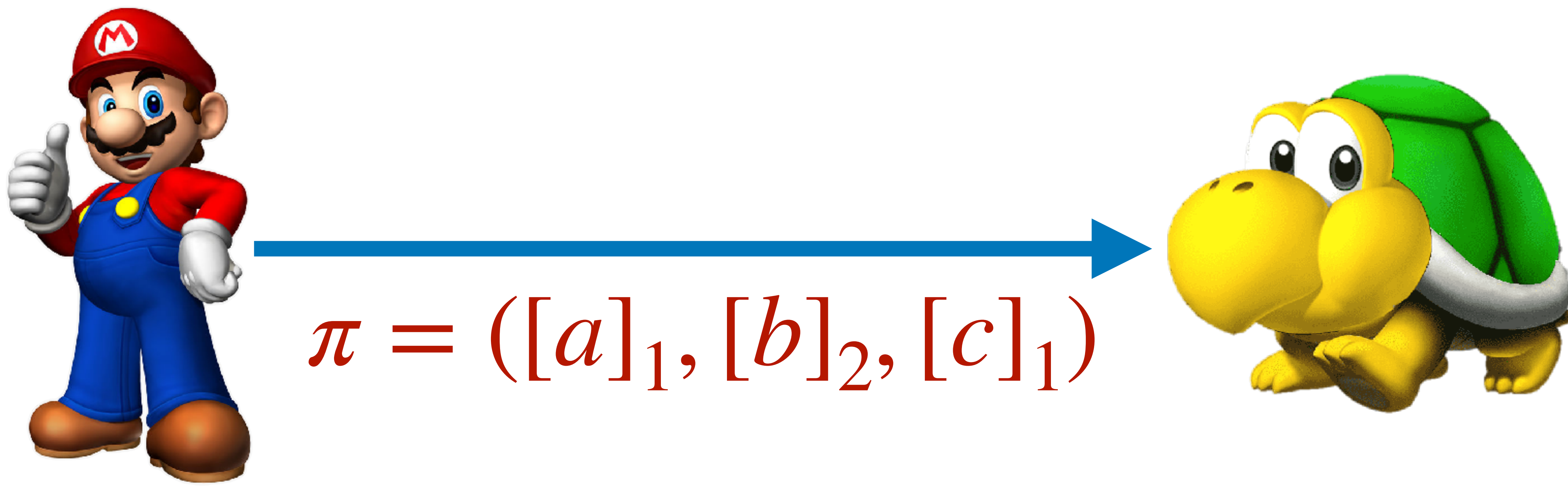
- Groth16 has **three** group elements
- Lower bound [Groth, EC16]:
  - **At least two group elements** needed

On Optimality

$\pi = ([a]_1, [b]_2, [c]_1)$

- Groth16 has **three** group elements
- Lower bound [Groth, EC16]:
  - **At least two group elements** needed
  - One of them has to be in $\mathbb{G}_2$

On Optimality



$$\pi = ([a]_1, [b]_2, [c]_1)$$

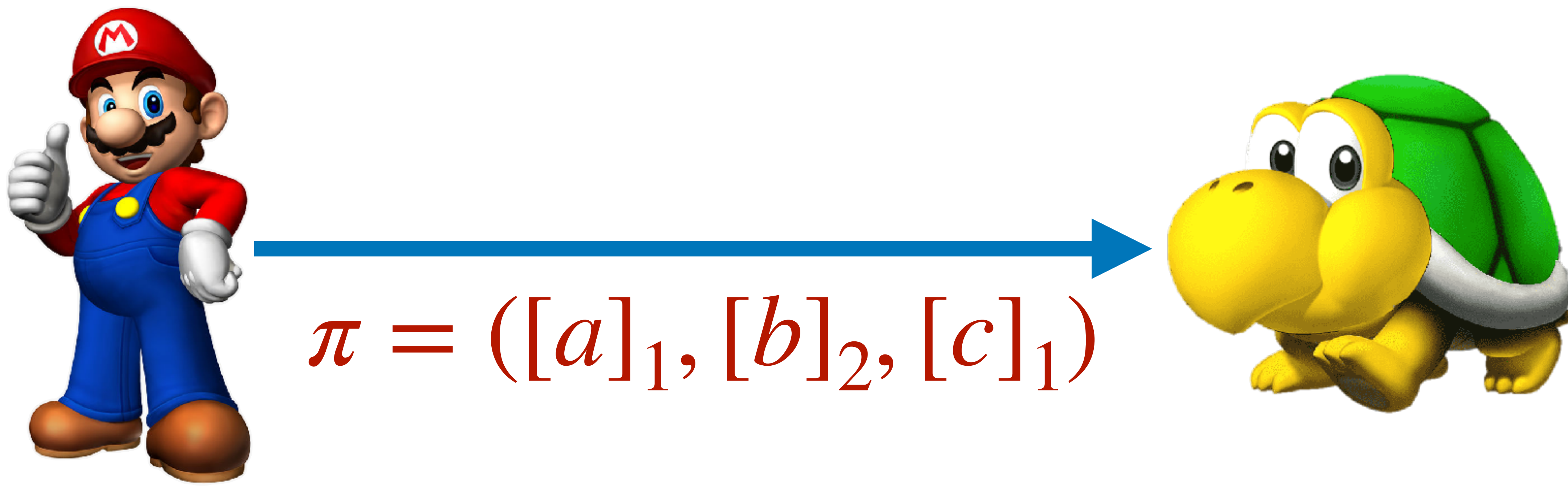
- Groth16 has **three** group elements
- Lower bound [Groth, EC16]:
  - **At least two group elements** needed
  - One of them has to be in $\mathbb{G}_2$
- Using a different arithmetization, one can have **two** group elements

