# Garuda and Pari:

## Faster and Smaller SNARKs via Equifficient Polynomial Commitments

**Michel Dellepere**

Ava Labs

**Alireza Shirzad**

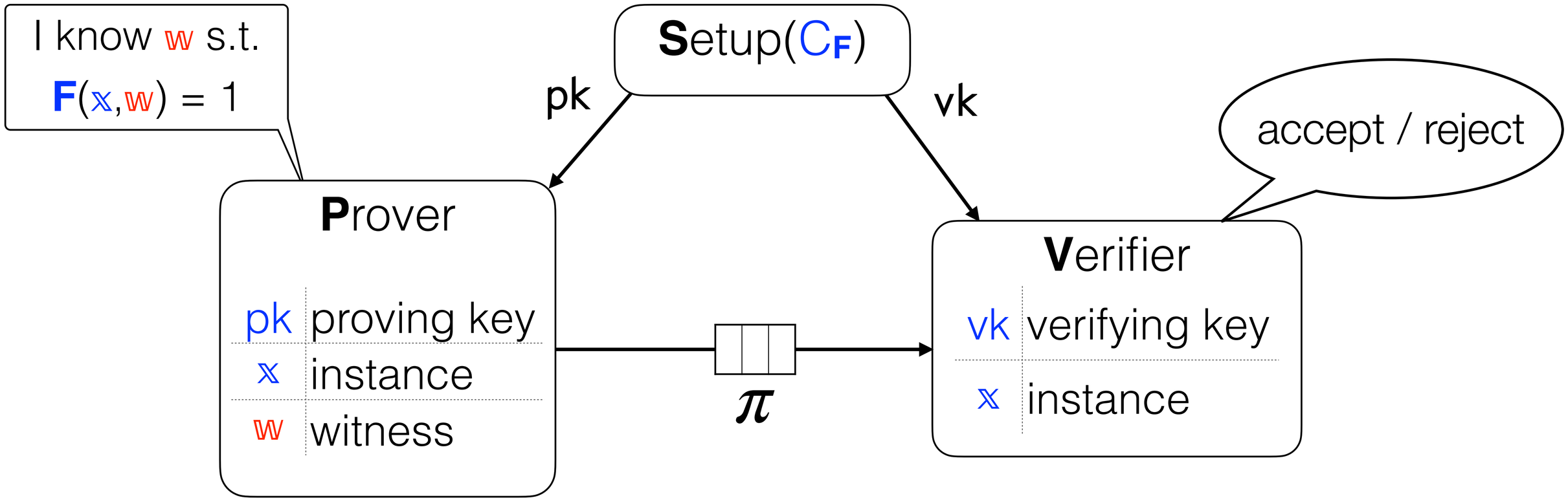
University of Pennsylvania

**Pratyush Mishra**

University of Pennsylvania

# Succinct Non-Interactive Argument of Knowledge (SNARK)

# Succinct Non-Interactive Argument of Knowledge (SNARK)



# Q1: How small can the proof $\pi$ be?

# Q1: How small can the proof $\pi$ be?

For blockchains, smaller is better!

# Q1: How small can the proof $\pi$ be?

For blockchains, smaller is better!



$|\pi|$
(on BLS12-381)

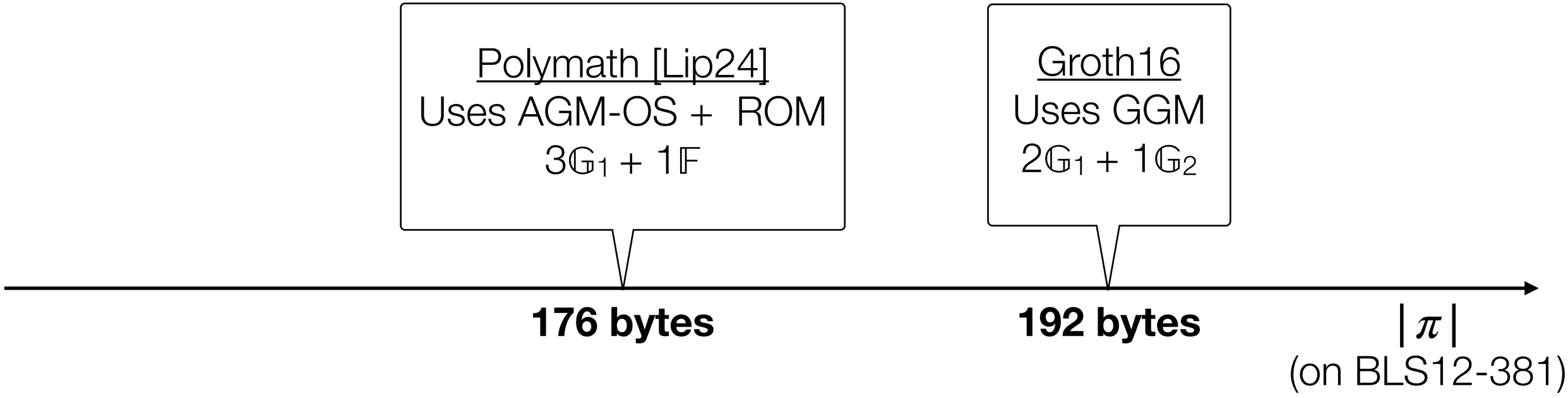
# Q1: How small can the proof $\pi$ be?

For blockchains, smaller is better!



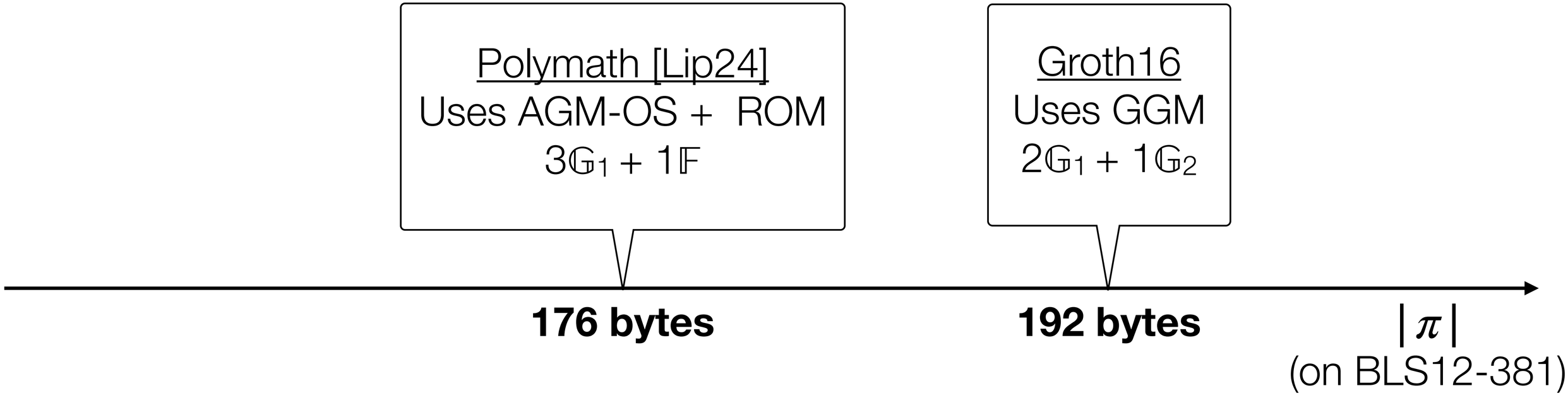
# Q1: How small can the proof $\pi$ be?

For blockchains, smaller is better!



# Q1: How small can the proof $\pi$ be?

For blockchains, smaller is better!




Can we go lower than 176 bytes?

# Q2: How fast can we prove?

# Q2: How fast can we prove?

Proving has a large overhead (~1000x) over native computation

How to reduce this cost?

# Q2: How fast can we prove?

Proving has a large overhead (~1000x) over native computation

How to reduce this cost?

# Q2: How fast can we prove?

Proving has a large overhead (~1000x) over native computation

How to reduce this cost?

**Approach 1:** Free addition gates

# Q2: How fast can we prove?

Proving has a large overhead (~1000x) over native computation

How to reduce this cost?

**Approach 1:** Free addition gates

- Only pay cryptographic (e.g., MSM) costs for multiplication gates

- Achieved by circuit-specific SNARKs [GGPR13, BCTV14, Groth16]

# Q2: How fast can we prove?

Proving has a large overhead (~1000x) over native computation

How to reduce this cost?

**Approach 1:** Free addition gates

- Only pay cryptographic (e.g., MSM) costs for multiplication gates
- Achieved by circuit-specific SNARKs [GGPR13, BCTV14, Groth16]

**Approach 2:** Custom gates

# Q2: How fast can we prove?

Proving has a large overhead (~1000x) over native computation

How to reduce this cost?

**Approach 1:** Free addition gates

- Only pay cryptographic (e.g., MSM) costs for multiplication gates
- Achieved by circuit-specific SNARKs [GGPR13, BCTV14, Groth16]

**Approach 2:** Custom gates

- Specialized gates for particular computations (e.g., EC addition, Poseidon S-box)
- Proposed recently for TurboPlonk [GW19], used widely [RISC0,Plonky3,CBBZ23,STW23]

# Q2: How fast can we prove?

Proving has a large overhead (~1000x) over native computation

How to reduce this cost?

**Approach 1:** Free addition gates

- Only pay cryptographic (e.g., MSM) costs for multiplication gates
- Achieved by circuit-specific SNARKs [GGPR13, BCTV14, Groth16]

**Approach 2:** Custom gates

- Specialized gates for particular computations (e.g., EC addition, Poseidon S-box)
- Proposed recently for TurboPlonk [GW19], used widely [RISC0,Plonky3,CBBZ23,STW23]

Unfortunately, no existing SNARK supports both!

Can we fix this?

# Our Contributions

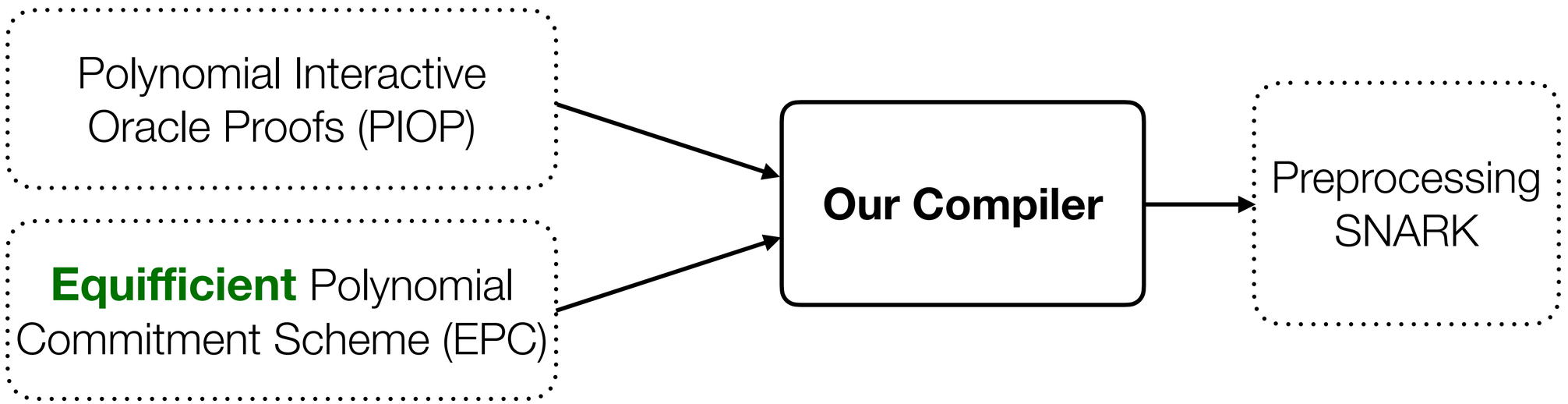
# New Methodology

# New Methodology

We extend existing SNARK methodologies [CHMMVW20, BFS20]
to construct our SNARKs

# New Methodology

We extend existing SNARK methodologies [CHMMVW20, BFS20]
to construct our SNARKs



# Pari: a SNARK with a tiny proof

# Pari: a SNARK with a tiny proof

Recipe: Univariate ZeroCheck PIOP + KZG-Based EPC

# Pari: a SNARK with a tiny proof

Recipe: Univariate ZeroCheck PIOP + KZG-Based EPC

- Proof size: $2\mathbb{G}_1 + 2\mathbb{F}$ = **160 Bytes** in BLS12-381

# Pari: a SNARK with a tiny proof

Recipe: Univariate ZeroCheck PIOP + KZG-Based EPC

- Proof size: $2\mathbb{G}_1 + 2\mathbb{F}$ = **160 Bytes** in BLS12-381
  - < Groth16 with a 192-Byte proof

# Pari: a SNARK with a tiny proof

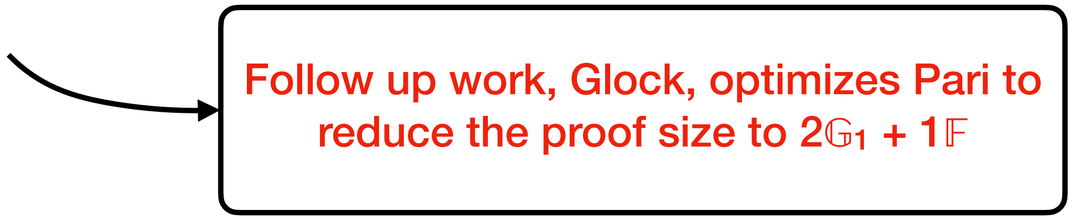
Recipe: Univariate ZeroCheck PIOP + KZG-Based EPC

- Proof size: $2\mathbb{G}_1 + 2\mathbb{F}$ = **160 Bytes** in BLS12-381
  - < Groth16 with a 192-Byte proof
  - < Polymath with a 176-Byte proof

# Pari: a SNARK with a tiny proof

Recipe: Univariate ZeroCheck PIOP + KZG-Based EPC

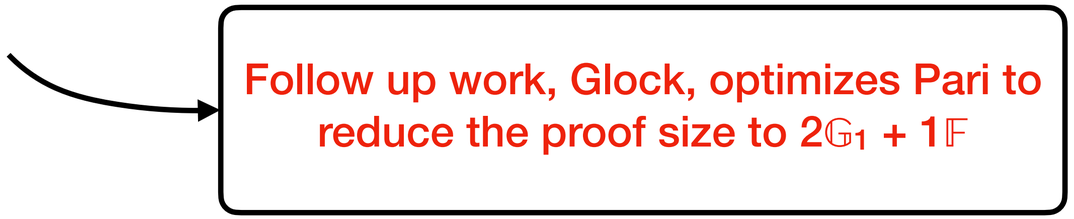
- Proof size: $2\mathbb{G}_1 + 2\mathbb{F}$ = **160 Bytes** in BLS12-381
  - < Groth16 with a 192-Byte proof
  - < Polymath with a 176-Byte proof



# Pari: a SNARK with a tiny proof

Recipe: Univariate ZeroCheck PIOP + KZG-Based EPC

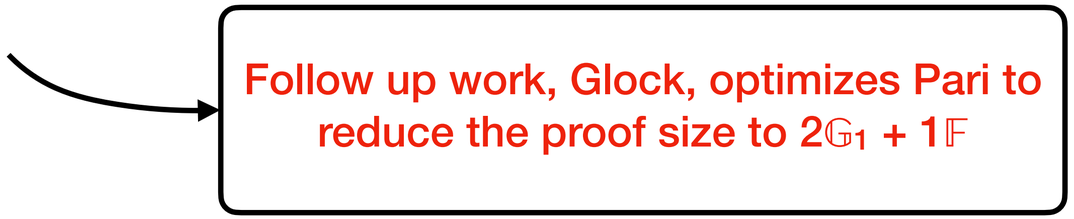
- Proof size: $2\mathbb{G}_1 + 2\mathbb{F}$ = **160 Bytes** in BLS12-381
  - < Groth16 with a 192-Byte proof
  - < Polymath with a 176-Byte proof
- Verifier time **< 1ms**



# Pari: a SNARK with a tiny proof

Recipe: Univariate ZeroCheck PIOP + KZG-Based EPC

- Proof size: $2\mathbb{G}_1 + 2\mathbb{F}$ = **160 Bytes** in BLS12-381
  - < Groth16 with a 192-Byte proof
  - < Polymath with a 176-Byte proof
- Verifier time **< 1ms**
  - Faster or on-par with prior works



# Pari: a SNARK with a tiny proof

Recipe: Univariate ZeroCheck PIOP + KZG-Based EPC

- Proof size: $2\mathbb{G}_1 + 2\mathbb{F}$ = **160 Bytes** in BLS12-381
  - < Groth16 with a 192-Byte proof
  - < Polymath with a 176-Byte proof

Follow up work, Glock, optimizes Pari to reduce the proof size to $2\mathbb{G}_1 + 1\mathbb{F}$

- Verifier time **< 1ms**
  - Faster or on-par with prior works

Cheapest on-chain verifier

# Pari: a SNARK with a tiny proof

Recipe: Univariate ZeroCheck PIOP + KZG-Based EPC

- Proof size: $2\mathbb{G}_1 + 2\mathbb{F}$ = **160 Bytes** in BLS12-381
  - < Groth16 with a 192-Byte proof
  - < Polymath with a 176-Byte proof

Follow up work, Glock, optimizes Pari to reduce the proof size to $2\mathbb{G}_1 + 1\mathbb{F}$

- Verifier time **< 1ms**
  - Faster or on-par with prior works

Cheapest on-chain verifier

- Prover time: **Quasi-Linear**, like prior works

# Pari: a SNARK with a tiny proof

Recipe: Univariate ZeroCheck PIOP + KZG-Based EPC

- Proof size: $2\mathbb{G}_1 + 2\mathbb{F}$ = **160 Bytes** in BLS12-381
  - < Groth16 with a 192-Byte proof
  - < Polymath with a 176-Byte proof

Follow up work, Glock, optimizes Pari to reduce the proof size to $2\mathbb{G}_1 + 1\mathbb{F}$

- Verifier time **< 1ms**
  - Faster or on-par with prior works

Cheapest on-chain verifier

- Prover time: **Quasi-Linear**, like prior works
  - 2x slower than Groth16

# Pari: a SNARK with a tiny proof

Recipe: Univariate ZeroCheck PIOP + KZG-Based EPC

- Proof size: $2\mathbb{G}_1 + 2\mathbb{F}$ = **160 Bytes** in BLS12-381
  - < Groth16 with a 192-Byte proof
  - < Polymath with a 176-Byte proof

Follow up work, Glock, optimizes Pari to reduce the proof size to $2\mathbb{G}_1 + 1\mathbb{F}$

- Verifier time **< 1ms**
  - Faster or on-par with prior works

Cheapest on-chain verifier

- Prover time: **Quasi-Linear**, like prior works
  - 2x slower than Groth16
  - 2x faster than Polymath

# Garuda: a SNARK with fast prover

# Garuda: a SNARK with fast prover

Recipe: Multivariate ZeroCheck PIOP + PST-Based EPC

# Garuda: a SNARK with fast prover

Recipe: Multivariate ZeroCheck PIOP + PST-Based EPC

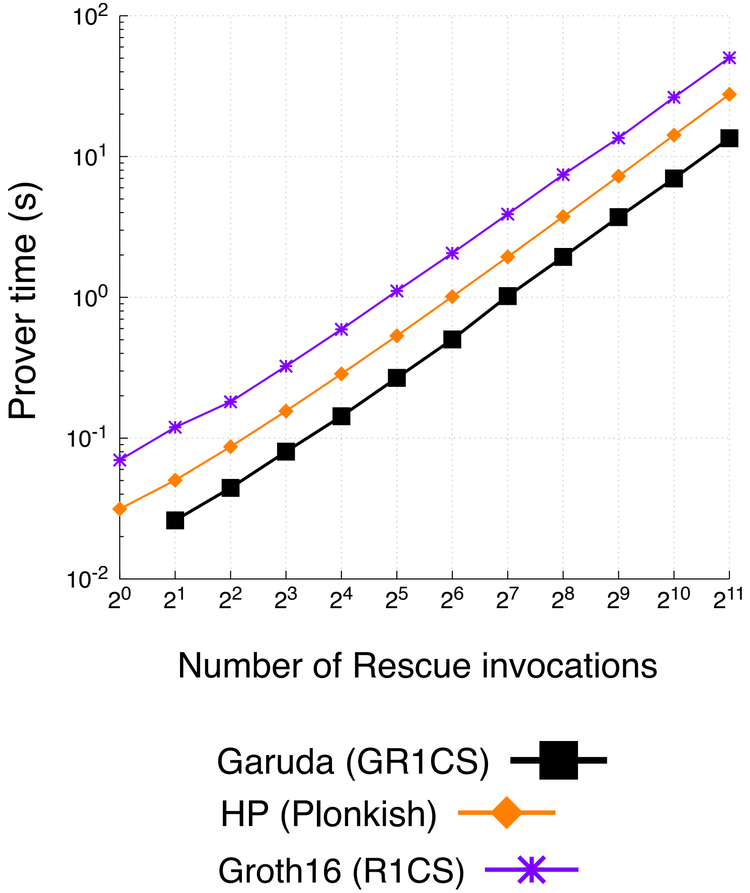
Supports ✅ free addition gates ✅ custom gates

# Garuda: a SNARK with fast prover

Recipe: Multivariate ZeroCheck PIOP + PST-Based EPC

Supports ✅ free addition gates ✅ custom gates

- Prover Time: Linear, asymptotically optimal

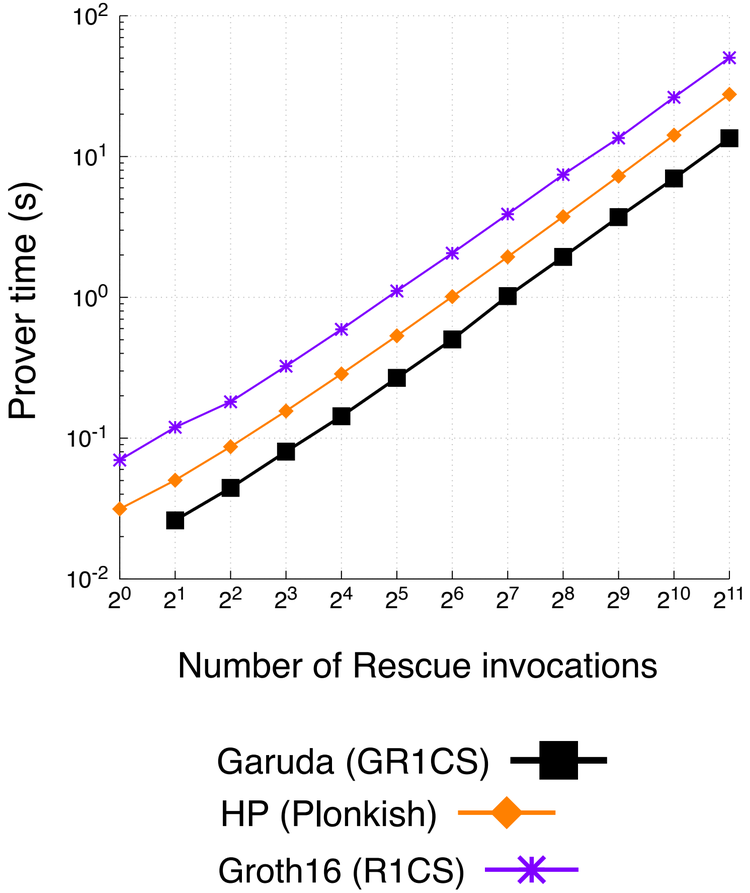


# Garuda: a SNARK with fast prover

Recipe: Multivariate ZeroCheck PIOP + PST-Based EPC

Supports ✅ free addition gates ✅ custom gates

- Prover Time: Linear, asymptotically optimal
  - >2x faster than Groth16, Hyperplonk

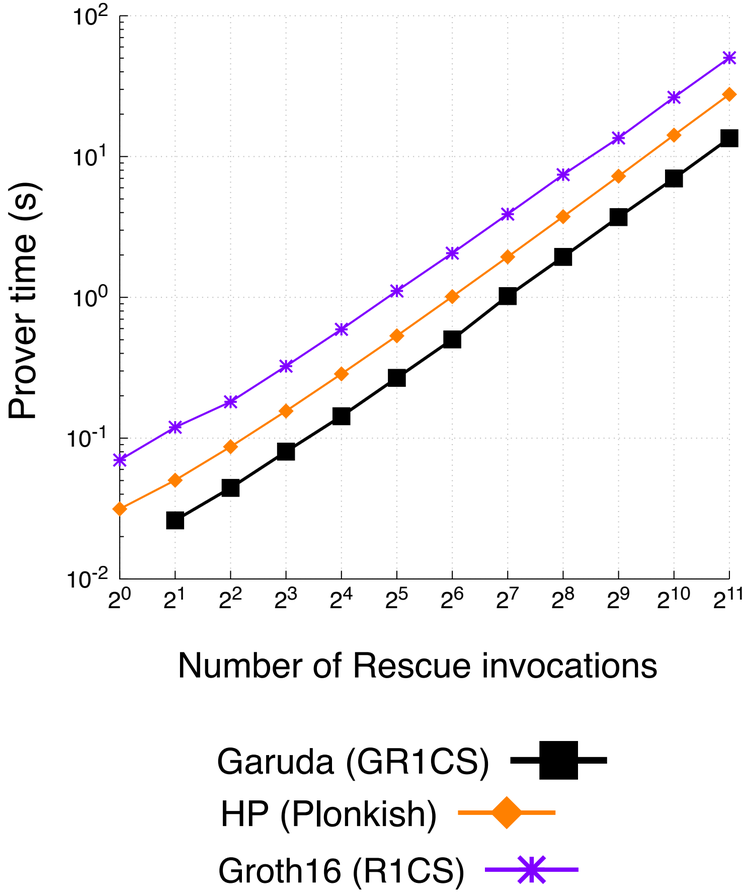


# Garuda: a SNARK with fast prover

Recipe: Multivariate ZeroCheck PIOP + PST-Based EPC

Supports ✅ free addition gates ✅ custom gates

- Prover Time: Linear, asymptotically optimal
  - >2x faster than Groth16, Hyperplonk
- Verifier time = **2~3ms**

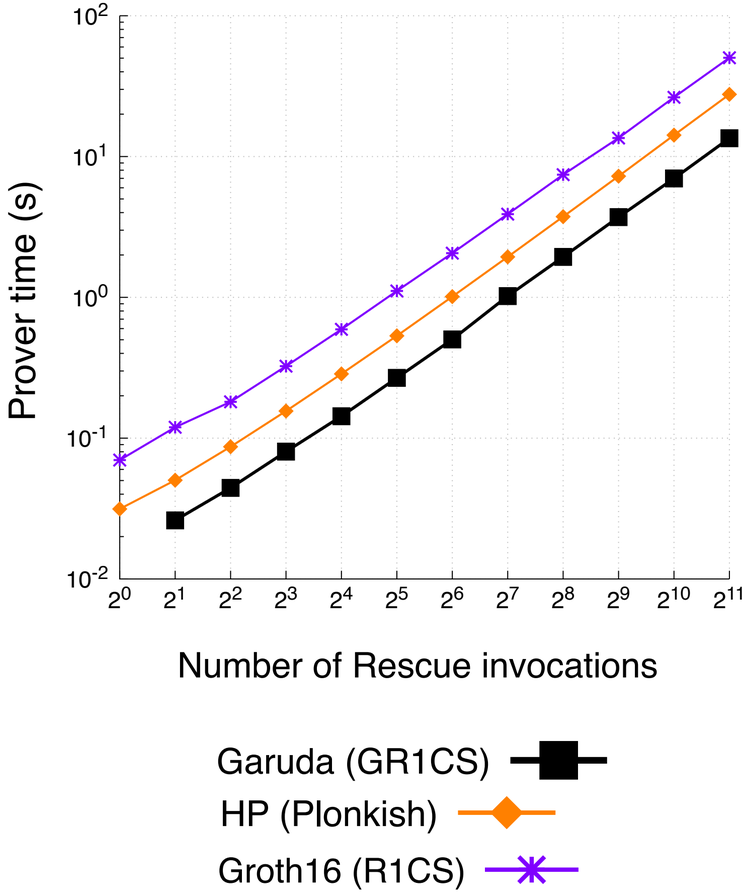


# Garuda: a SNARK with fast prover

Recipe: Multivariate ZeroCheck PIOP + PST-Based EPC

Supports ✅ free addition gates ✅ custom gates

- Prover Time: Linear, asymptotically optimal
  - >2x faster than Groth16, Hyperplonk
- Verifier time = **2~3ms**
  - slower than Groth16

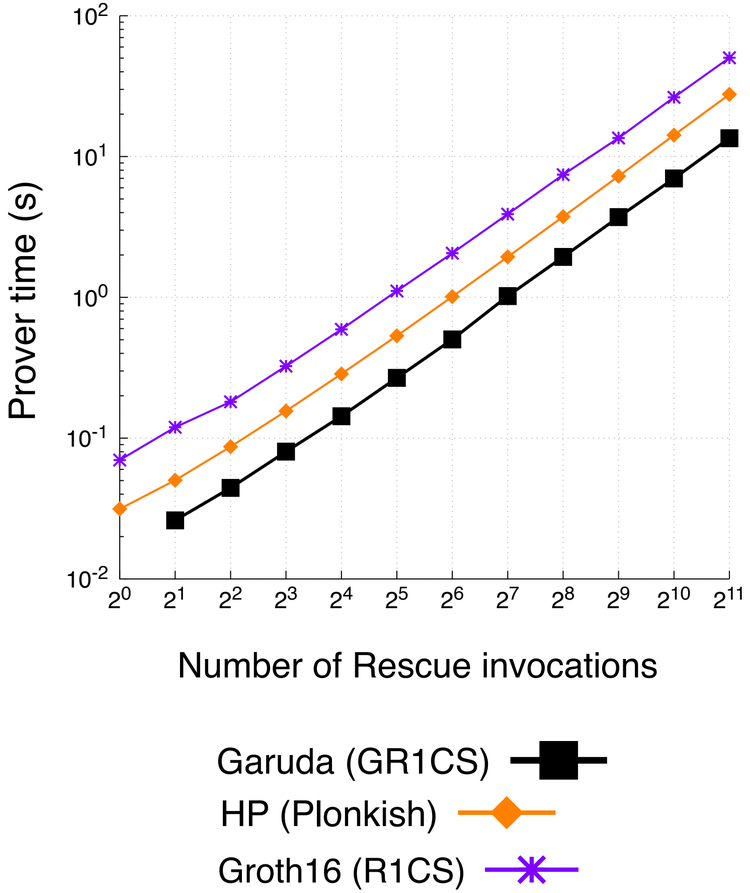


# Garuda: a SNARK with fast prover

Recipe: Multivariate ZeroCheck PIOP + PST-Based EPC

Supports ✅ free addition gates ✅ custom gates

- Prover Time: Linear, asymptotically optimal
  - >2x faster than Groth16, Hyperplonk
- Verifier time = **2~3ms**
  - slower than Groth16
  - 3x faster than Hyperplonk

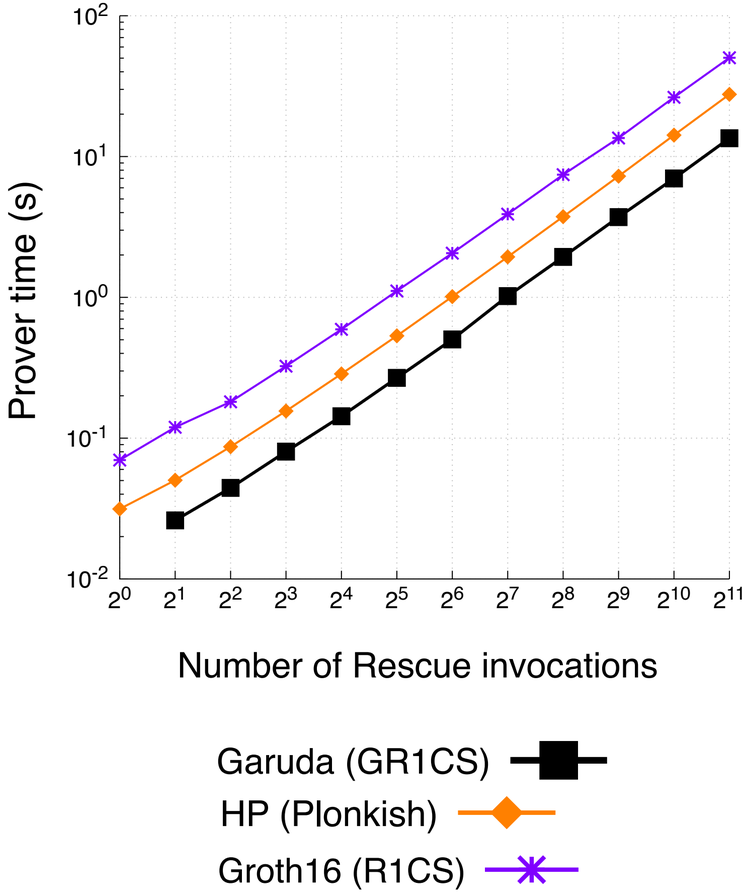


# Garuda: a SNARK with fast prover

Recipe: Multivariate ZeroCheck PIOP + PST-Based EPC

Supports ✅ free addition gates ✅ custom gates

- Prover Time: Linear, asymptotically optimal
  - >2x faster than Groth16, Hyperplonk
- Verifier time = **2~3ms**
  - slower than Groth16
  - 3x faster than Hyperplonk
- Proof size: **5~10 KB**

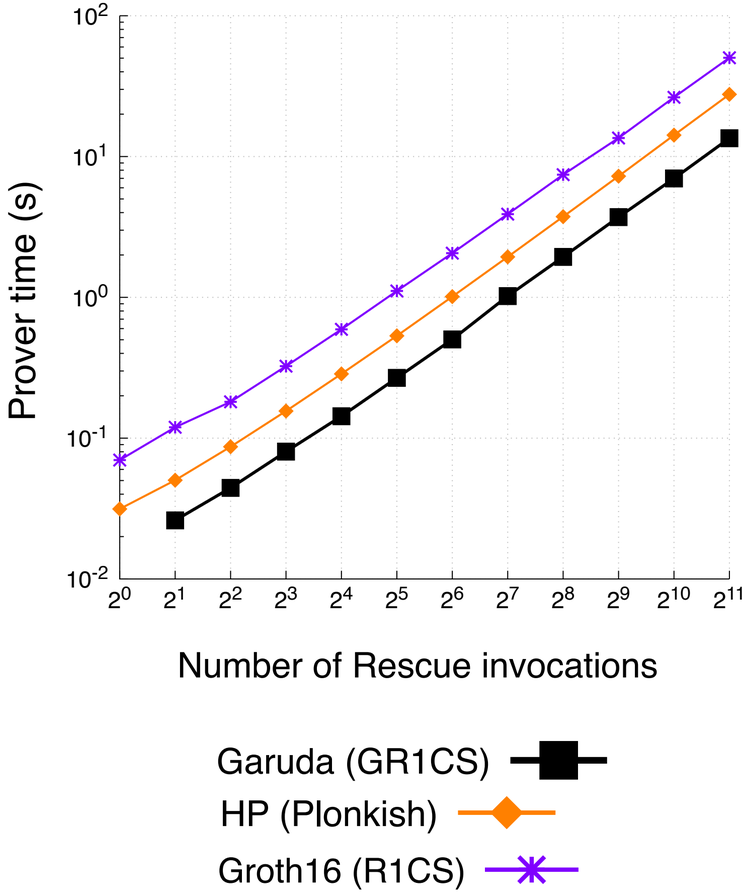


# Garuda: a SNARK with fast prover

Recipe: Multivariate ZeroCheck PIOP + PST-Based EPC

Supports ✅ free addition gates ✅ custom gates

- Prover Time: Linear, asymptotically optimal
  - >2x faster than Groth16, Hyperplonk
- Verifier time = **2~3ms**
  - slower than Groth16
  - 3x faster than Hyperplonk
- Proof size: **5~10 KB**
  - Larger than Groth16

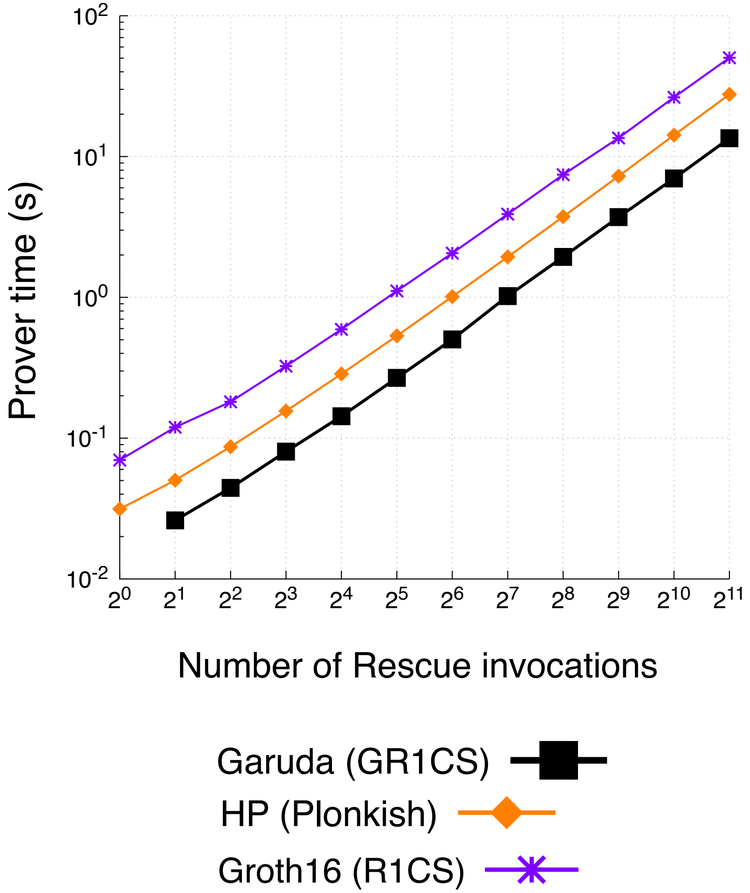


# Garuda: a SNARK with fast prover

Recipe: Multivariate ZeroCheck PIOP + PST-Based EPC

Supports ✅ free addition gates ✅ custom gates

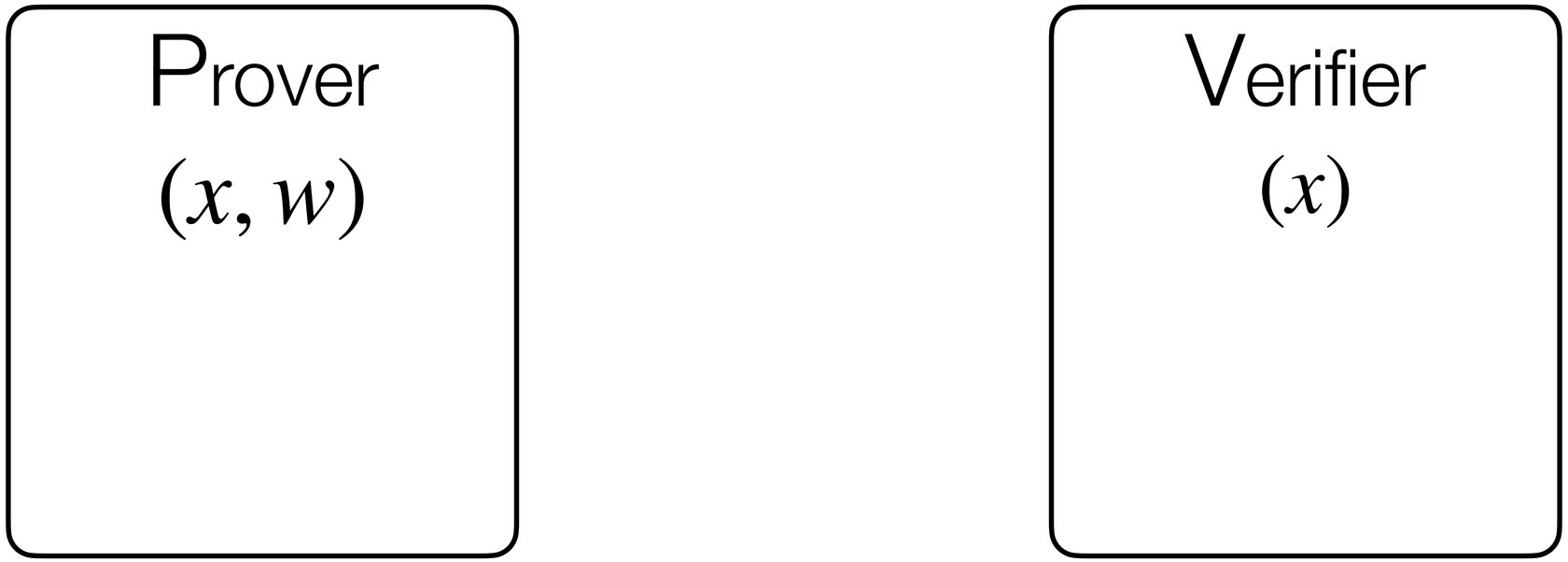
- Prover Time: Linear, asymptotically optimal
  - >2x faster than Groth16, Hyperplonk
- Verifier time = **2~3ms**
  - slower than Groth16
  - 3x faster than Hyperplonk
- Proof size: **5~10 KB**
  - Larger than Groth16
  - >2x smaller than Hyperplonk