On Optimality

$$\pi = ([a]_1, [b]_2, [c]_1)$$

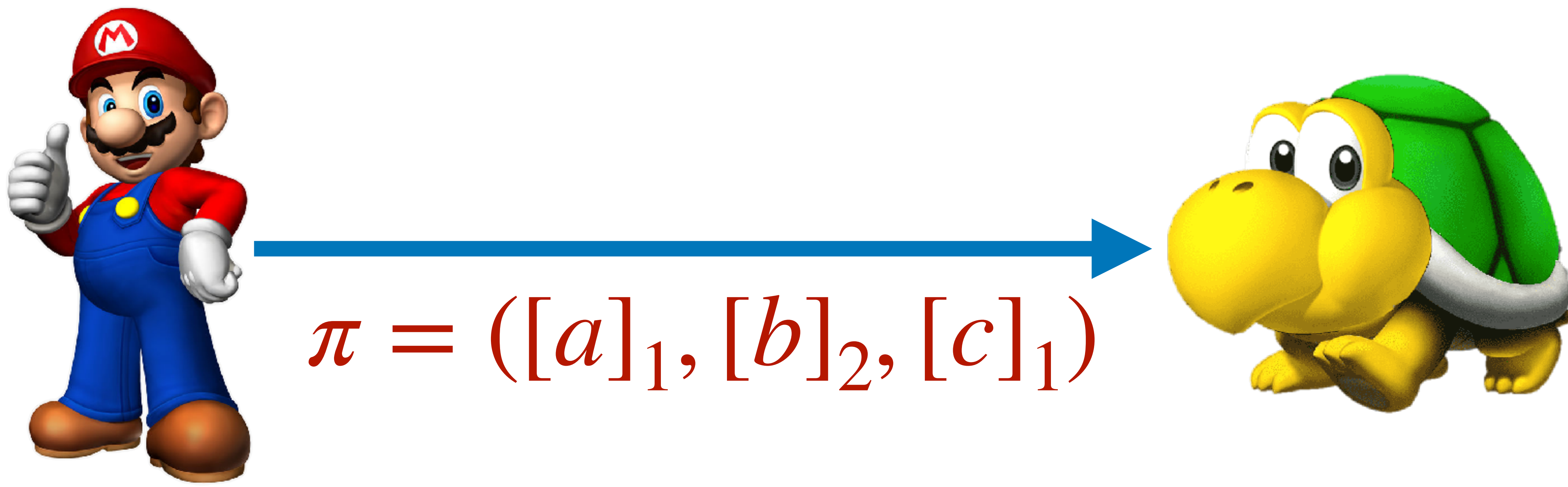
- Groth16 has **three** group elements
- Lower bound [Groth, EC16]:
  - **At least two group elements** needed
  - One of them has to be in $\mathbb{G}_2$
- Using a different arithmetization, one can have **two** group elements
  - **SAP** (Square Arithmetic Programming) instead of **R1CS**

On Optimality



- Groth16 has **three** group elements
- Lower bound [Groth, EC16]:
  - **At least two group elements** needed
  - One of them has to be in $\mathbb{G}_2$
- Using a different arithmetization, one can have **two** group elements
  - **SAP** (Square Arithmetic Programming) instead of **R1CS**
  - However, then one relies on **symmetric** pairings

On Optimality



- Groth16 has **three** group elements
- Lower bound [Groth, EC16]:
  - **At least two group elements** needed
  - One of them has to be in $\mathbb{G}_2$
- Using a different arithmetization, one can have **two** group elements
  - **SAP** (Square Arithmetic Programming) instead of **R1CS**
  - However, then one relies on **symmetric** pairings
  - Group elements will be considerably longer => **worse in practice**

On Optimality

$\pi = ([a]_1, [b]_2, [c]_1)$

- Groth16 has **three** group elements
- Lower bound [Groth, EC16]:
  - **At least two group elements** needed
  - One of them has to be in $\mathbb{G}_2$
- Using a different arithmetization, one can have **two** group elements
  - **SAP** (Square Arithmetic Programming) instead of **R1CS**
  - However, then one relies on **symmetric** pairings
  - Group elements will be considerably longer => **worse in practice**

- The lower bound works in the **standard model** (no RO)

On Optimality

$$\pi = ([a]_1, [b]_2, [c]_1)$$

- Groth16 has **three** group elements
- Lower bound [Groth, EC16]:
  - **At least two group elements** needed
  - One of them has to be in $\mathbb{G}_2$
- Using a different arithmetization, one can have **two** group elements
  - **SAP** (Square Arithmetic Programming) instead of **R1CS**
  - However, then one relies on **symmetric** pairings
  - Group elements will be considerably longer => **worse in practice**

- The lower bound works in the **standard model** (no RO)
- It talks about **#group elements**, not **bit-length**

Scenic Route to Polymath

For non-muggles

$\pi = ([a]_1, [b]_2, [c]_1)$

- Problem: $\mathbb{G}_2$ elements are long

Scenic Route to Polymath

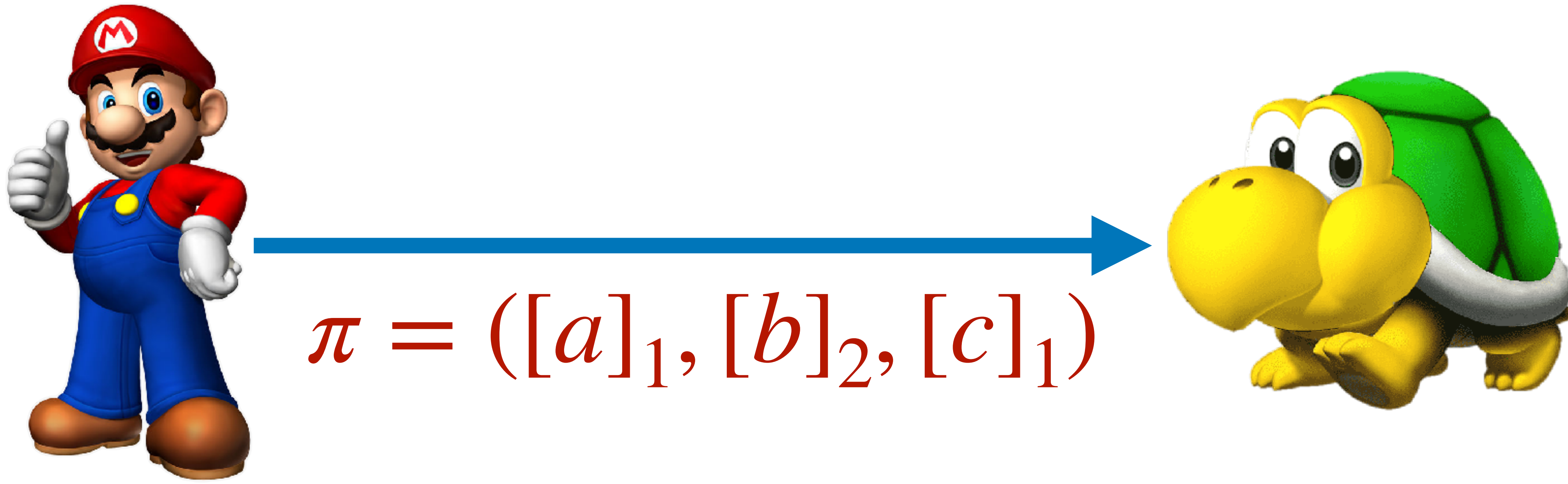
$\pi = ([a]_1, [b]_2, [c]_1)$

For non-muggles

- Problem: $\mathbb{G}_2$ elements are long
- $[b]_2 \implies [b]_1$, but how?

Scenic Route to Polymath

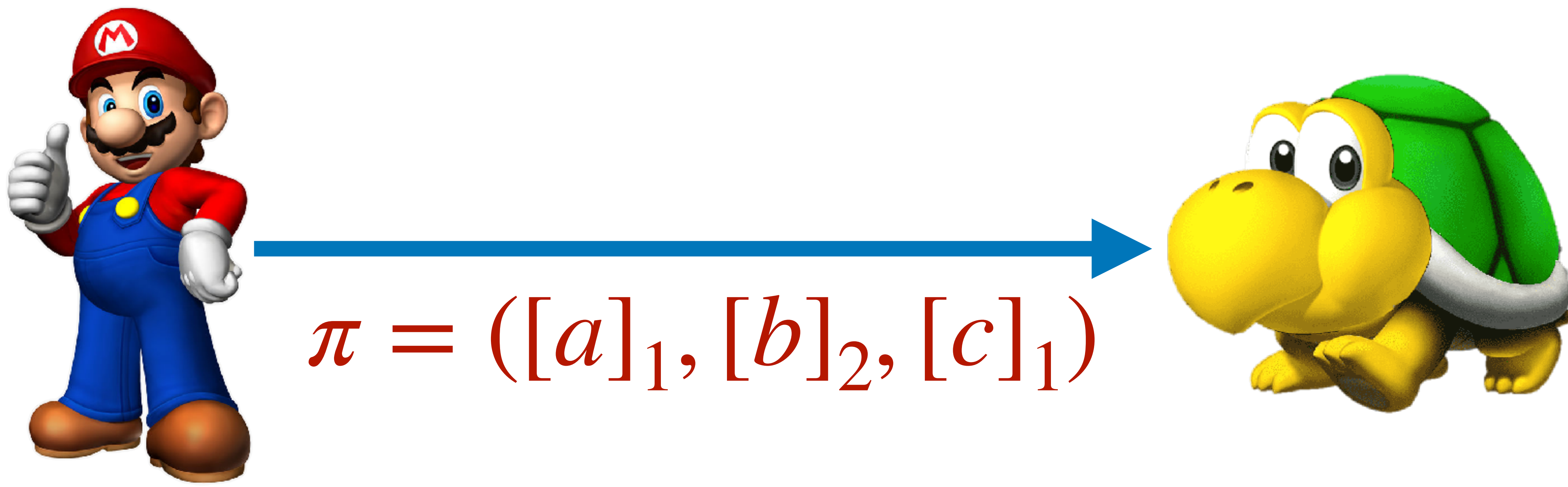
For non-muggles



- Problem: $\mathbb{G}_2$ elements are long
- $[b]_2 \implies [b]_1$, but how?
- Groth16 uses pairings to do quadratic checks

Scenic Route to Polymath

For non-muggles



- Problem: $\mathbb{G}_2$ elements are long
- $[b]_2 \implies [b]_1$, but how?
- Groth16 uses pairings to do quadratic checks
- We can KZG-open the polynomial commitment $[b]_1$ to some $\bar{b}$ and do quadratic checks by using $\bar{b}$

Scenic Route to Polymath

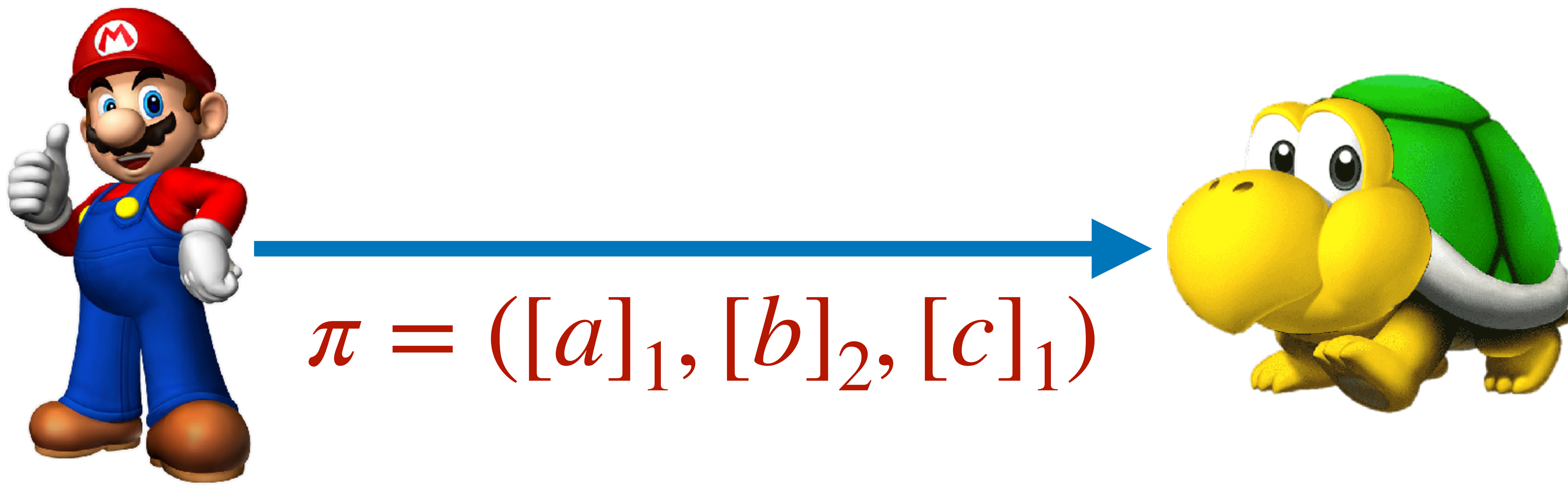
For non-muggles

$\pi = ([a]_1, [b]_2, [c]_1)$

- Problem: $\mathbb{G}_2$ elements are long
- $[b]_2 \implies [b]_1$, but how?
- Groth16 uses pairings to do quadratic checks
- We can KZG-open the polynomial commitment $[b]_1$ to some $\bar{b}$ and do quadratic checks by using $\bar{b}$
- KZG opening is shorter than a $\mathbb{G}_2$ element