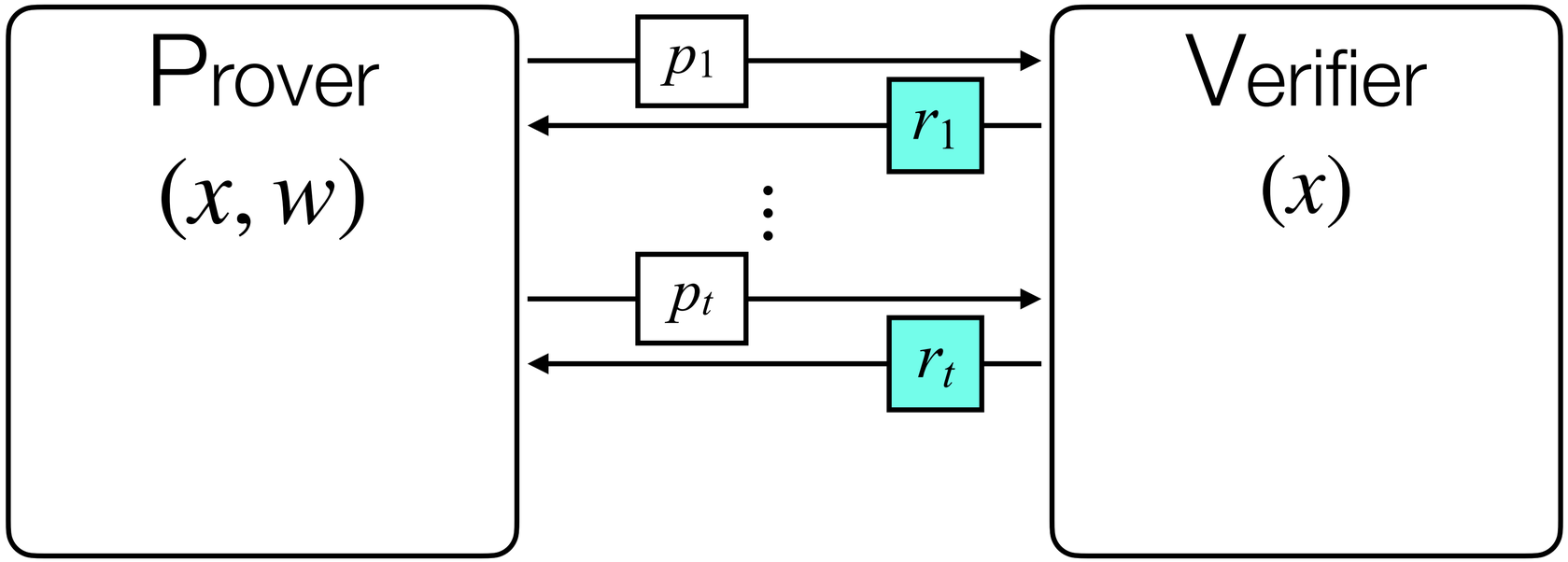
# Background

# Background: Polynomial IOPs

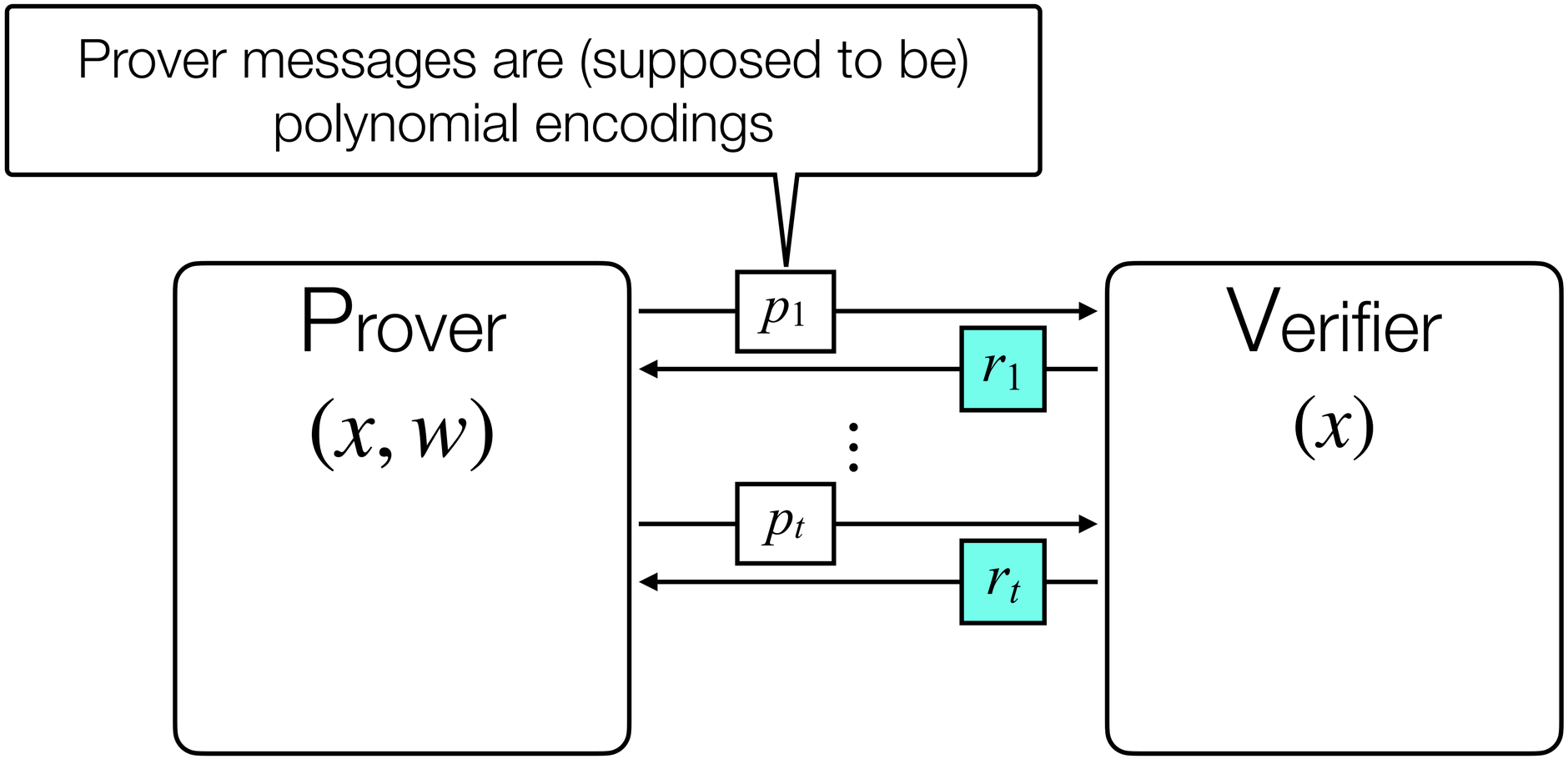
# Background: Polynomial IOPs



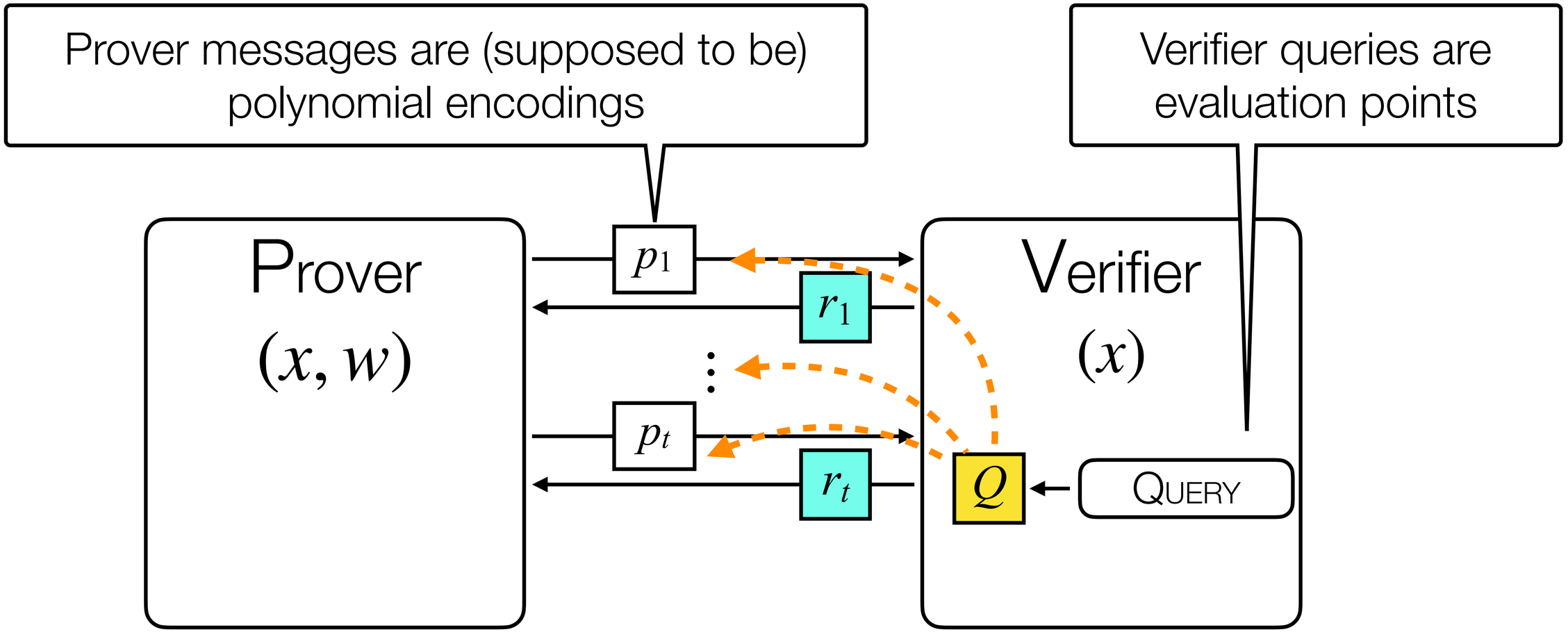
# Background: Polynomial IOPs



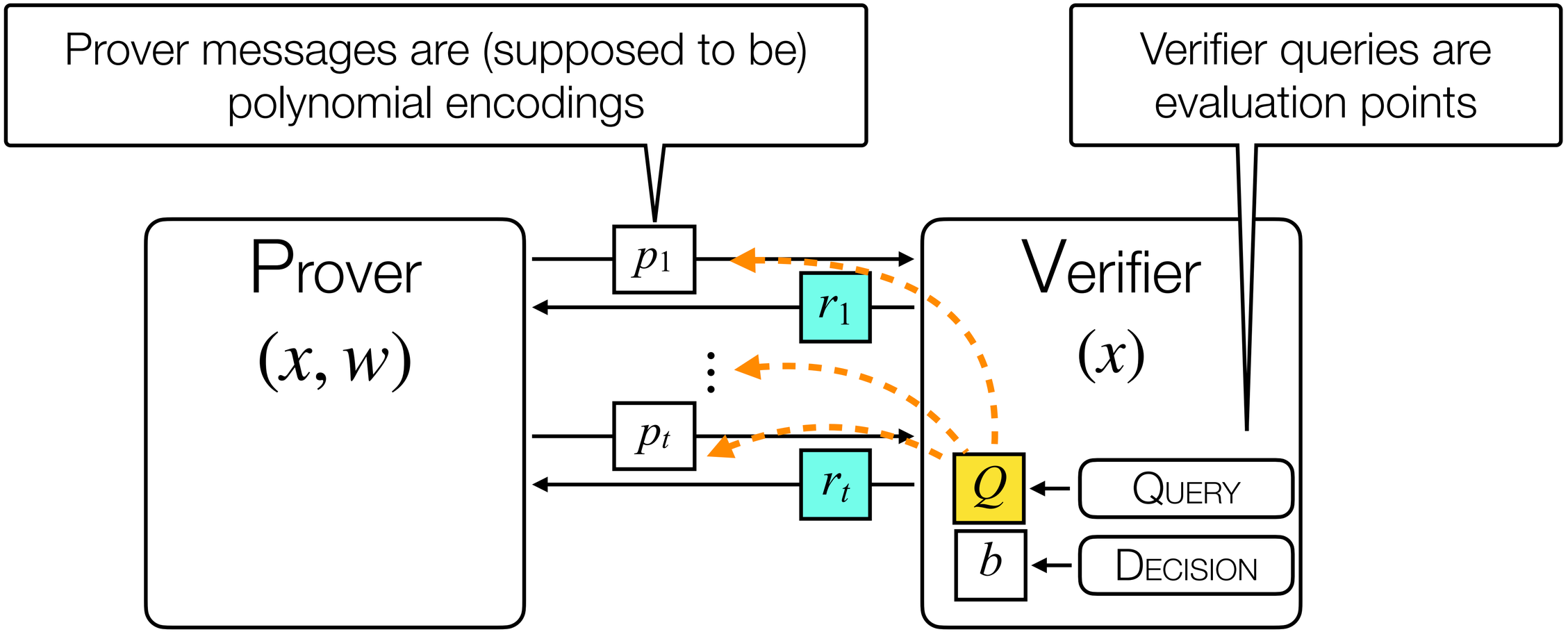
# Background: Polynomial IOPs



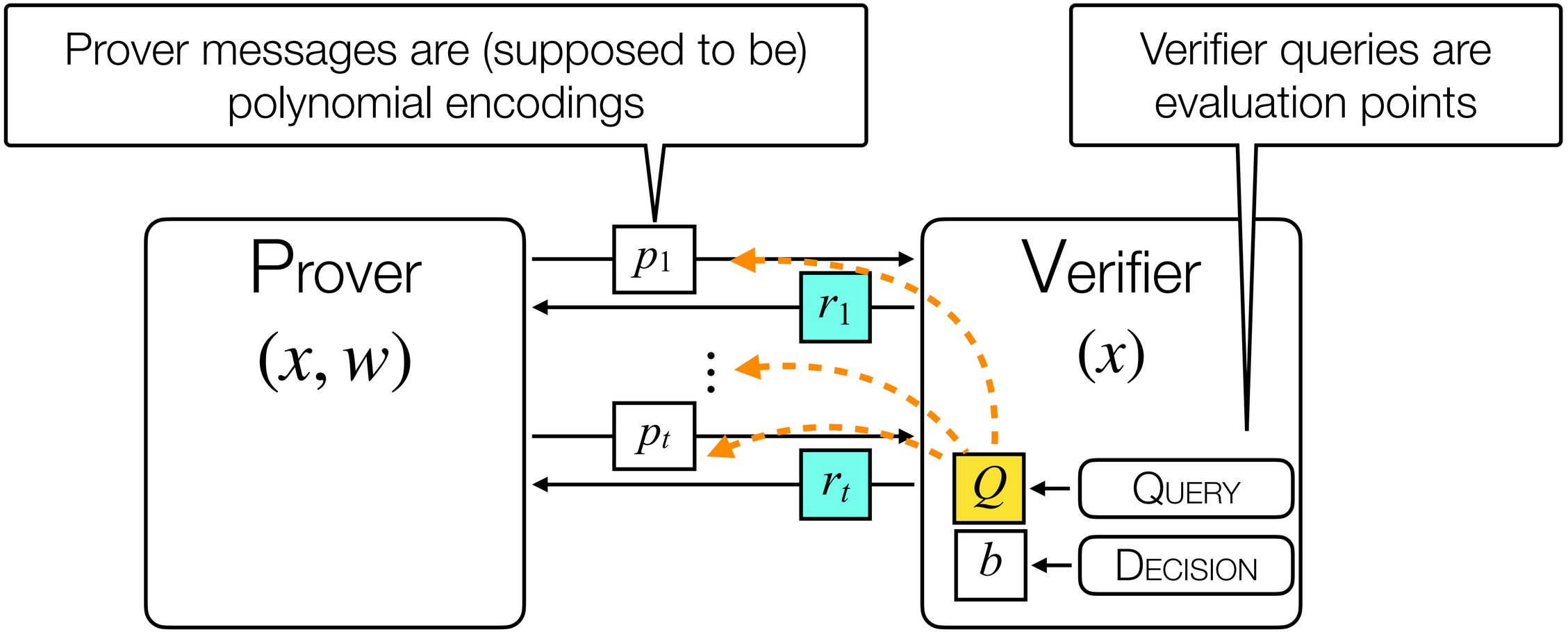
# Background: Polynomial IOPs



# Background: Polynomial IOPs

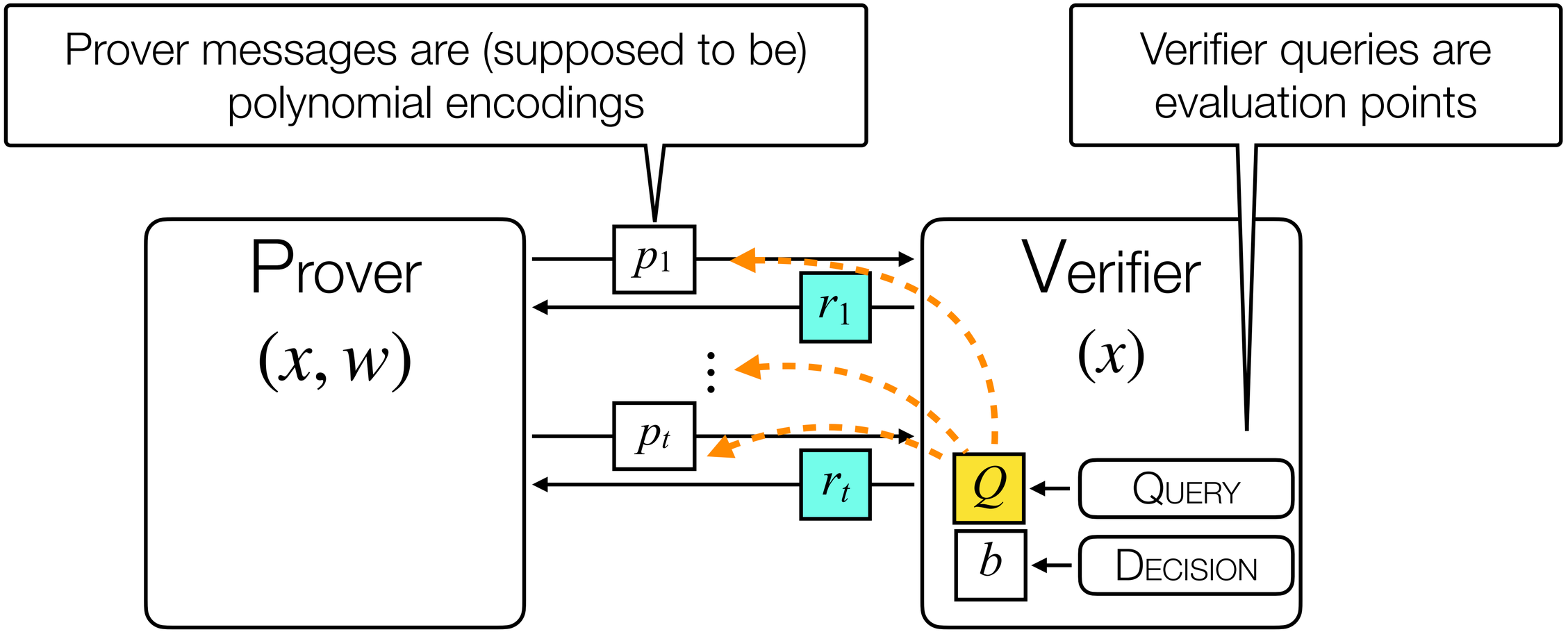


# Background: Polynomial IOPs




- **Completeness:** If $F(x, w) = 1$, then $\mathcal{V}$ accepts

# Background: Polynomial IOPs



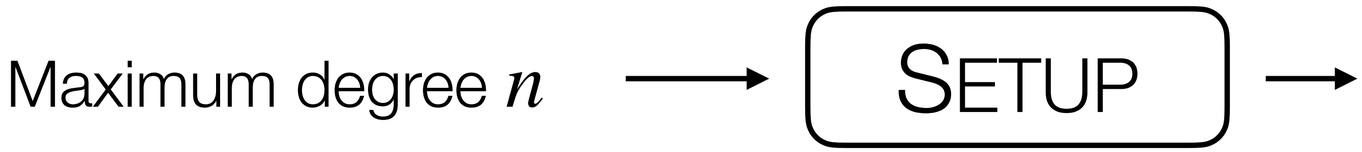

- **Completeness:** If $F(x, w) = 1$, then $\mathcal{V}$ accepts
- **Soundness:** If $F(x, w) \neq 1$, then $\mathcal{V}$ rejects

# Background: PC schemes

Commit to a polynomial, and later on prove its correct evaluation

# Background: PC schemes

Commit to a polynomial, and later on prove its correct evaluation



# Background: PC schemes

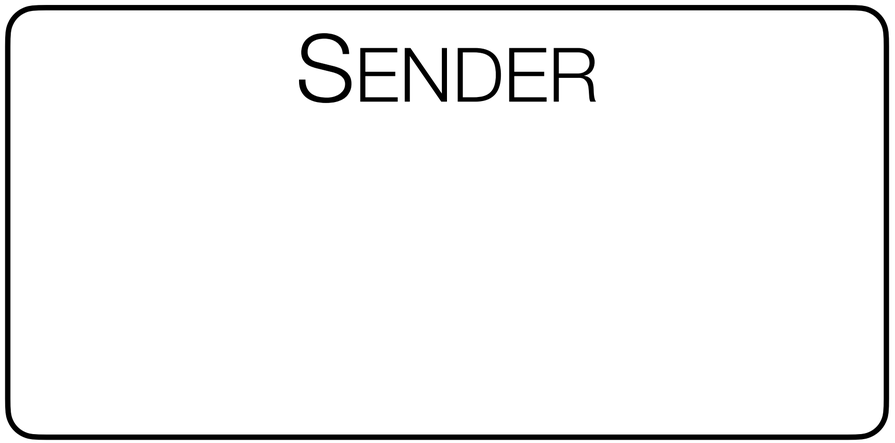
Commit to a polynomial, and later on prove its correct evaluation