Scenic Route to Polymath

For non-muggles

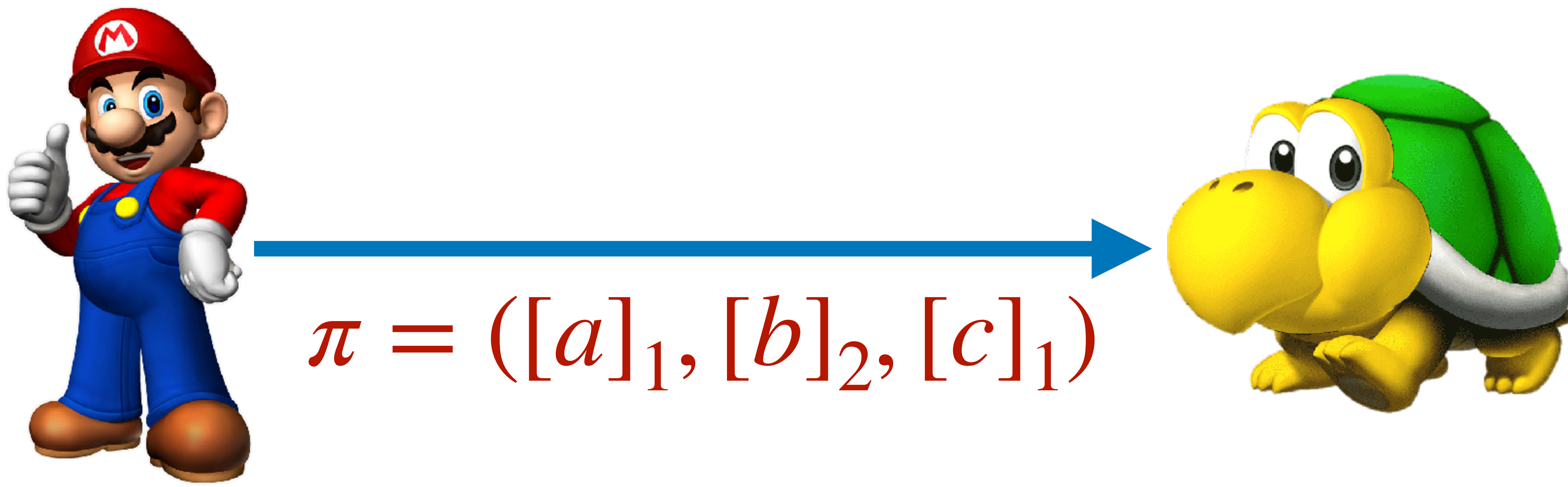


- Problem: $\mathbb{G}_2$ elements are long
- $[b]_2 \implies [b]_1$, but how?
- Groth16 uses pairings to do quadratic checks

We can KZG-open the polynomial commitment $[b]_1$ to some $\bar{b}$ and do quadratic checks by using $\bar{b}$

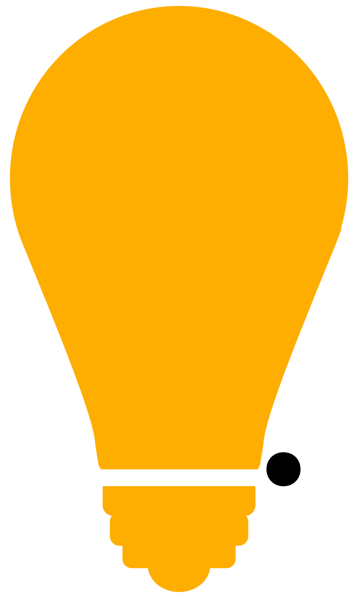
- KZG opening is shorter than a $\mathbb{G}_2$ element
  - (a field element $\bar{b}$ and a $\mathbb{G}_1$ element $[h]_1$)

Scenic Route to Polymath



For non-muggles

- Problem: $\mathbb{G}_2$ elements are long
- $[b]_2 \Longrightarrow [b]_1$, but how?
- Groth16 uses pairings to do quadratic checks

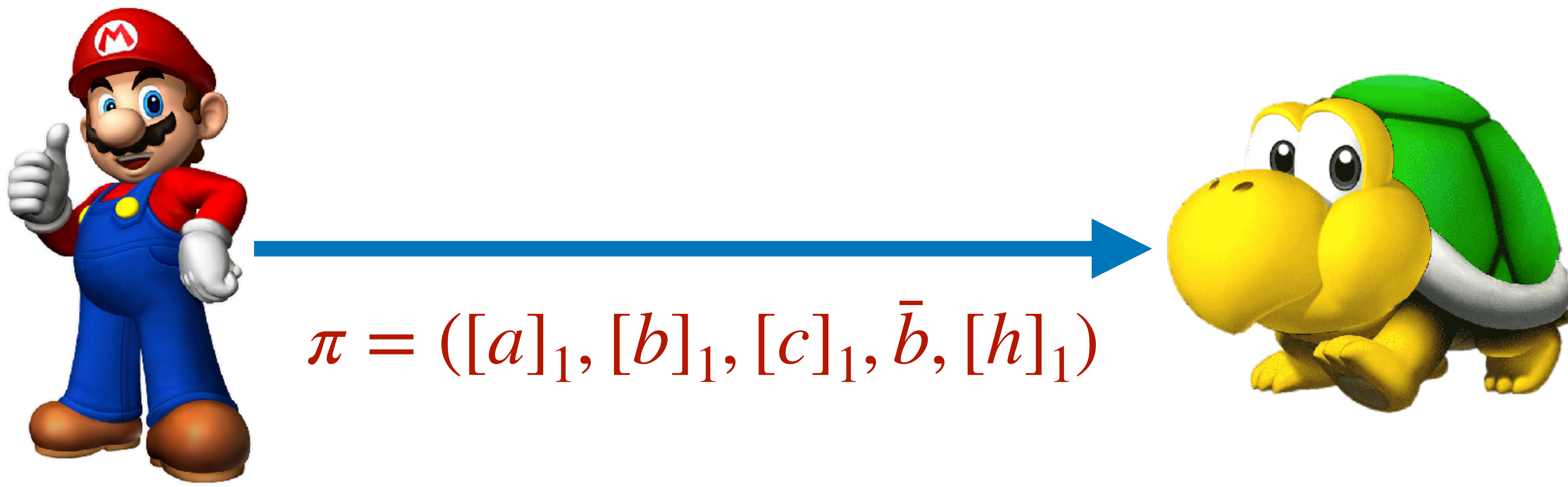


- We can KZG-open the polynomial commitment $[b]_1$ to some $\bar{b}$ and do quadratic checks by using $\bar{b}$
- KZG opening is shorter than a $\mathbb{G}_2$ element
  - (a field element $\bar{b}$ and a $\mathbb{G}_1$ element $[h]_1$)

Problem:

- we still have $[b]_1$ in the argument!
- $\ell([b]_2) < \ell([b]_1) + \ell(\bar{b}) + \ell([h]_1)$ in 128-bit level