# Background: PC schemes

Commit to a polynomial, and later on prove its correct evaluation

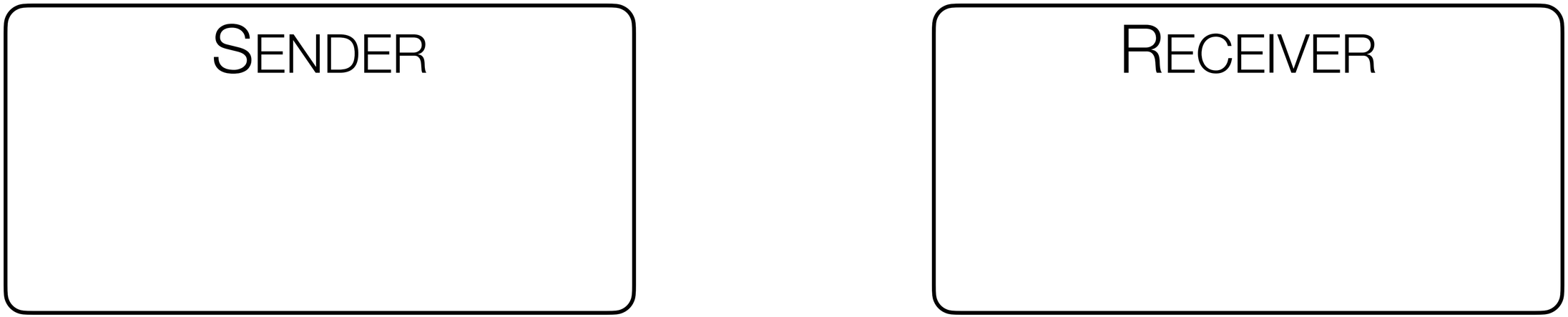


# Background: PC schemes

Commit to a polynomial, and later on prove its correct evaluation

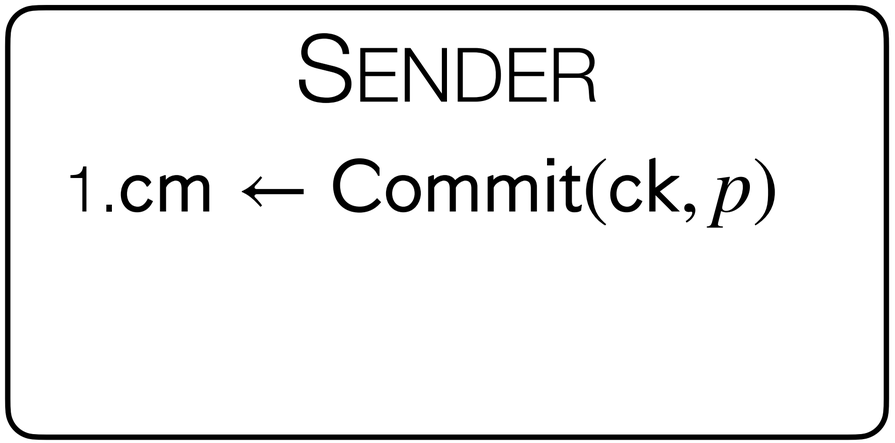


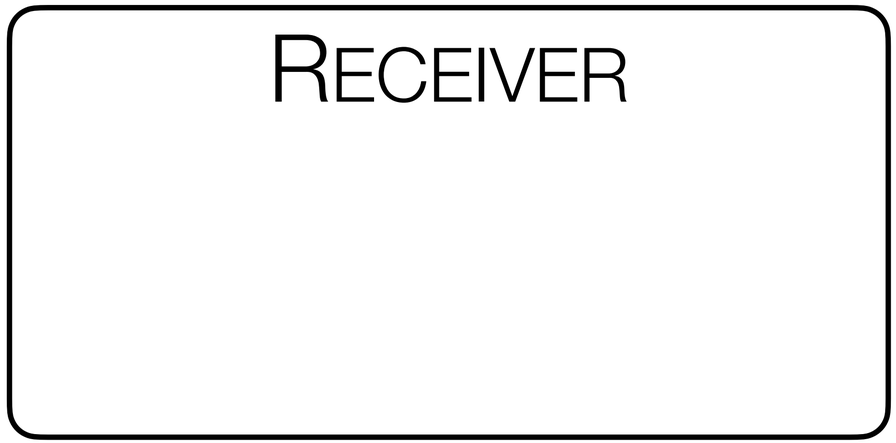



# Background: PC schemes

Commit to a polynomial, and later on prove its correct evaluation

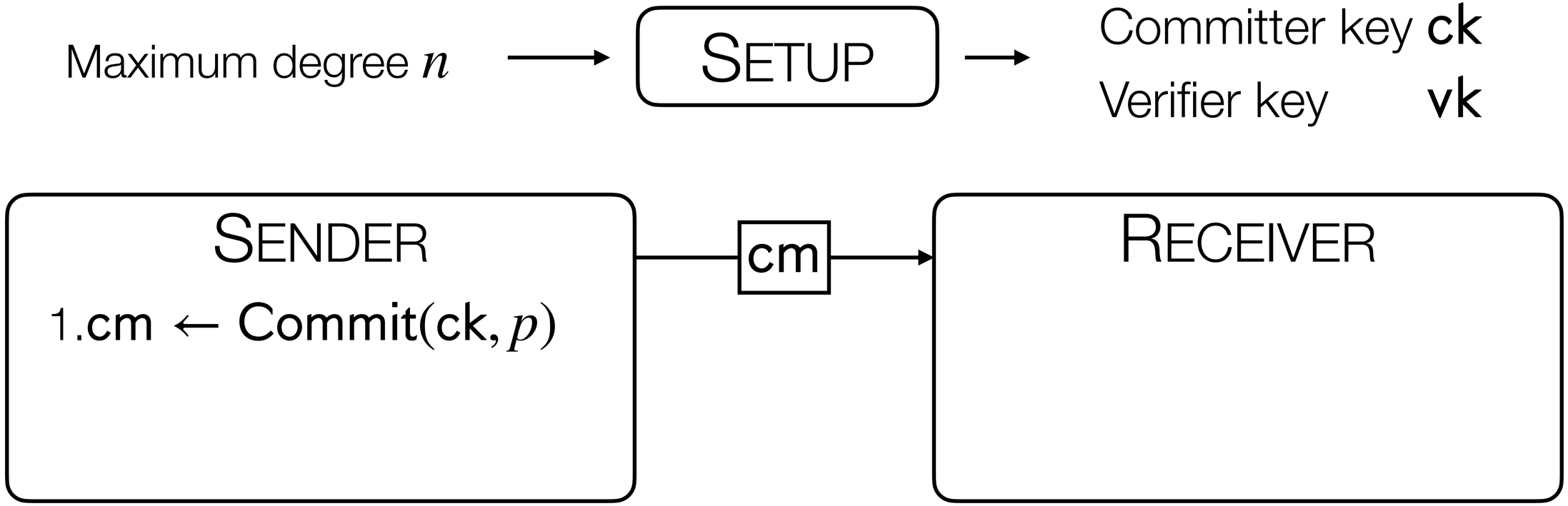


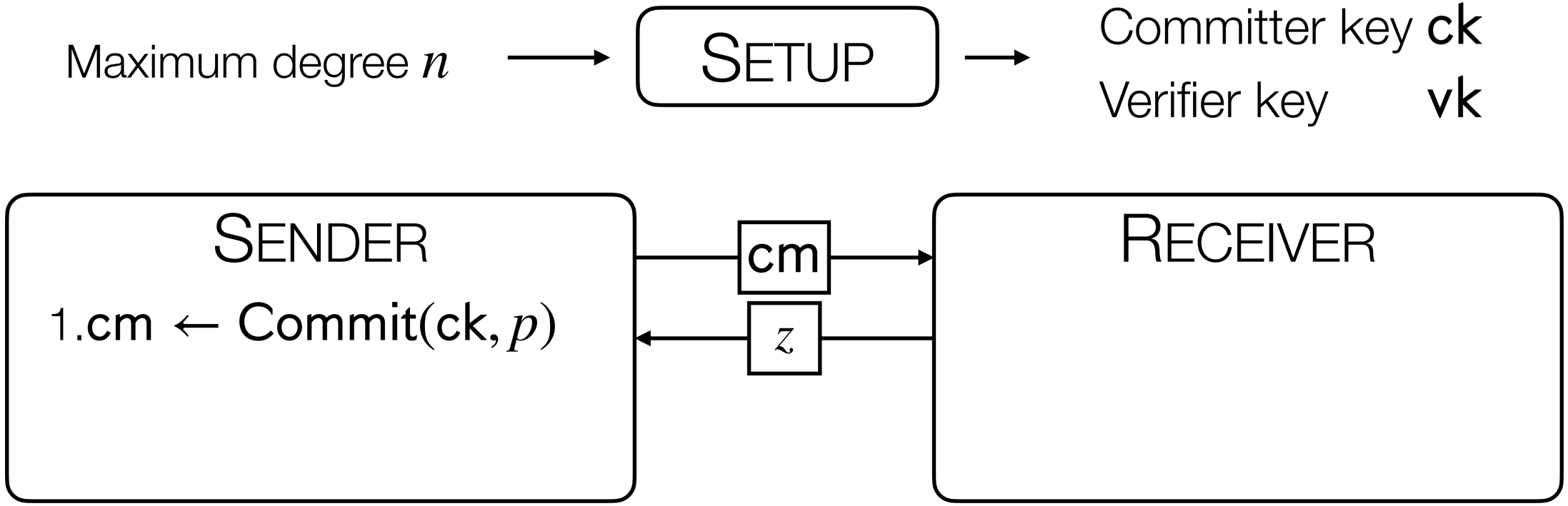
# Background: PC schemes

Commit to a polynomial, and later on prove its correct evaluation



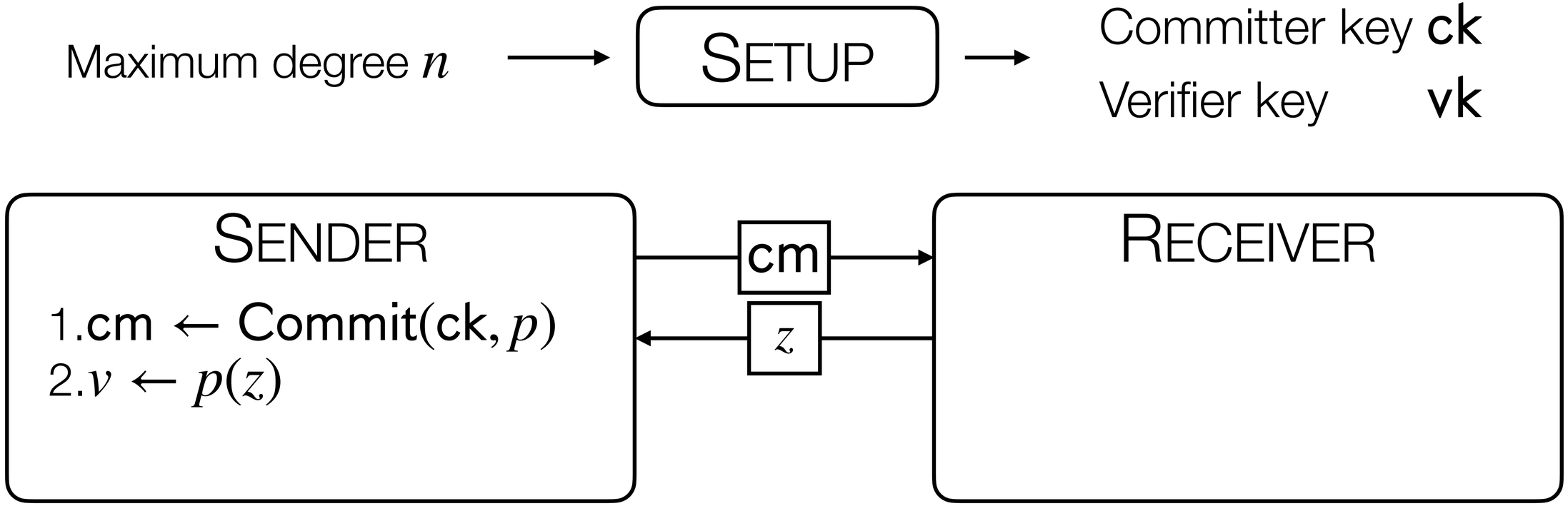
# Background: PC schemes

Commit to a polynomial, and later on prove its correct evaluation



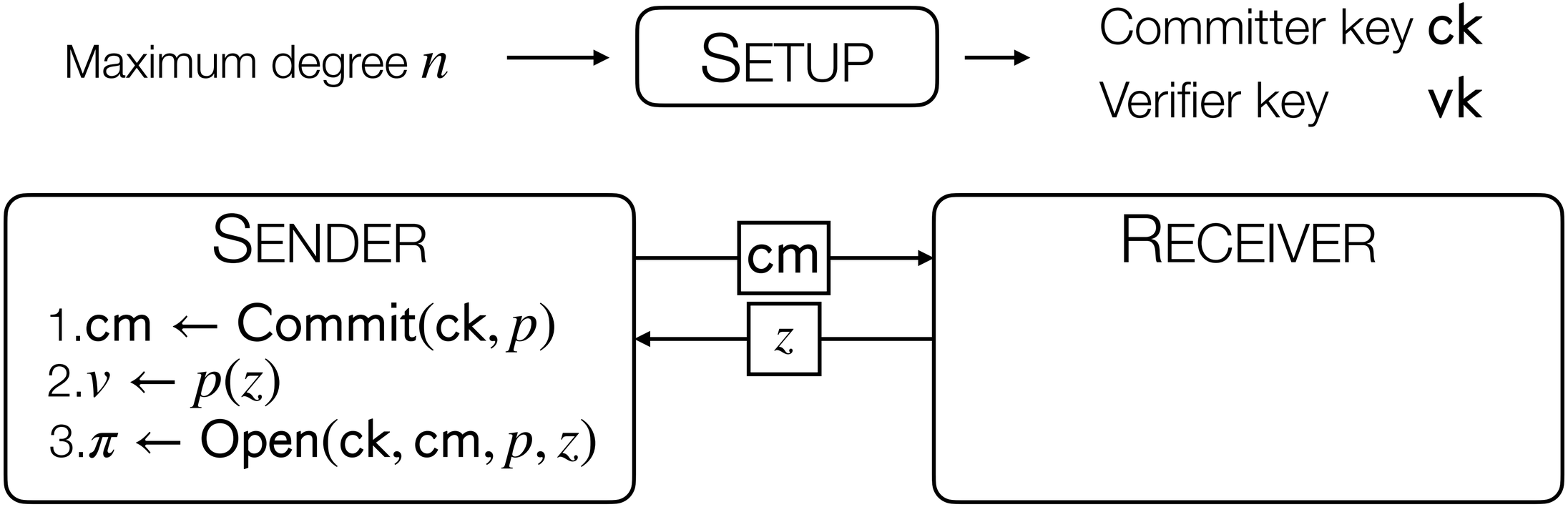
# Background: PC schemes

Commit to a polynomial, and later on prove its correct evaluation



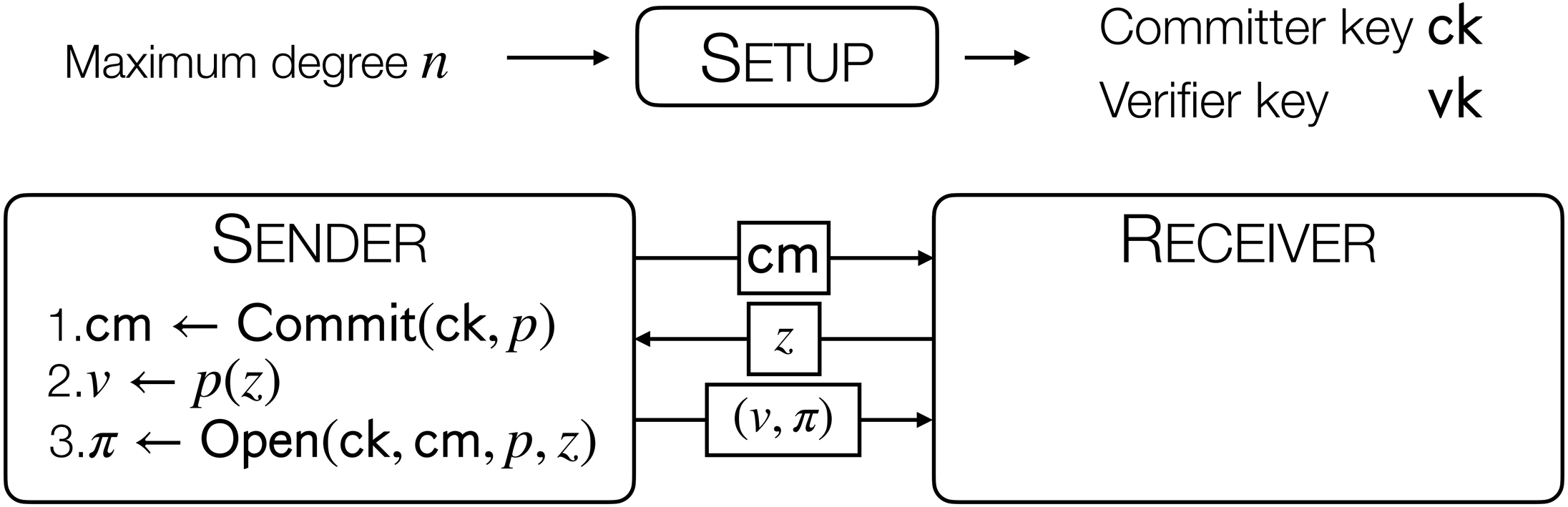
# Background: PC schemes

Commit to a polynomial, and later on prove its correct evaluation



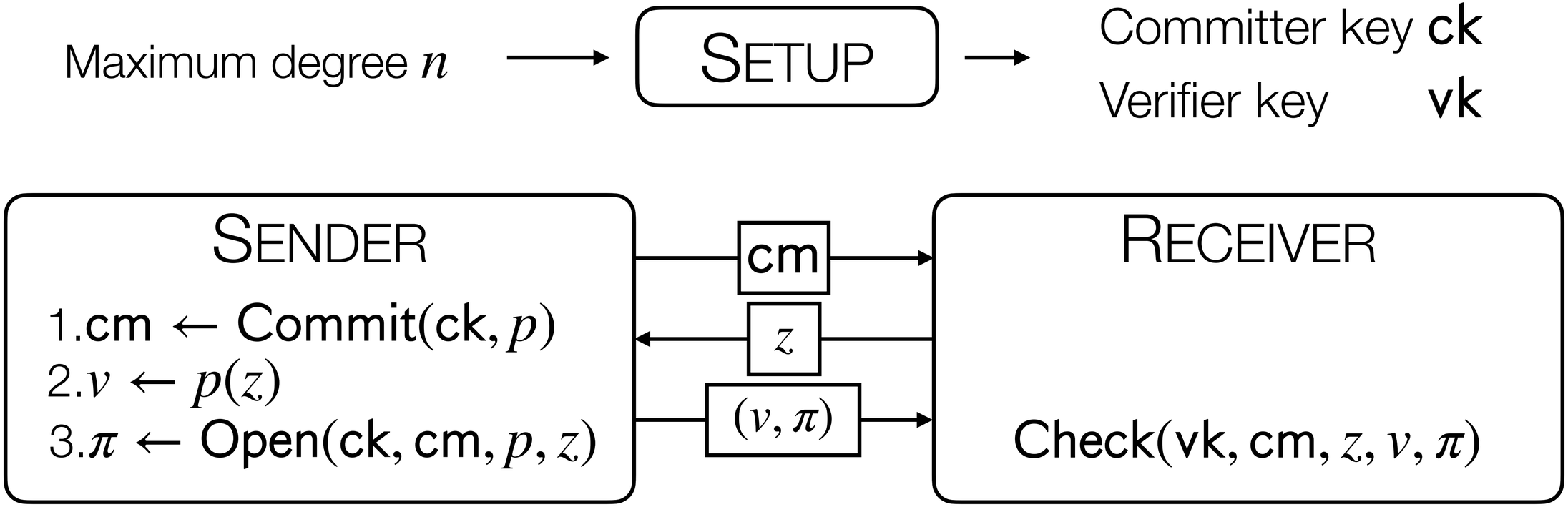
# Background: PC schemes

Commit to a polynomial, and later on prove its correct evaluation



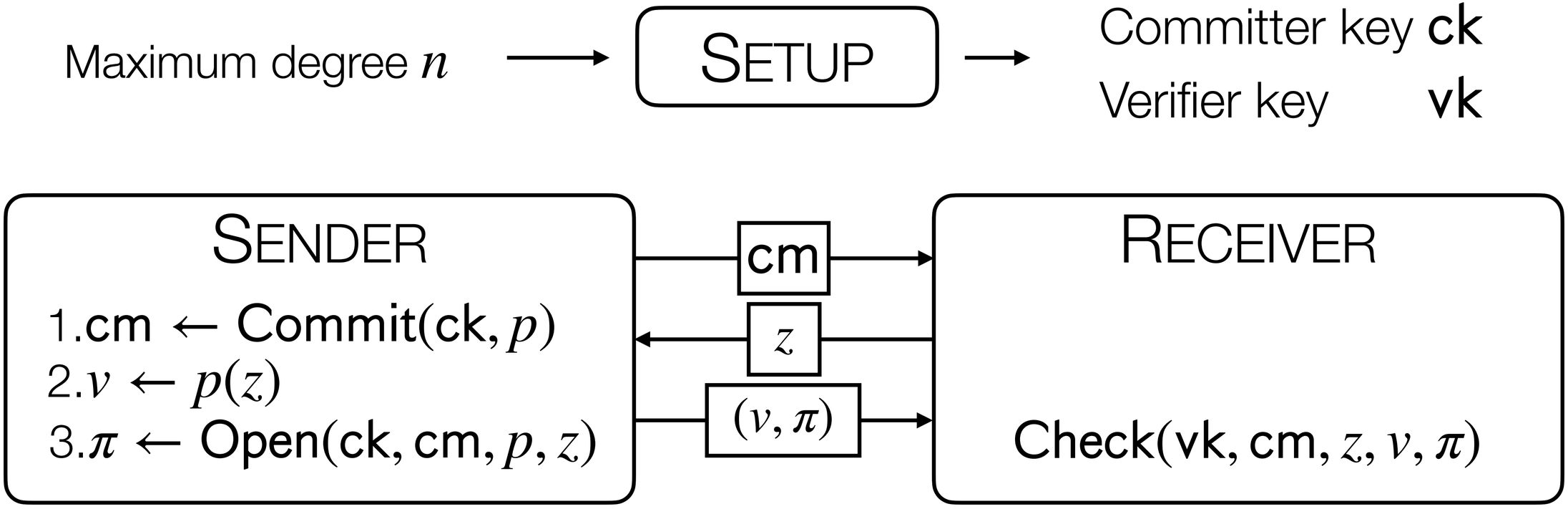
# Background: PC schemes

Commit to a polynomial, and later on prove its correct evaluation



# Background: PC schemes

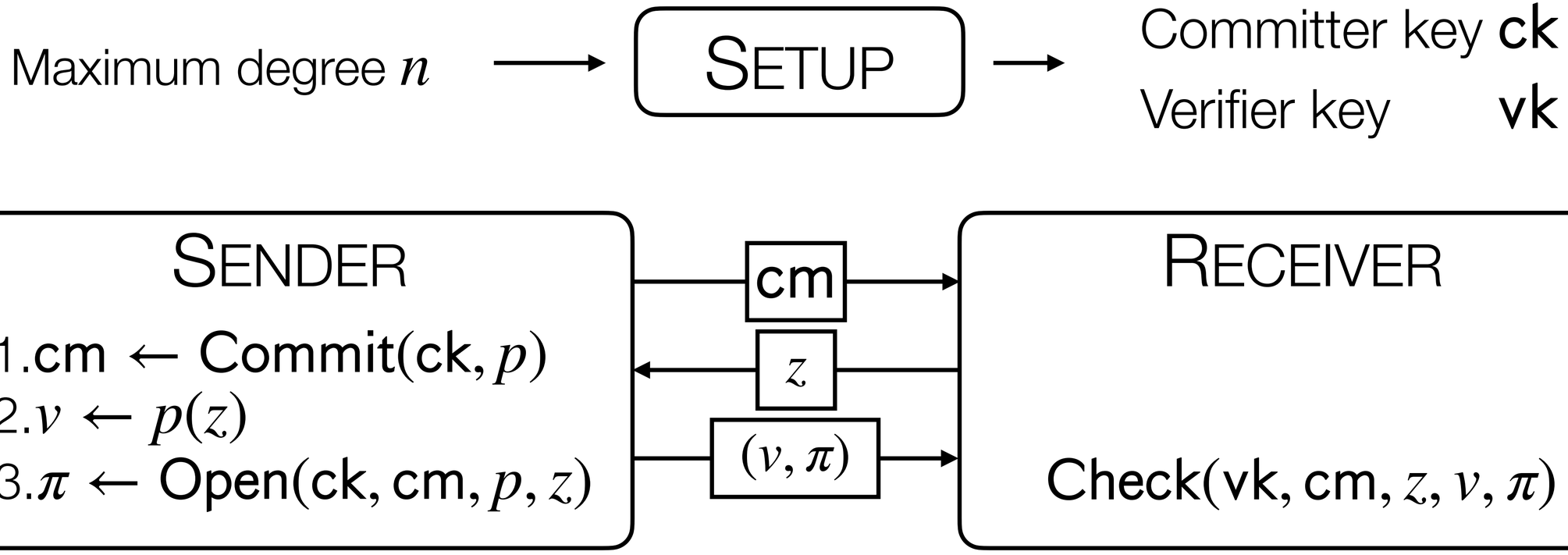
Commit to a polynomial, and later on prove its correct evaluation




- **Completeness:** Whenever $p(z) = v$, the Receiver accepts.

# Background: PC schemes

Commit to a polynomial, and later on prove its correct evaluation



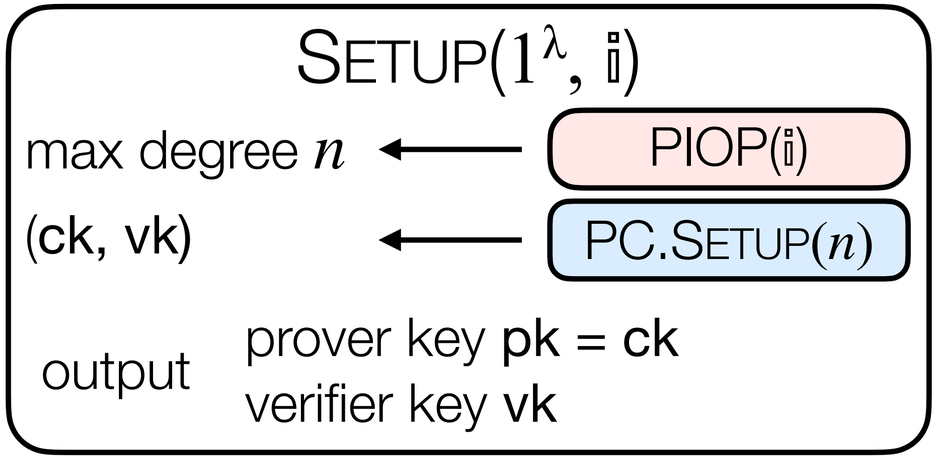

- **Completeness:** Whenever $p(z) = v$, the Receiver accepts.
- **Extractability:** Whenever the Receiver accepts, the Sender's commitment **cm** "contains" a polynomial $p$ satisfying $p(z) = v$.

# PIOPs + PC Schemes → SNARK

[CHMMVW20, BFS20]

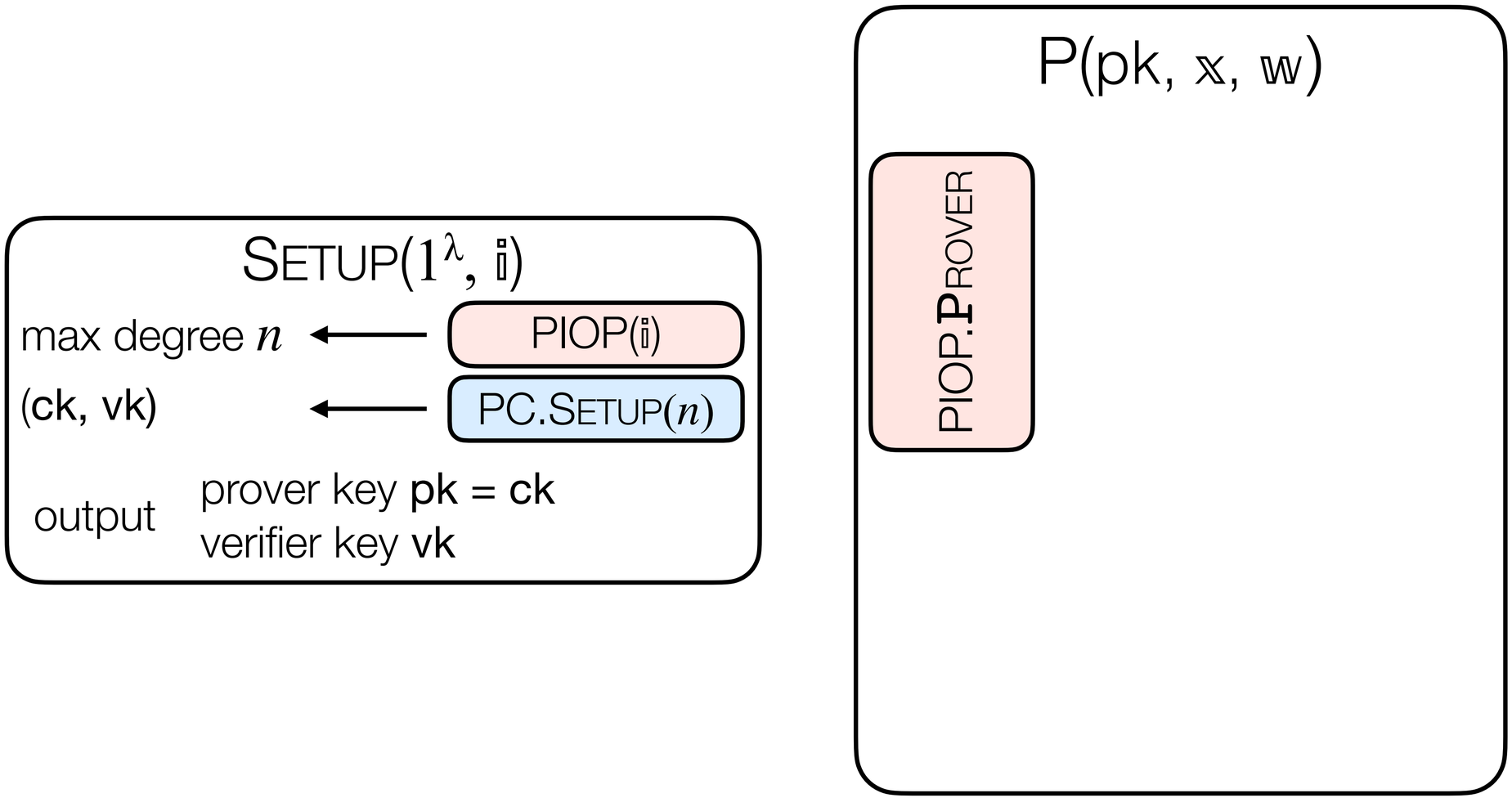
# PIOPs + PC Schemes → SNARK

[CHMMVW20, BFS20]



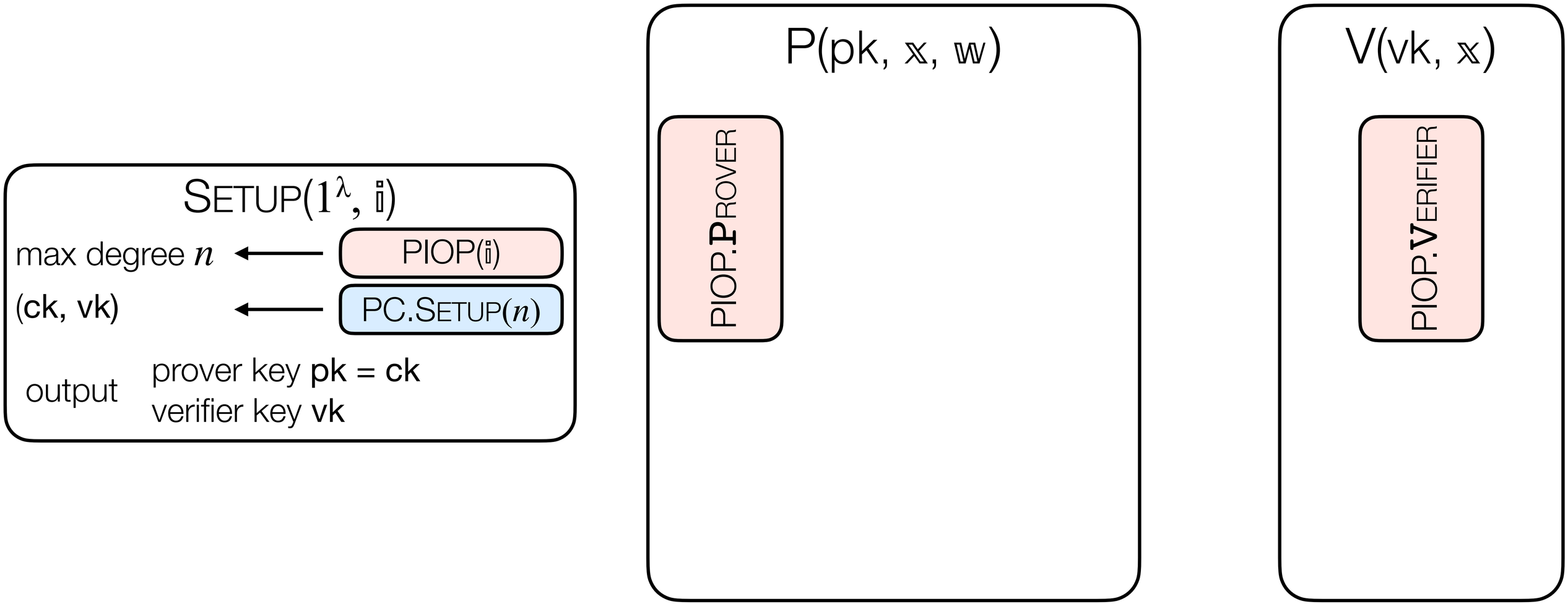
# PIOPs + PC Schemes → SNARK

[CHMMVW20, BFS20]



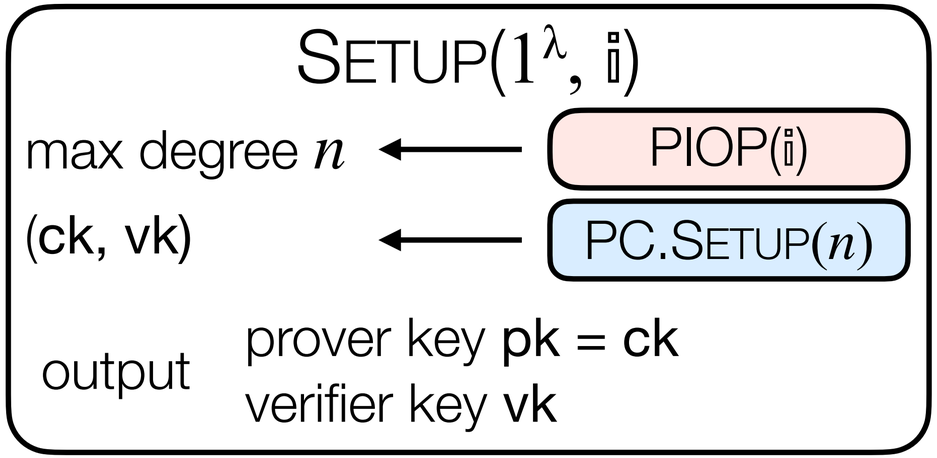
# PIOPs + PC Schemes → SNARK

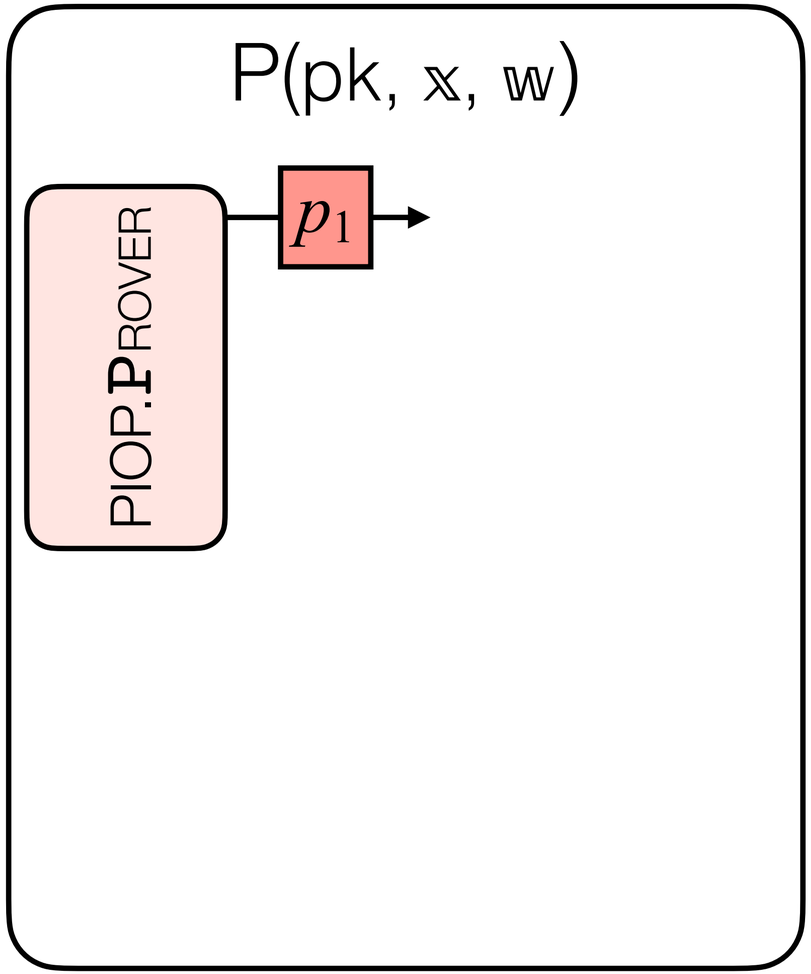
[CHMMVW20, BFS20]

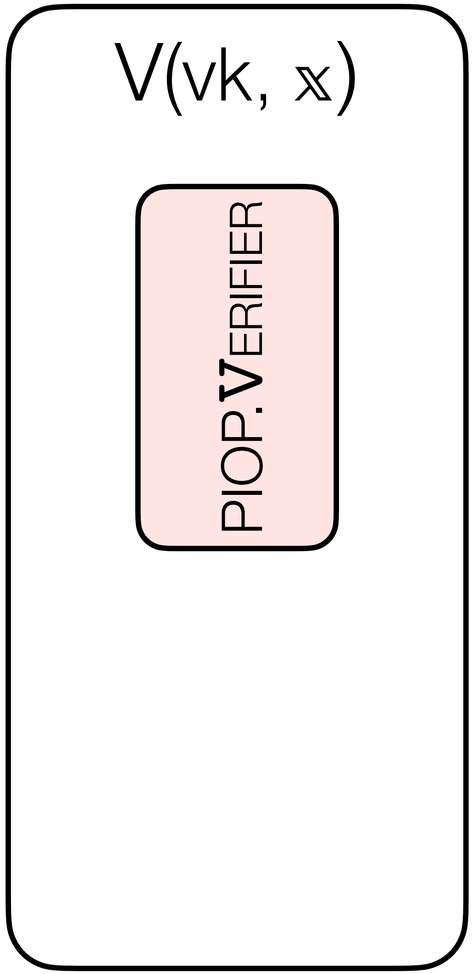


# PIOPs + PC Schemes → SNARK

[CHMMVW20, BFS20]

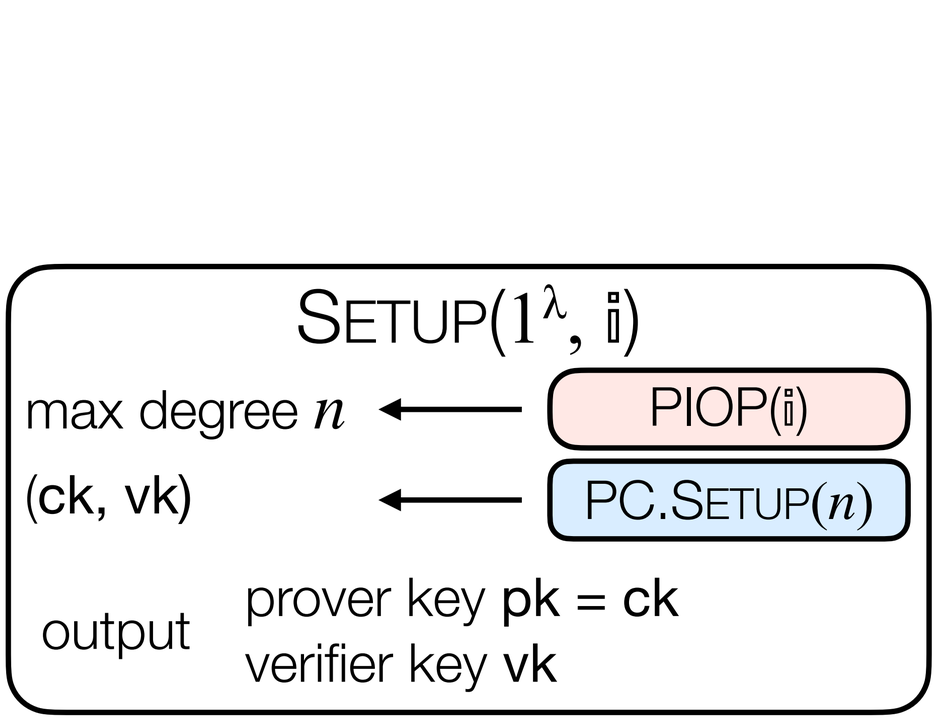


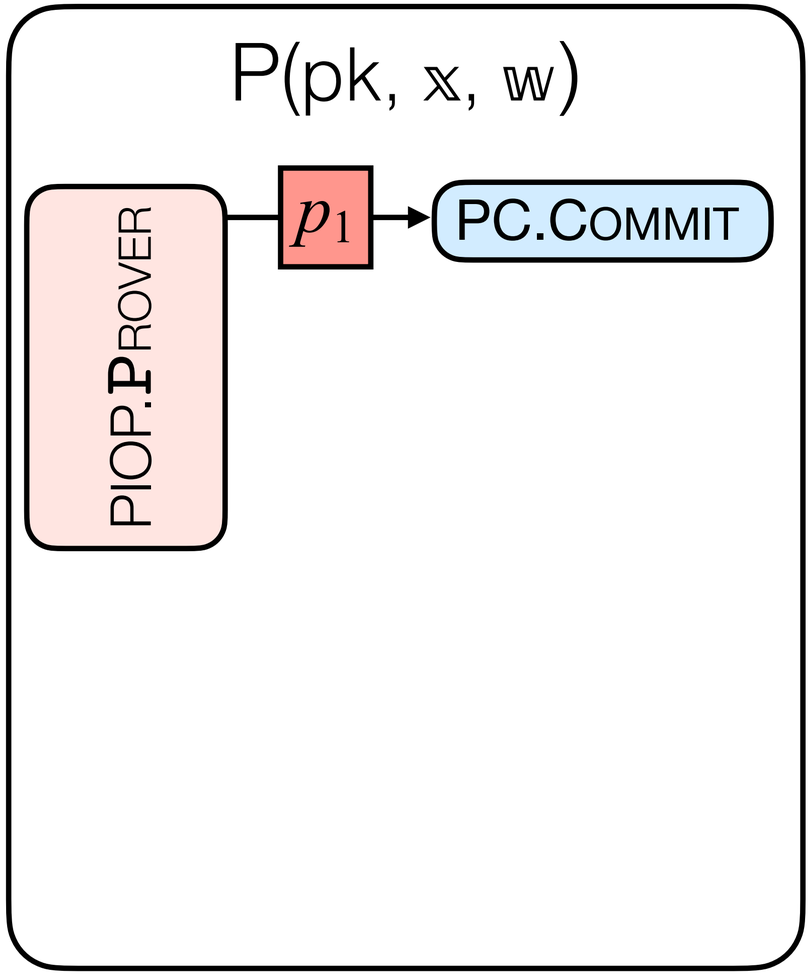



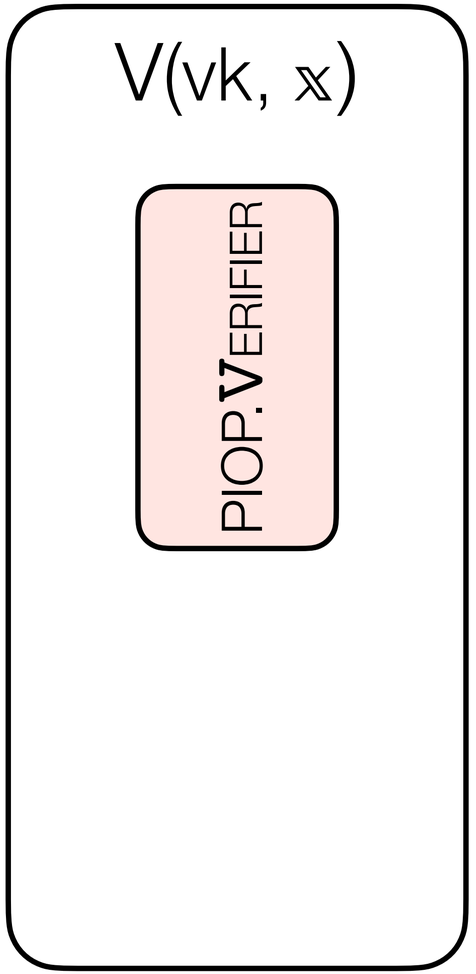



# PIOPs + PC Schemes → SNARK

[CHMMVW20, BFS20]

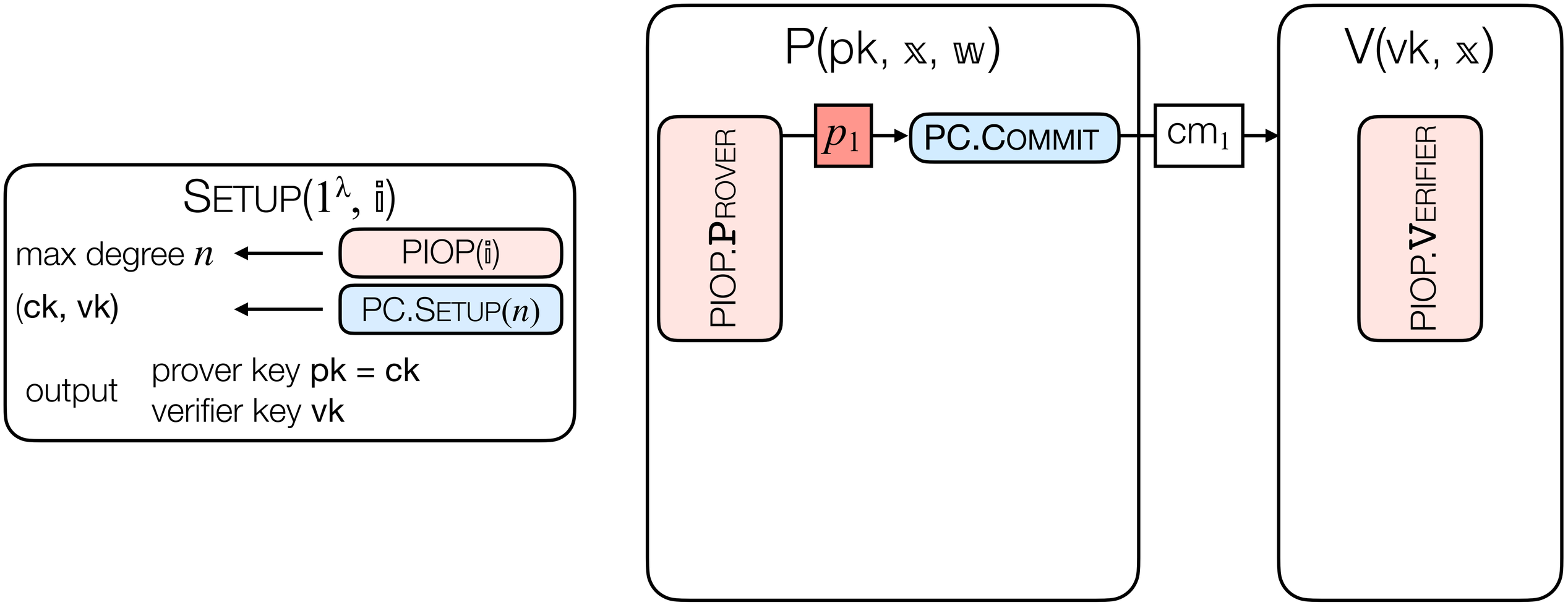


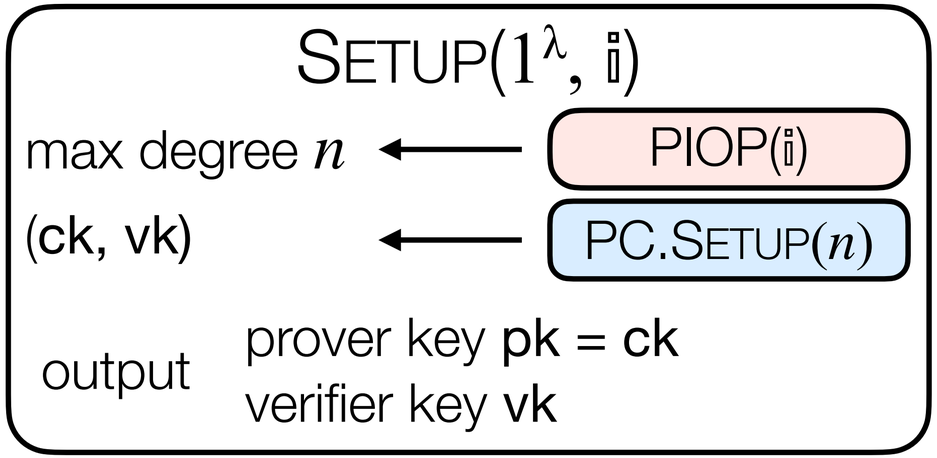
# PIOPs + PC Schemes → SNARK

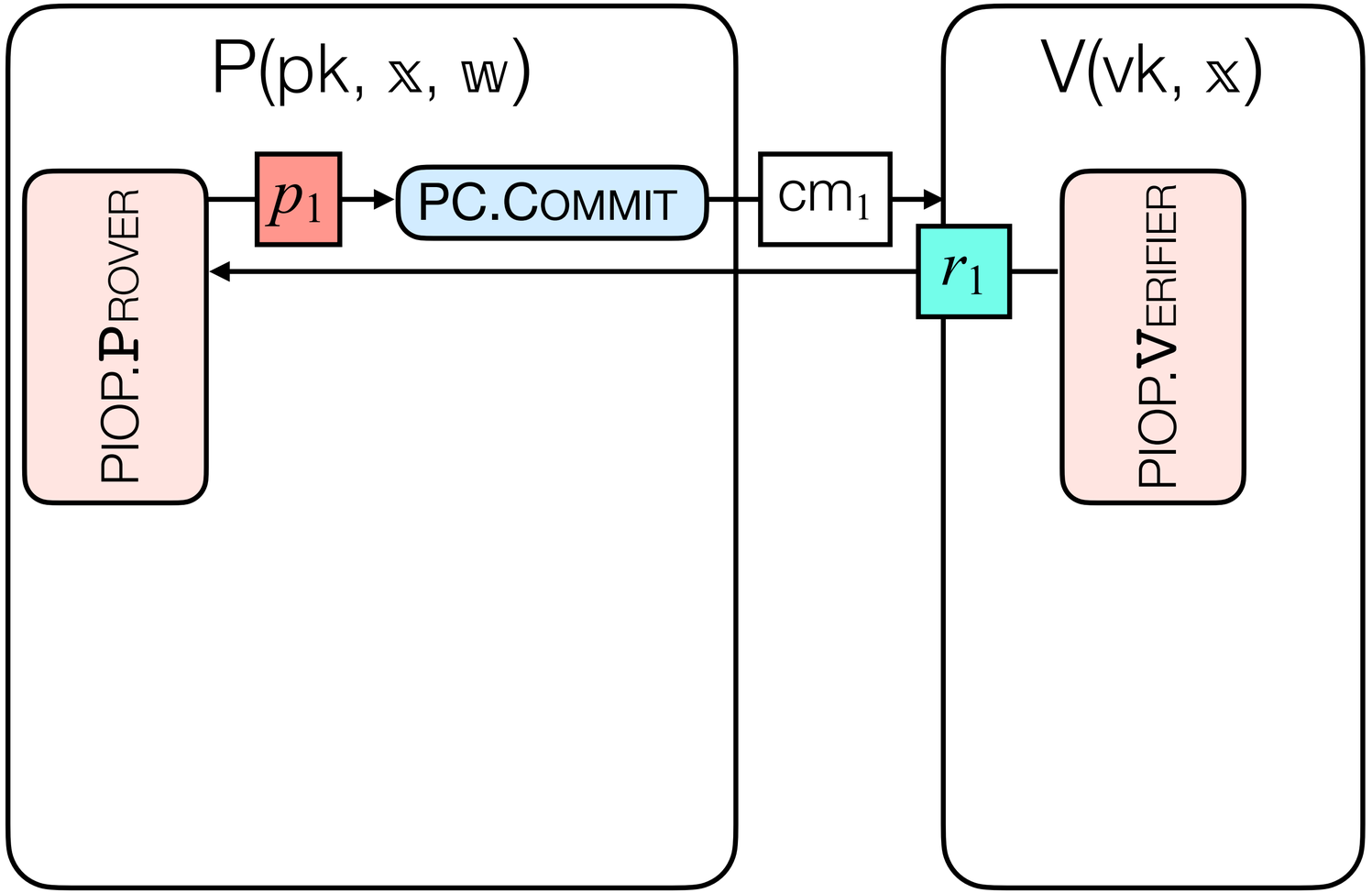
[CHMMVW20, BFS20]



# PIOPs + PC Schemes → SNARK

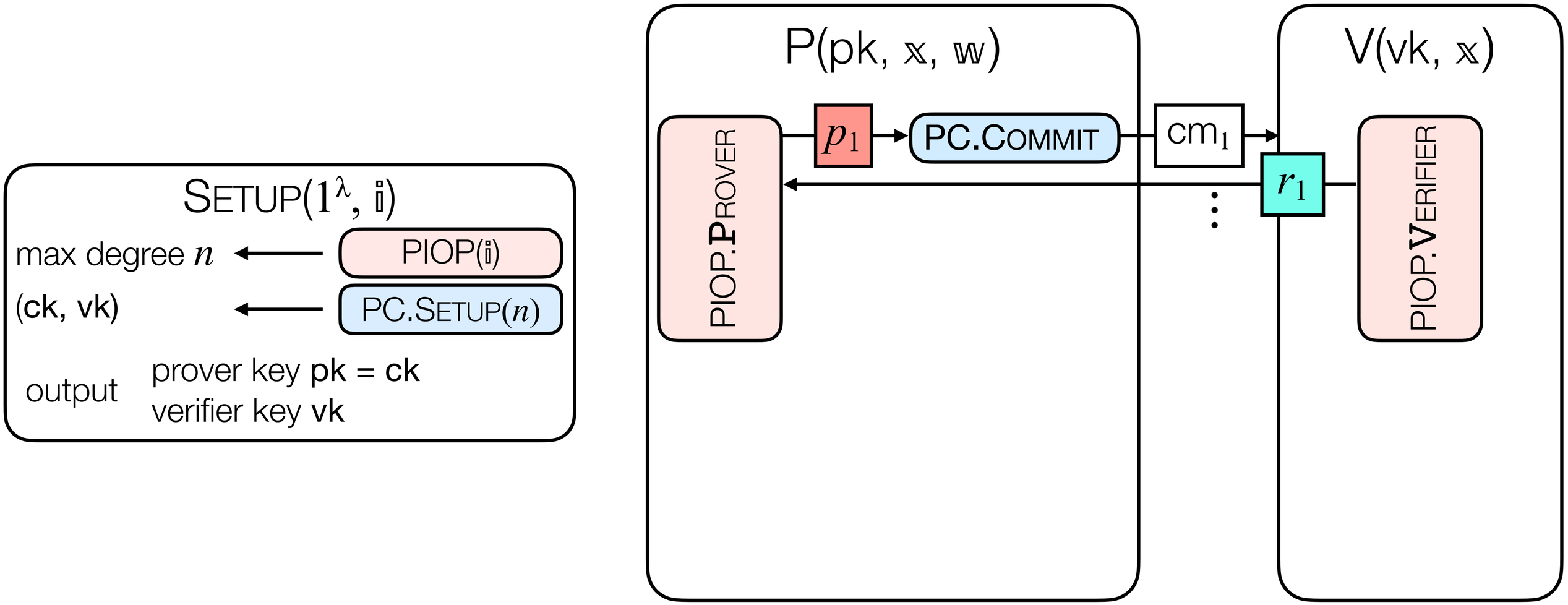
[CHMMVW20, BFS20]