Scenic Route to Polymath



If we use SAP instead of R1CS, we get $[b]_1 = [a]_1$

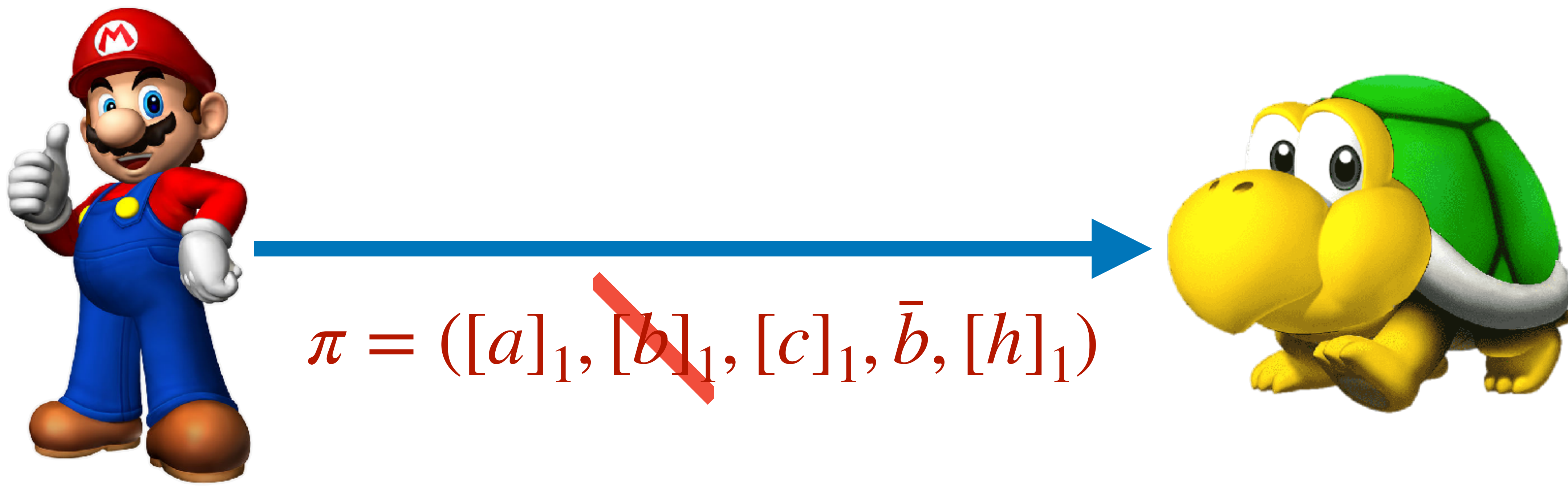
Scenic Route to Polymath

$\pi = ([a]_1, \cancel{[b]_1}, [c]_1, \bar{b}, [h]_1)$

If we use SAP instead of R1CS, we get $[b]_1 = [a]_1$

- No need to send it!

Scenic Route to Polymath



If we use SAP instead of R1CS, we get $[b]_1 = [a]_1$

- No need to send it!

- **Cost:** circuit ≈ 2 longer => slower prover

Scenic Route to Polymath

$\pi = ([a]_1, \cancel{[b]_1}, [c]_1, \bar{b}, [h]_1)$

If we use SAP instead of R1CS, we get $[b]_1 = [a]_1$

- No need to send it!

- **Cost:** circuit ≈ 2 longer => slower prover
  - Multiplication gates => squaring gates

Scenic Route to Polymath

$\pi = ([a]_1, \cancel{[b]_1}, [c]_1, \bar{b}, [h]_1)$

If we use SAP instead of R1CS, we get $[b]_1 = [a]_1$

- No need to send it!

- **Cost:** circuit ≈ 2 longer => slower prover
  - Multiplication gates => squaring gates

Problem:
- Groth16 has five trapdoors, KZG is univariate
- Not clear how to use KZG

Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

- Univariatization:

Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

- Univariatization:
    - Replace each trapdoor with x^i for some i and a single trapdoor x

Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

- Univariatization:
  - Replace each trapdoor with x^i for some i and a single trapdoor x
- Also done in [Lipmaa, PKC 2022] who used exhaustive search to find i's

Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

- Univariatization:
  - Replace each trapdoor with x^i for some i and a single trapdoor x
- Also done in [Lipmaa, PKC 2022] who used exhaustive search to find i's

Problem:

- even after exhaustive search, the exponents i are quite large
- KZG prover time Ω(polynomial degree)
  - => Results in high prover complexity

Scenic Route to Polymath

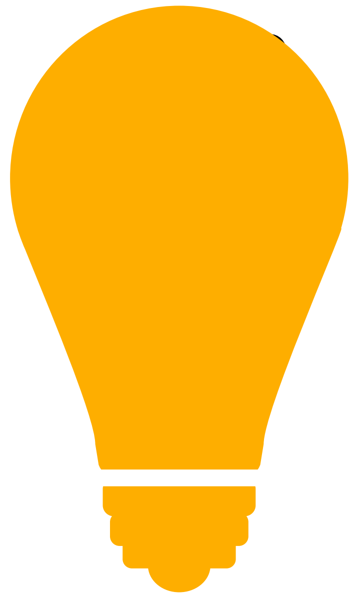
$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

- **Observation 1:** Groth16 for SAP has -1 trapdoor

Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

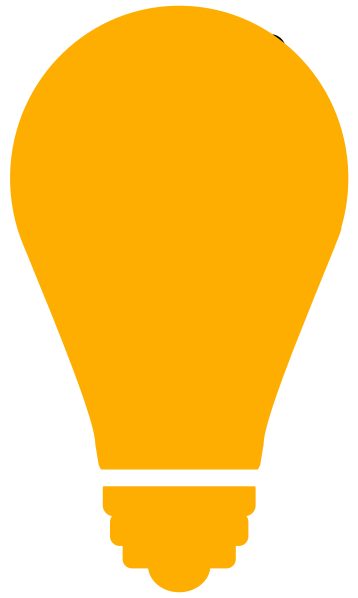
- **Observation 1:** Groth16 for SAP has -1 trapdoor
- **Observation 2:**



Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

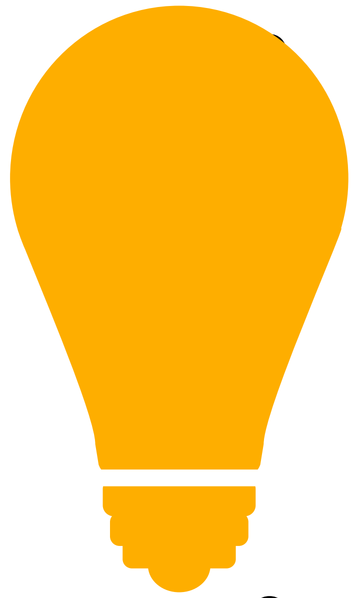
- **Observation 1:** Groth16 for SAP has -1 trapdoor



- **Observation 2:**
  - One trapdoor in Groth16 is only needed to mask SRS elements corresponding to the statement

Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

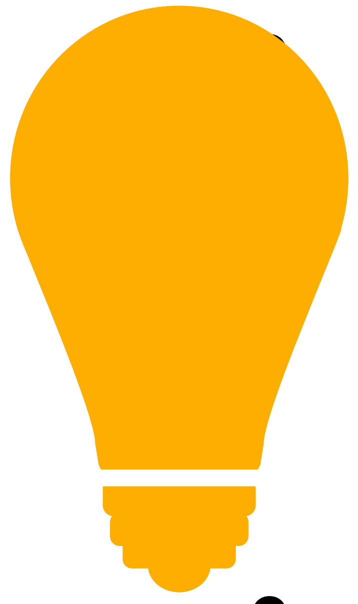


- **Observation 1:** Groth16 for SAP has -1 trapdoor
- **Observation 2:**
  - One trapdoor in Groth16 is only needed to mask SRS elements corresponding to the statement
- We use a different verification algorithm, getting rid of "statement" trapdoor

Scenic Route to Polymath

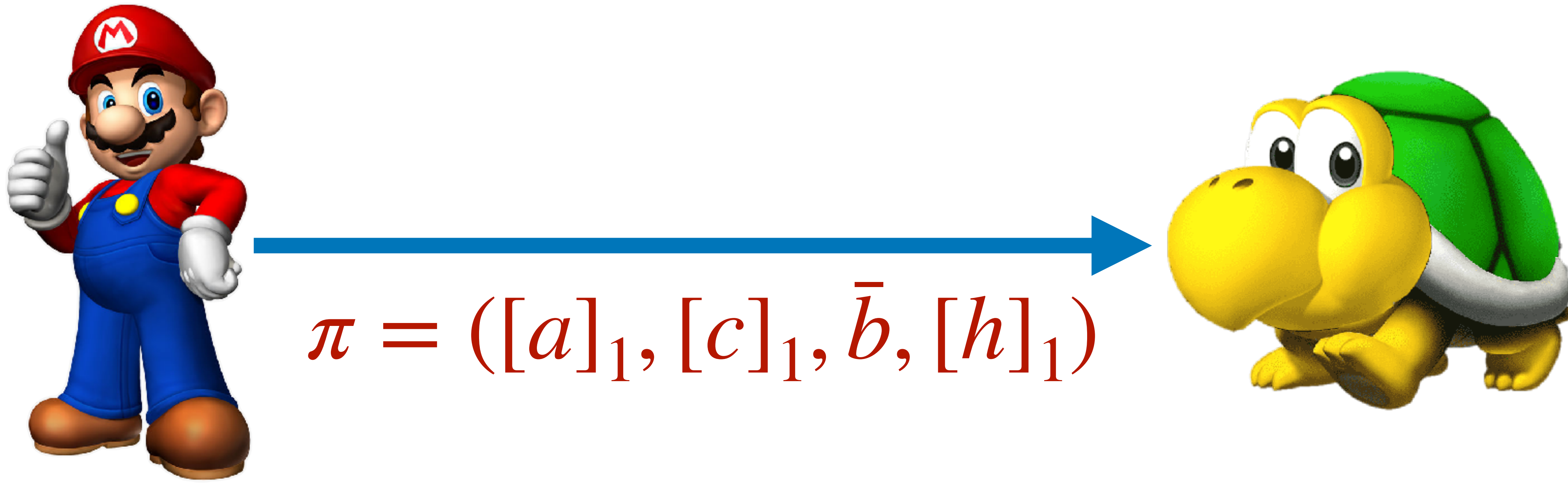
$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

- **Observation 1:** Groth16 for SAP has -1 trapdoor



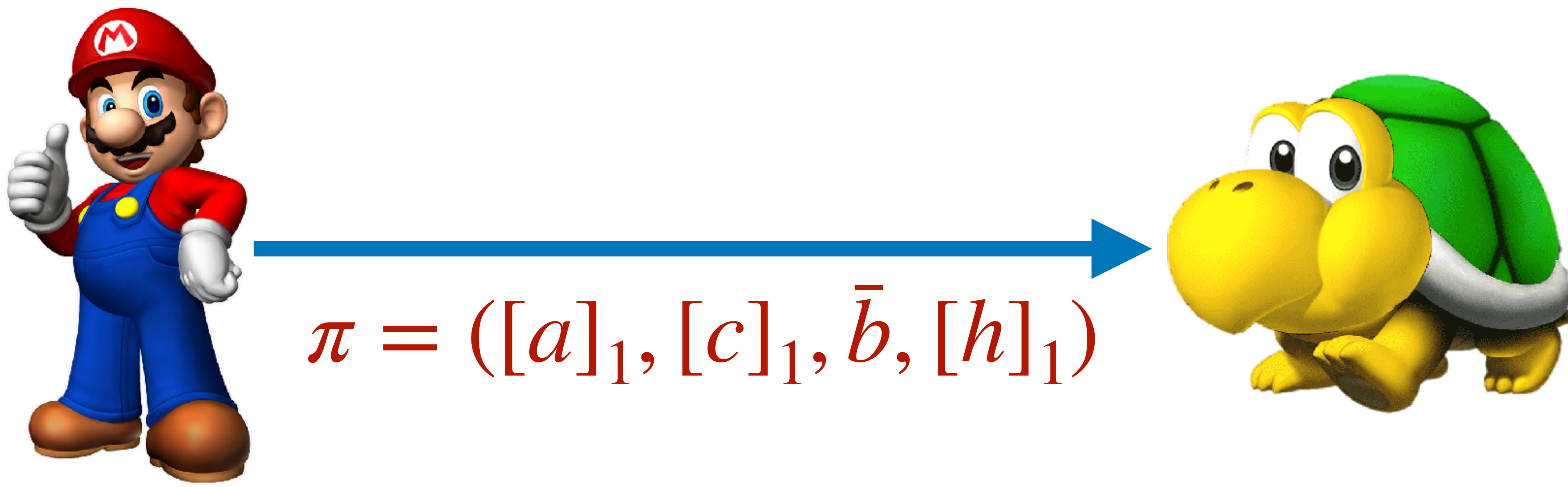
- **Observation 2:**
    - One trapdoor in Groth16 is only needed to mask SRS elements corresponding to the statement
- We use a different verification algorithm, getting rid of "statement" trapdoor
    - Verifier becomes faster

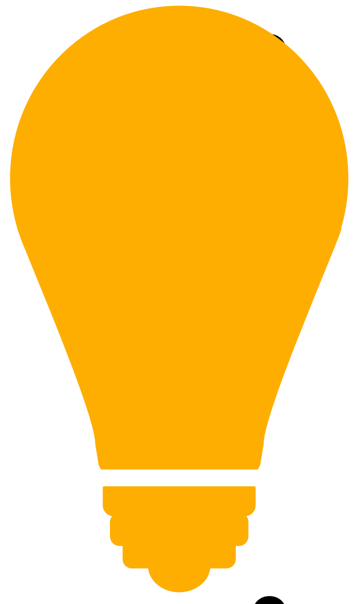
Scenic Route to Polymath



- **Observation 1:** Groth16 for SAP has -1 trapdoor
- **Observation 2:**
  - One trapdoor in Groth16 is only needed to mask SRS elements corresponding to the statement
- We use a different verification algorithm, getting rid of "statement" trapdoor
  - Verifier becomes faster
  - In Polymath, V interpolates a polynomial in $\mathbb{F}_p$ of degree $|x|$