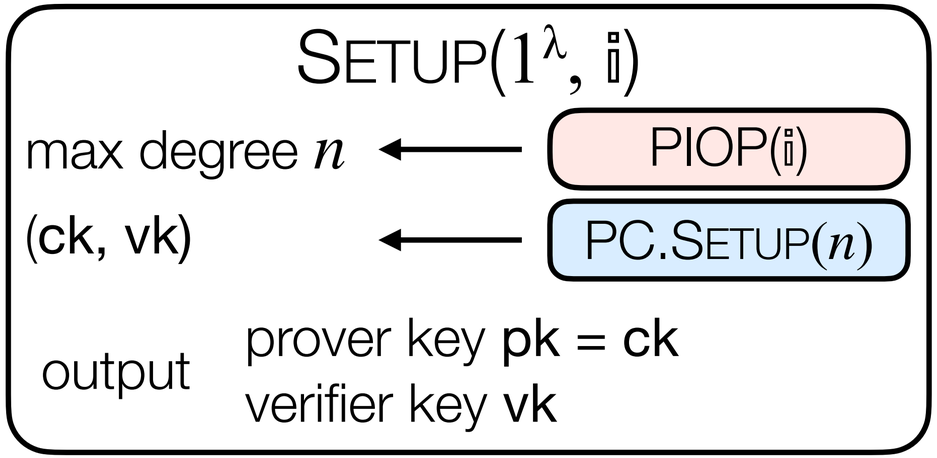
# PIOPs + PC Schemes → SNARK

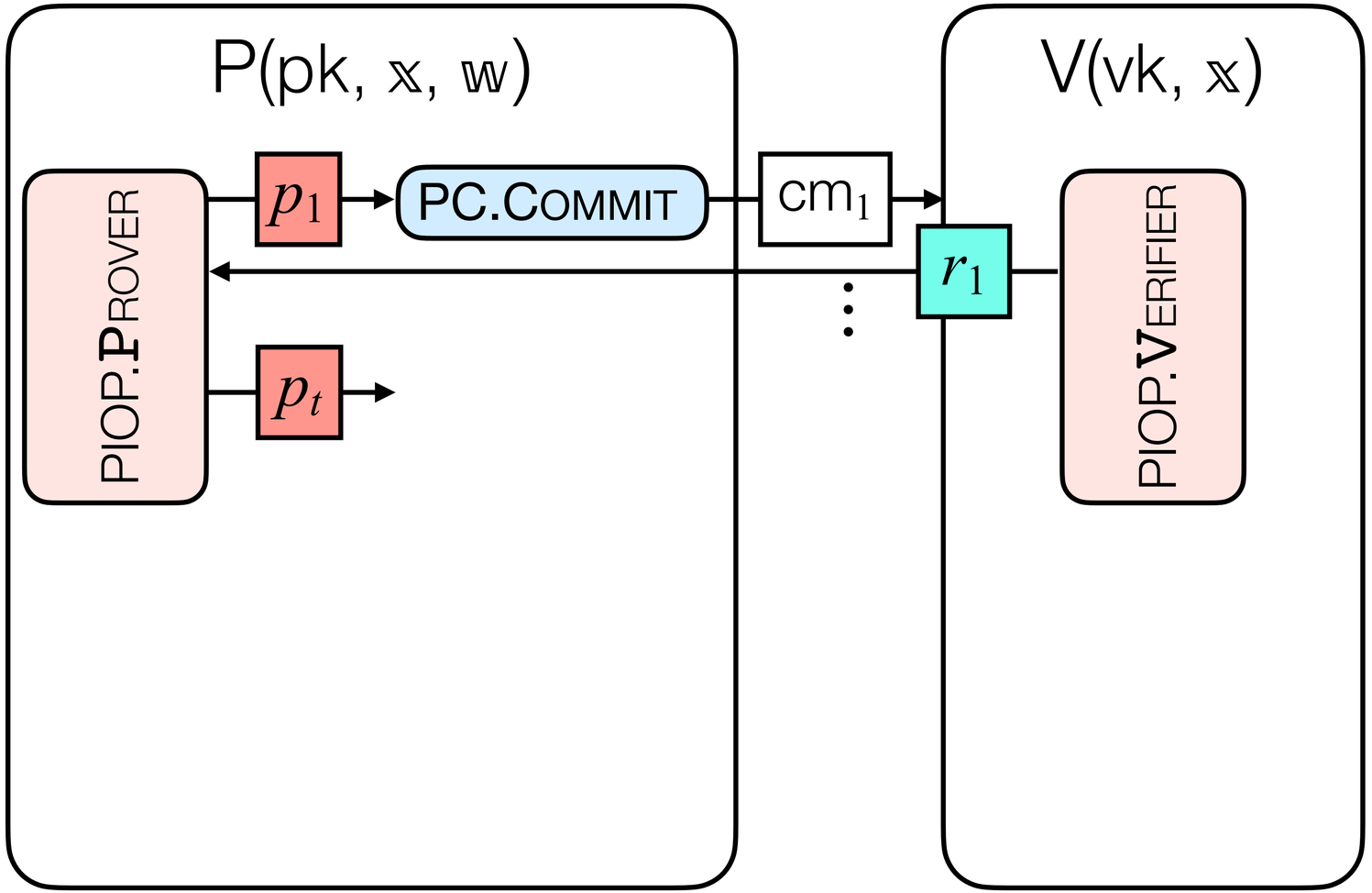
[CHMMVW20, BFS20]



# PIOPs + PC Schemes → SNARK

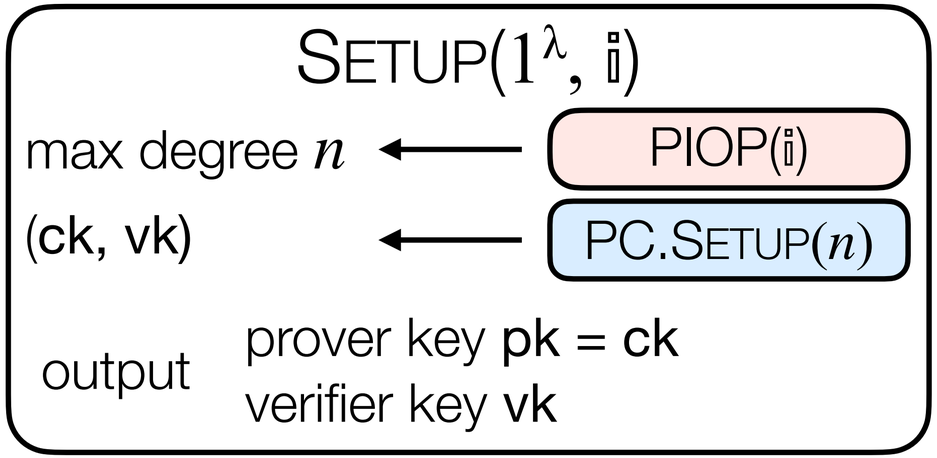
[CHMMVW20, BFS20]

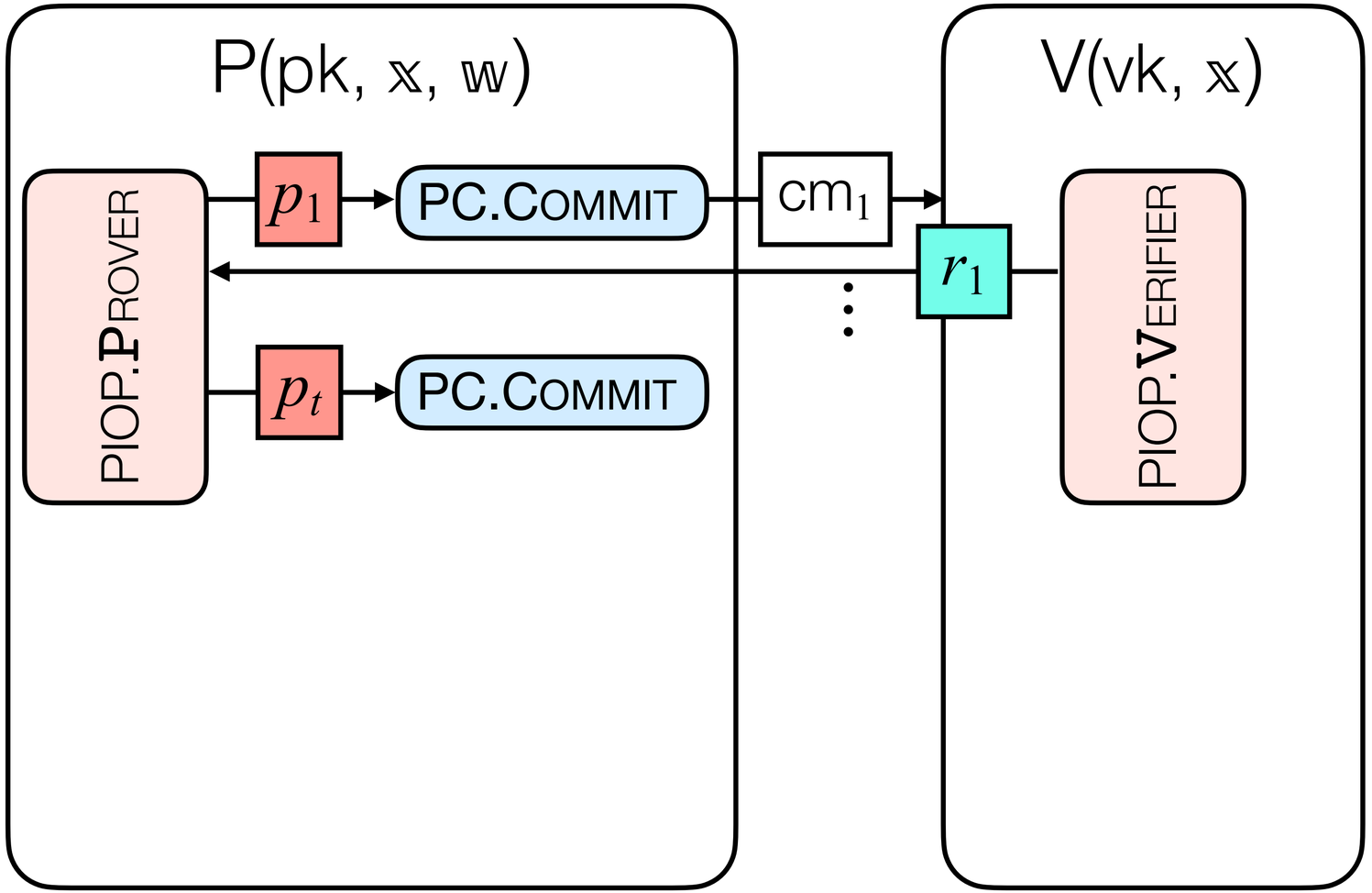


# PIOPs + PC Schemes → SNARK

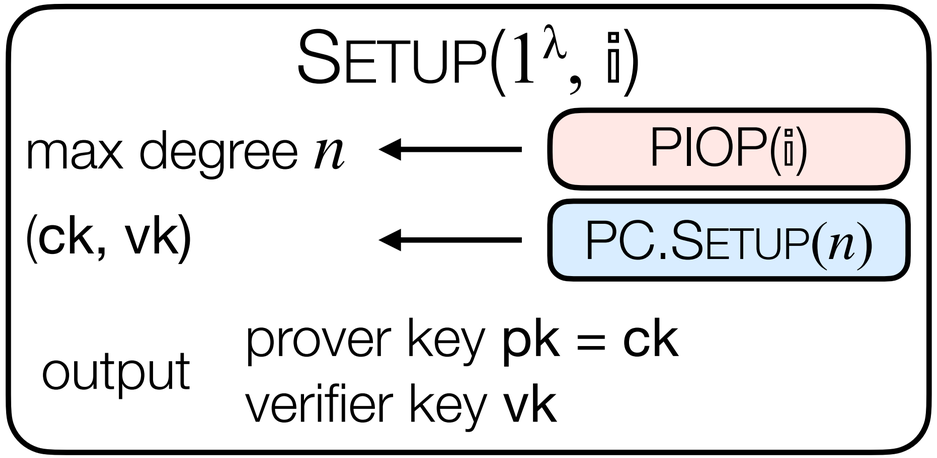
[CHMMVW20, BFS20]

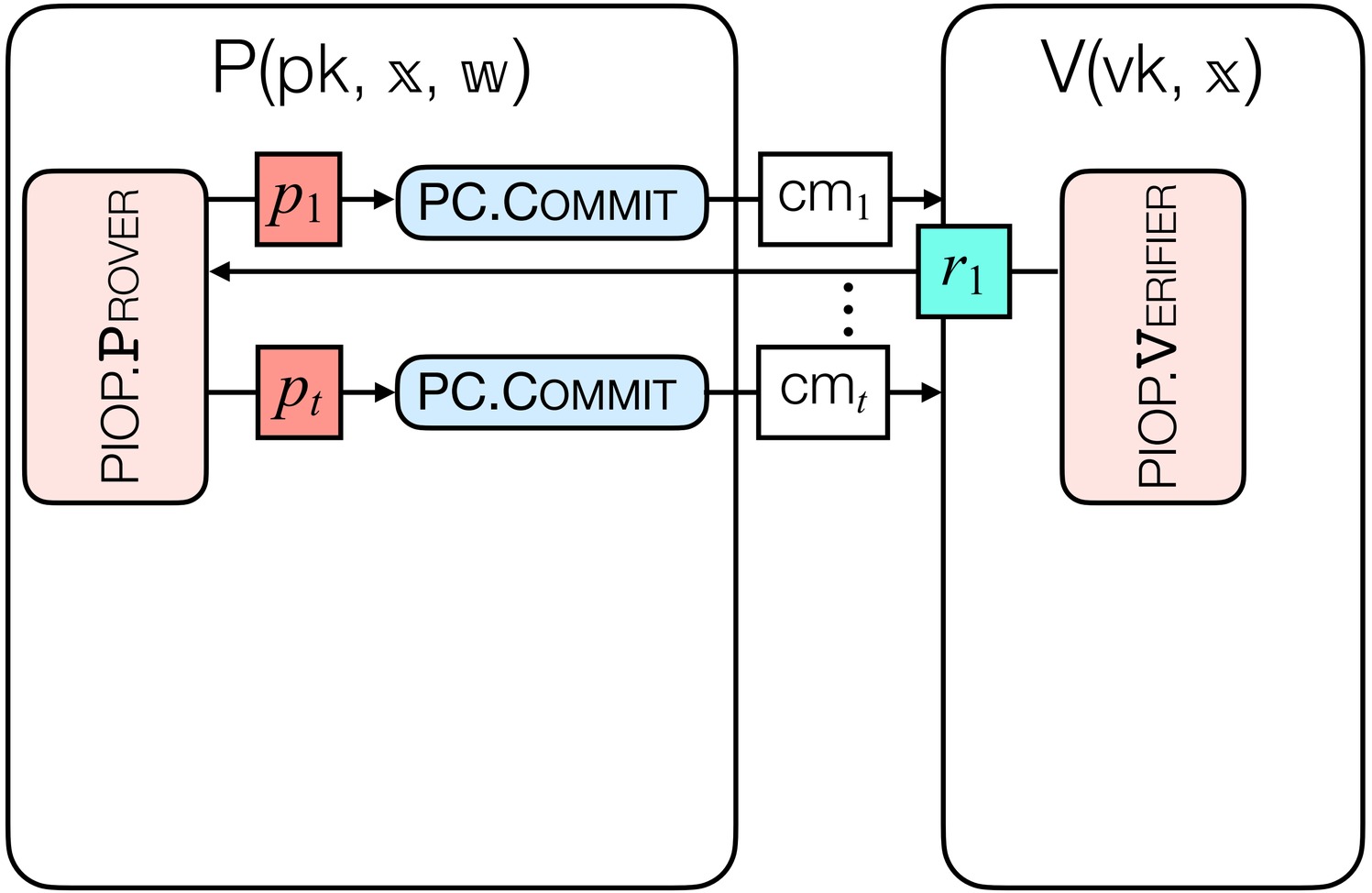


# PIOPs + PC Schemes → SNARK

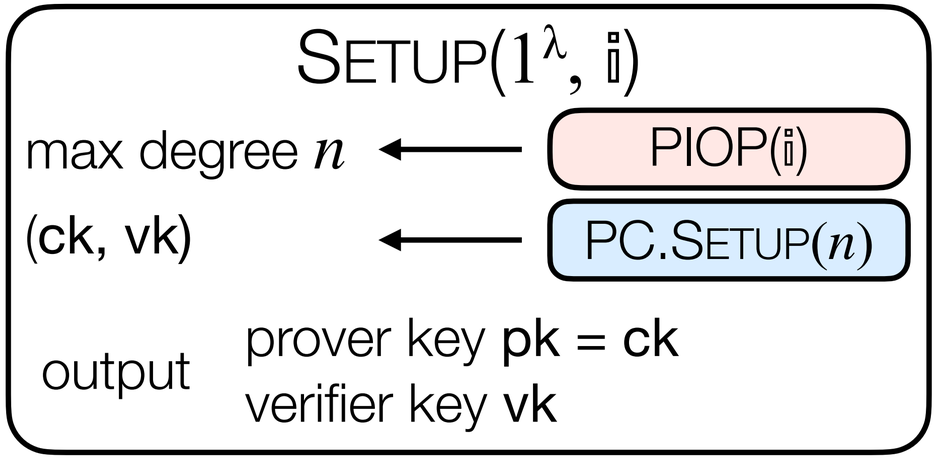
[CHMMVW20, BFS20]