Scenic Route to Polymath





- **Observation 1:** Groth16 for SAP has -1 trapdoor
- **Observation 2:**
  - One trapdoor in Groth16 is only needed to mask SRS elements corresponding to the statement
- We use a different verification algorithm, getting rid of "statement" trapdoor
  - Verifier becomes faster
  - In Polymath, V interpolates a polynomial in $\mathbb{F}_p$ of degree |x|
    - Instead of doing |x|-long MSM in Groth16

Scenic Route to Polymath

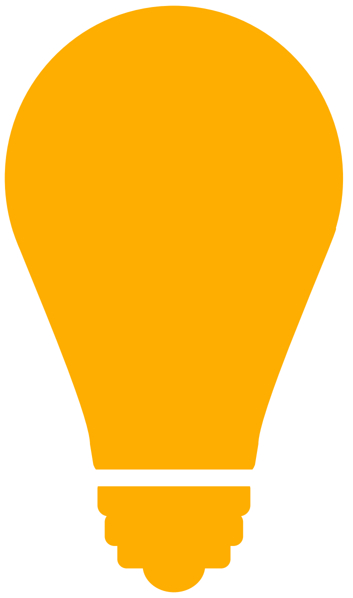
$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

- We only have three trapdoors

Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

- We only have three trapdoors
- It is easier to choose “good” exponents!



Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

- We only have three trapdoors
- It is easier to choose “good” exponents!

We replace two trapdoors with y^α and y^γ, where $y = x^\sigma$

Scenic Route to Polymath

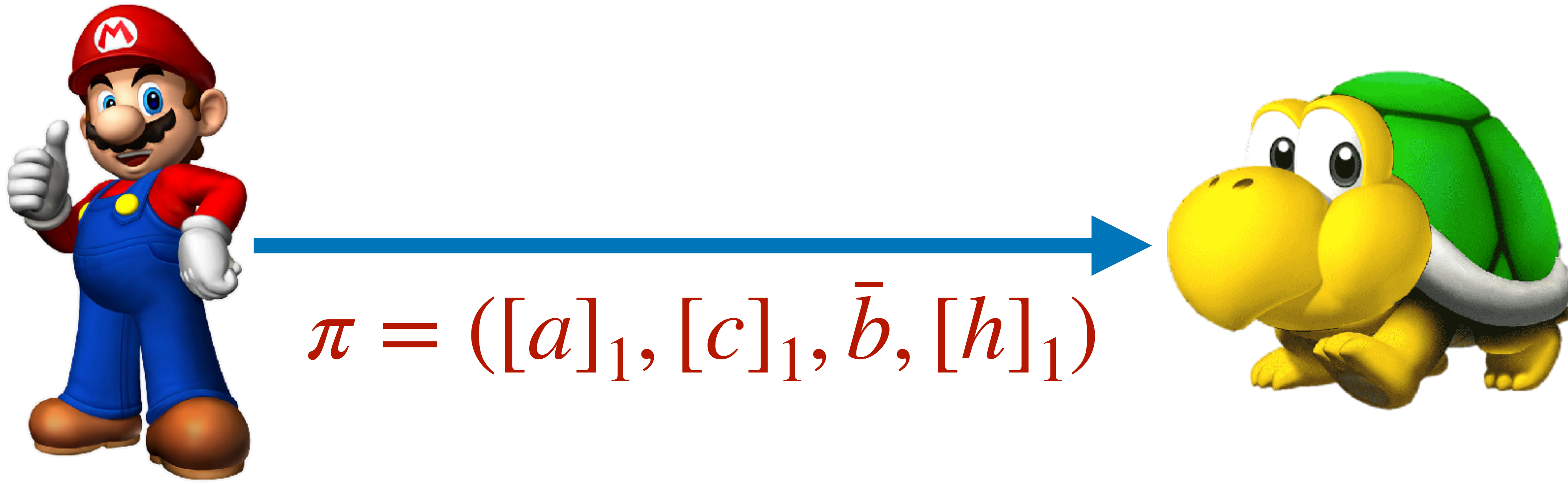
$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

- We only have three trapdoors
- It is easier to choose "good" exponents!

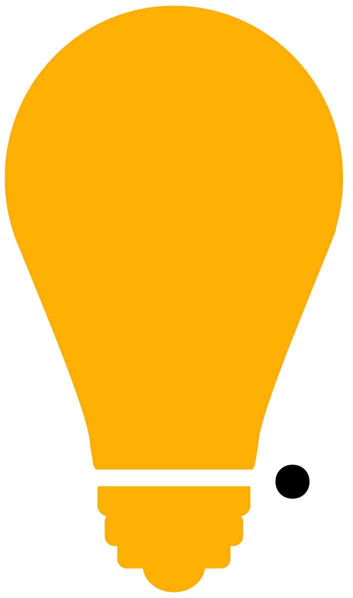
We replace two trapdoors with y^α and y^γ, where $y = x^\sigma$

- x is real trapdoor, α, γ, σ are field elements

Scenic Route to Polymath



- We only have three trapdoors
- It is easier to choose “good” exponents!



We replace two trapdoors with y^{α} and y^{γ}, where $y = x^{\sigma}$

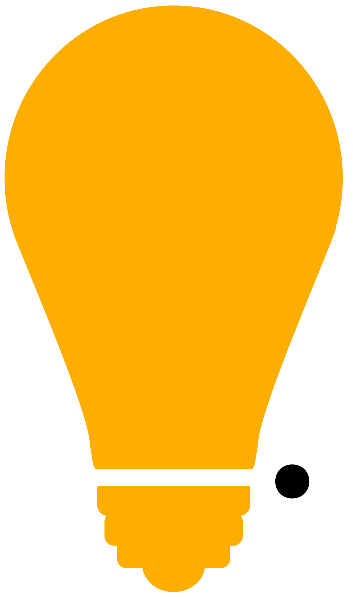
- x is real trapdoor, α, γ, σ are field elements

- Exhaustive search to find "smallest" exponents that result in knowledge-soundness

Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

- We only have three trapdoors
- It is easier to choose “good” exponents!

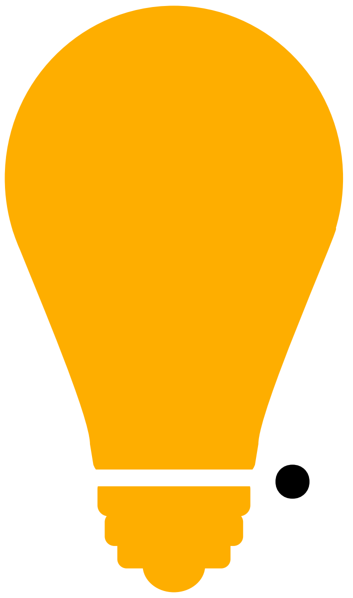


- We replace two trapdoors with y^{α} and y^{γ}, where $y = x^{\sigma}$
  - x is real trapdoor, α, γ, σ are field elements
- Exhaustive search to find "smallest" exponents that result in knowledge-soundness
- Kamek saith: $\alpha = 3,\ \gamma = -5,\ \sigma = n + 3$

Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

- We only have three trapdoors
- It is easier to choose “good” exponents!



We replace two trapdoors with y^α and y^γ, where $y = x^\sigma$

- x is real trapdoor, α, γ, σ are field elements

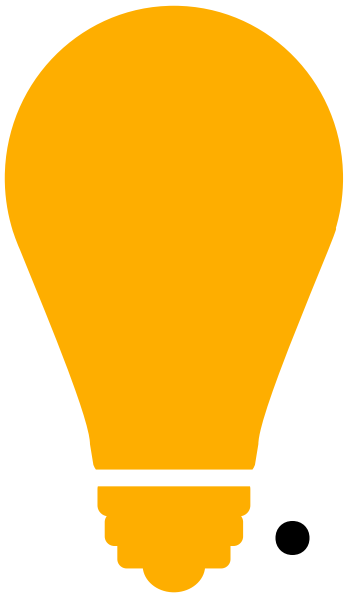
- Exhaustive search to find "smallest" exponents that result in knowledge-soundness
- Kamek saith: $\alpha = 3,\ \gamma = -5,\ \sigma = n + 3$

Problem:

- SRS is circuit-dependent
- It does not contain enough elements to compute $[h]_1$

Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$



We add another trapdoor $[z]_1$ that is **only** used to compute KZG opening

- This adds elements to the SRS

Scenic Route to Polymath

$\pi = ([a]_1, [c]_1, \bar{b}, [h]_1)$

We add another trapdoor $[z]_1$ that is **only** used to compute KZG opening

- This adds elements to the SRS

Profit

On Security Proofs

- Completeness and zero-knowledge proofs are short and standard

On Security Proofs

- Completeness and zero-knowledge proofs are short and standard
- Soundness proof is seven pages

On Security Proofs

- Completeness and zero-knowledge proofs are short and standard
- Soundness proof is seven pages

1. Groth16 itself has a complicated proof

On Security Proofs

- Completeness and zero-knowledge proofs are short and standard
- Soundness proof is seven pages

1. Groth16 itself has a complicated proof
2. Using virtual trapdoors x^i instead of independent trapdoors makes proof more complicated

On Security Proofs

- Completeness and zero-knowledge proofs are short and standard
- Soundness proof is seven pages

1. Groth16 itself has a complicated proof
2. Using virtual trapdoors x^i instead of independent trapdoors makes proof more complicated
3. To get tight security after Fiat-Shamir, we prove special-soundness

On Security Proofs

- Completeness and zero-knowledge proofs are short and standard
- Soundness proof is seven pages

1. Groth16 itself has a complicated proof
2. Using virtual trapdoors x^i instead of independent trapdoors makes proof more complicated
3. To get tight security after Fiat-Shamir, we prove special-soundness
4. Proof is in **AGM** with **Oblivious Sampling** [Lipmaa-Parisella-Siim, TCC 2023]

On Security Proofs

- Completeness and zero-knowledge proofs are short and standard
- Soundness proof is seven pages

1. Groth16 itself has a complicated proof
2. Using virtual trapdoors x^i instead of independent trapdoors makes proof more complicated
3. To get tight security after Fiat-Shamir, we prove special-soundness
4. Proof is in **AGM** with **Oblivious Sampling** [Lipmaa-Parisella-Siim, TCC 2023]
    - LPS23 noted that KZG is often used wrongly

On Security Proofs

- Completeness and zero-knowledge proofs are short and standard
- Soundness proof is seven pages

1. Groth16 itself has a complicated proof
2. Using virtual trapdoors x^i instead of independent trapdoors makes proof more complicated
3. To get tight security after Fiat-Shamir, we prove special-soundness
4. Proof is in **AGM** with **Oblivious Sampling** [Lipmaa-Parisella-Siim, TCC 2023]
    - LPS23 noted that KZG is often used wrongly
    - Constructions are secure in AGM but not when adversary can do o.s.

On Security Proofs

- Completeness and zero-knowledge proofs are short and standard
- Soundness proof is seven pages

1. Groth16 itself has a complicated proof
2. Using virtual trapdoors x^i instead of independent trapdoors makes proof more complicated
3. To get tight security after Fiat-Shamir, we prove special-soundness
4. Proof is in **AGM** with **Oblivious Sampling** [Lipmaa-Parisella-Siim, TCC 2023]
    - LPS23 noted that KZG is often used wrongly
    - Constructions are secure in AGM but not when adversary can do o.s.
    - And o.s. is for free!

On Security Proofs

- Completeness and zero-knowledge proofs are short and standard
- Soundness proof is seven pages

1. Groth16 itself has a complicated proof
2. Using virtual trapdoors x^i instead of independent trapdoors makes proof more complicated
3. To get tight security after Fiat-Shamir, we prove special-soundness
4. Proof is in **AGM** with **Oblivious Sampling** [Lipmaa-Parisella-Siim, TCC 2023]
    - LPS23 noted that KZG is often used wrongly
    - Constructions are secure in AGM but not when adversary can do o.s.
    - And o.s. is for free!

Part of Polymath's proof is machine-checked

Conclusion & Open Problems

- First improvement on Groth16 since 2016

Conclusion & Open Problems

- First improvement on Groth16 since 2016
  - Hopefully** encourages other researchers to improve Groth16 even further

Conclusion & Open Problems

- First improvement on Groth16 since 2016
  - Hopefully** encourages other researchers to improve Groth16 even further
- **Open problems:**

Conclusion & Open Problems

- First improvement on Groth16 since 2016
  - Hopefully** encourages other researchers to improve Groth16 even further
- **Open problems:**
  - Big bottleneck: prover's speed

Conclusion & Open Problems

- First improvement on Groth16 since 2016
  - Hopefully** encourages other researchers to improve Groth16 even further
- **Open problems:**
  - Big bottleneck: prover's speed
  - What is the lower bound in the RO for argument size?

Conclusion & Open Problems

- First improvement on Groth16 since 2016
  - Hopefully** encourages other researchers to improve Groth16 even further
- **Open problems:**
  - Big bottleneck: prover's speed
  - What is the lower bound in the RO for argument size?
  - Can we improve on verifier's speed (in any setting)?**

Conclusion & Open Problems

- First improvement on Groth16 since 2016
  - Hopefully** encourages other researchers to improve Groth16 even further
- **Open problems:**
  - Big bottleneck: prover’s speed
  - What is the lower bound in the RO for argument size?
  - Can we improve on verifier’s speed (in any setting)?**
    - Not necessarily pairing-based…