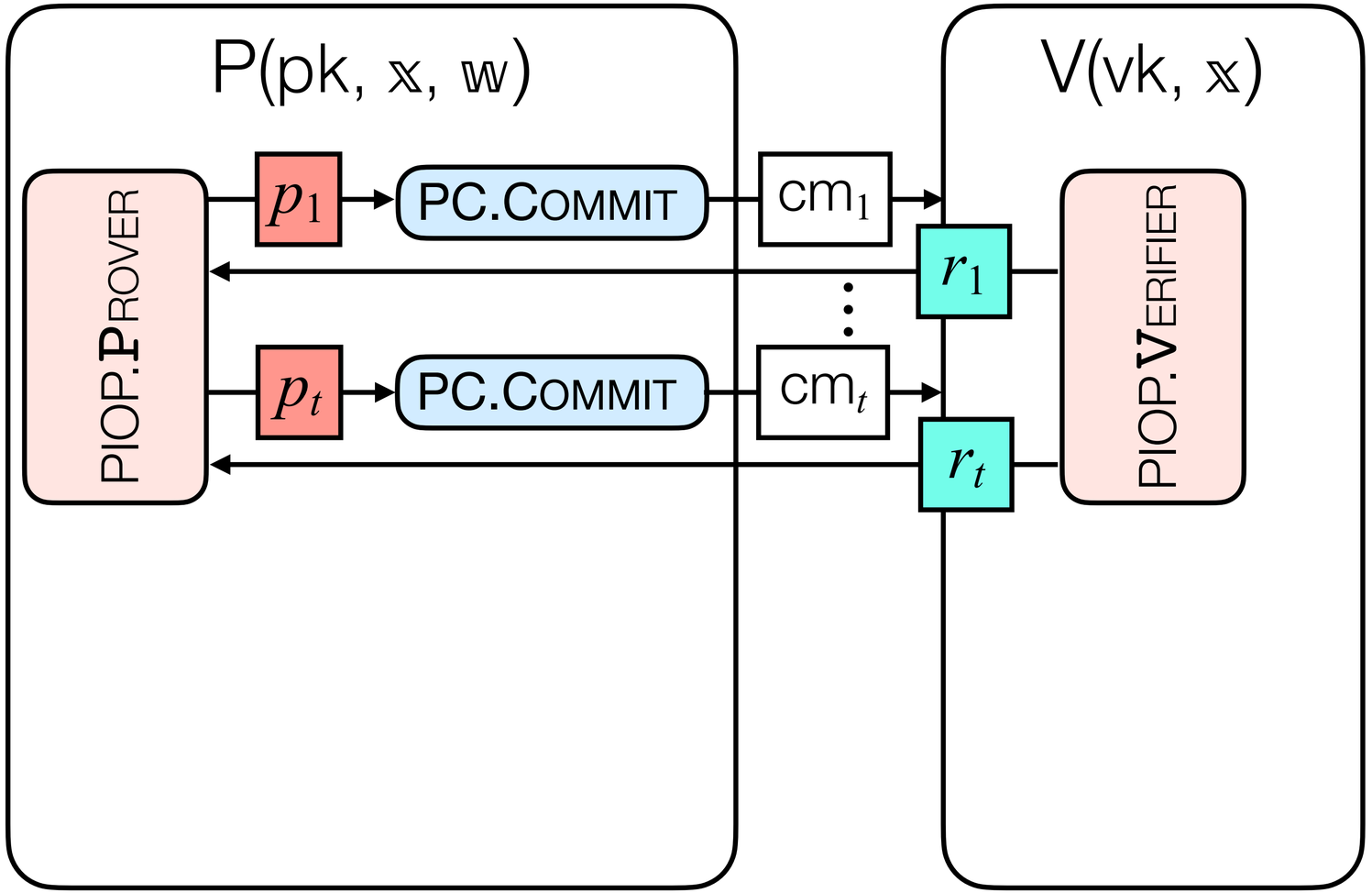


# PIOPs + PC Schemes → SNARK

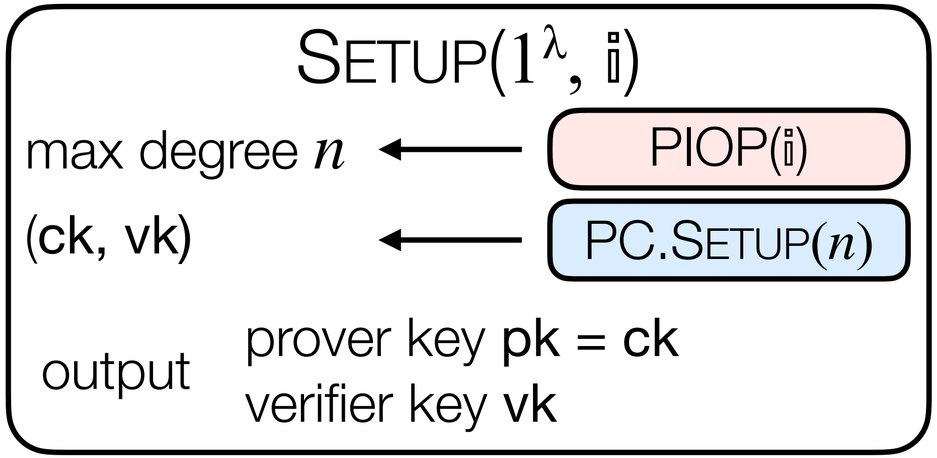
[CHMMVW20, BFS20]

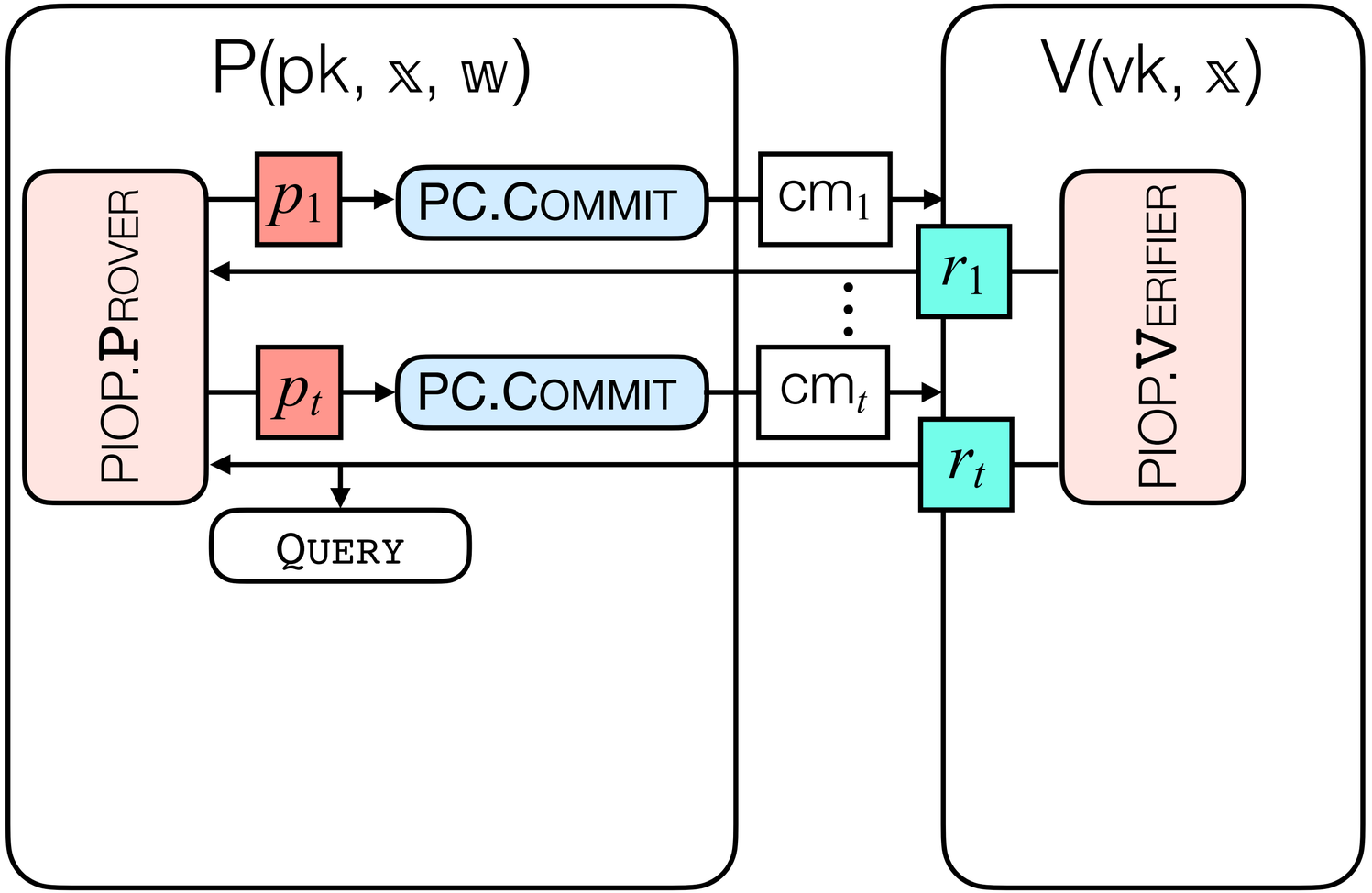


# PIOPs + PC Schemes → SNARK

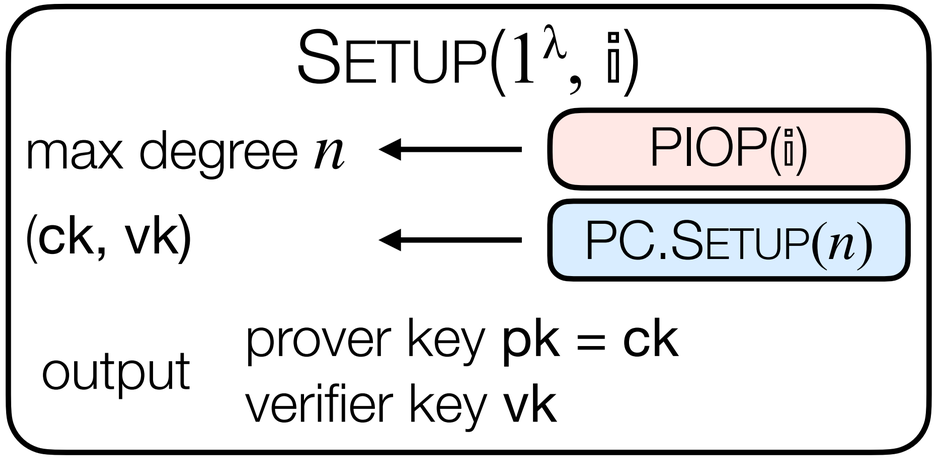
[CHMMVW20, BFS20]

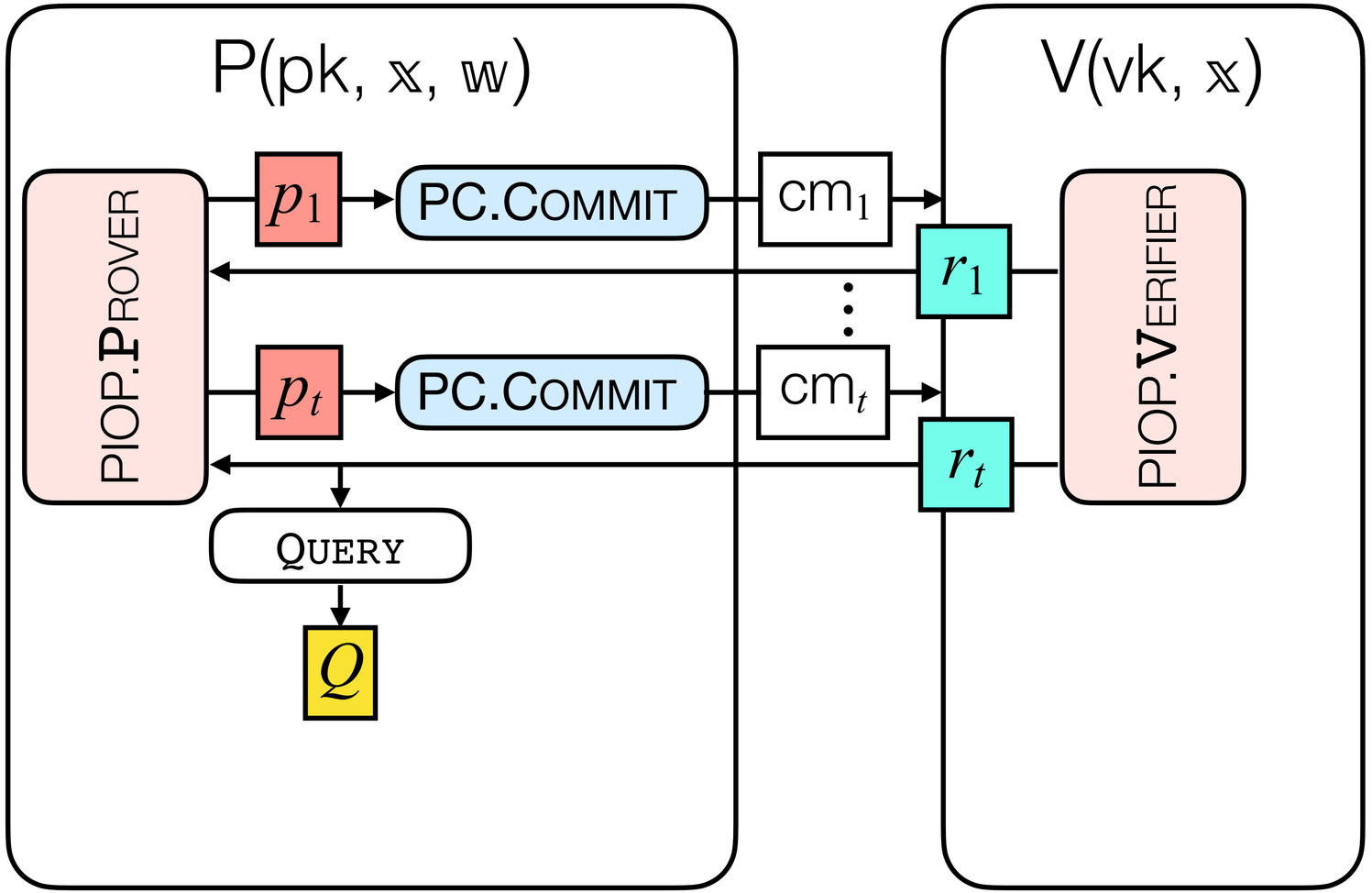


# PIOPs + PC Schemes → SNARK

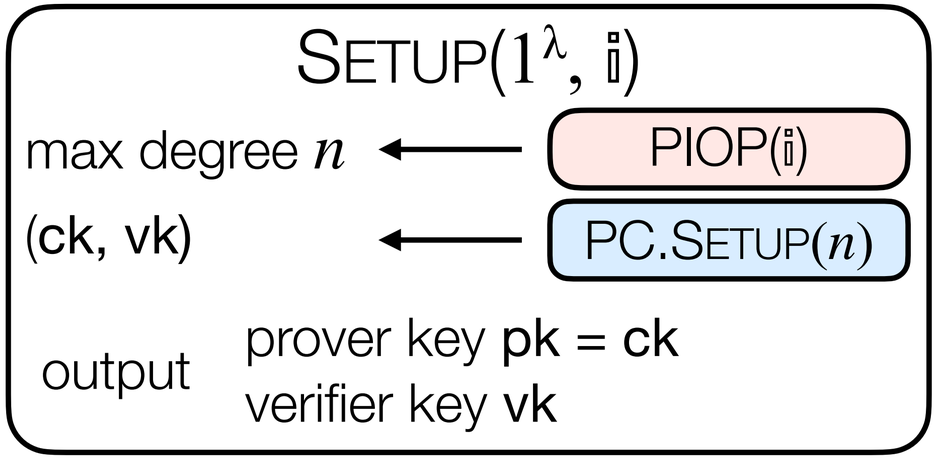
[CHMMVW20, BFS20]

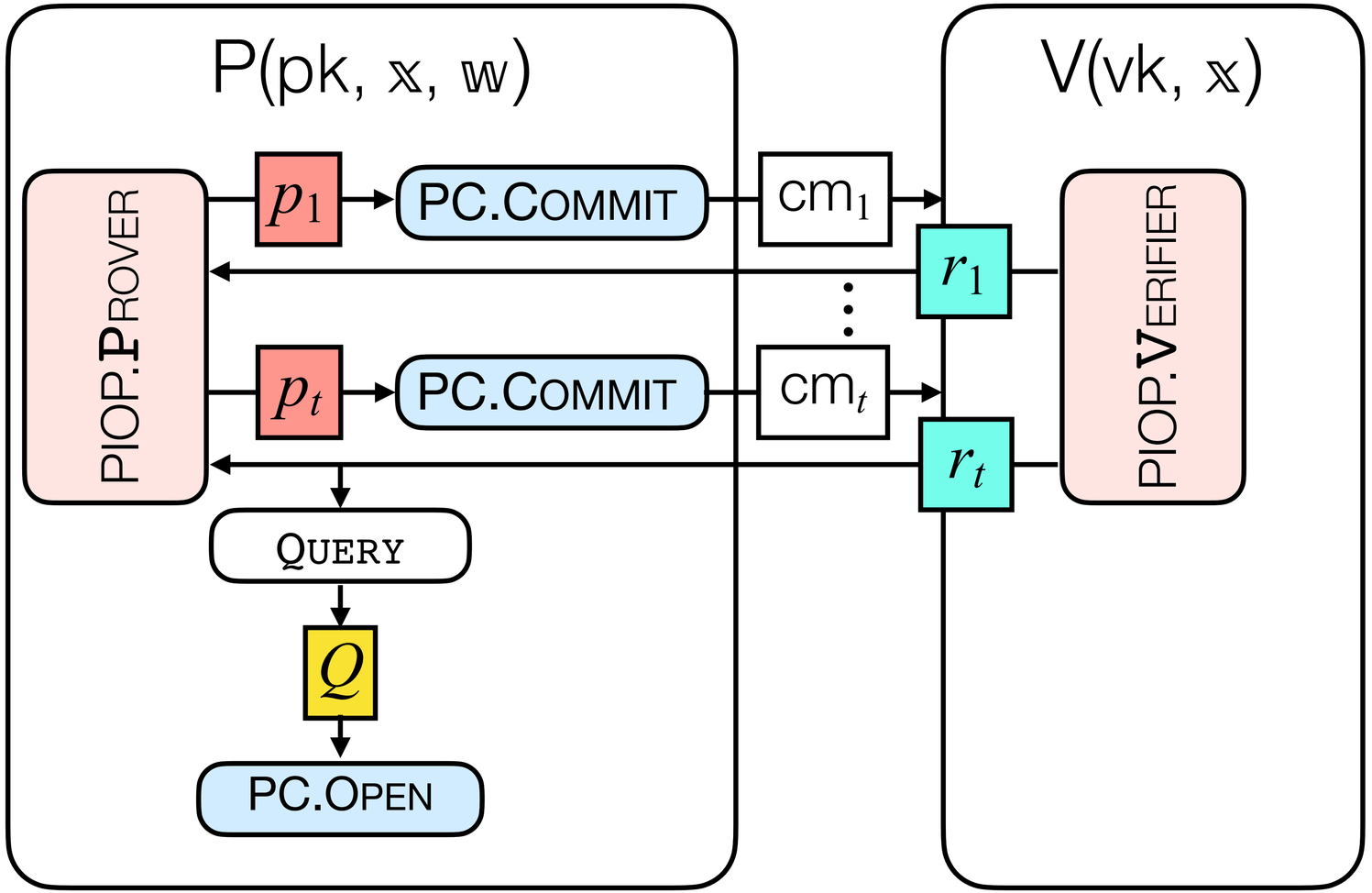


# PIOPs + PC Schemes → SNARK

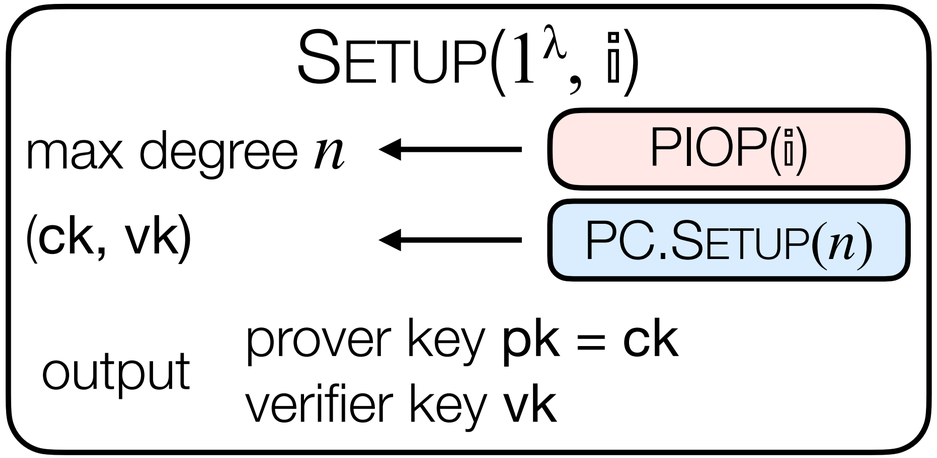
[CHMMVW20, BFS20]

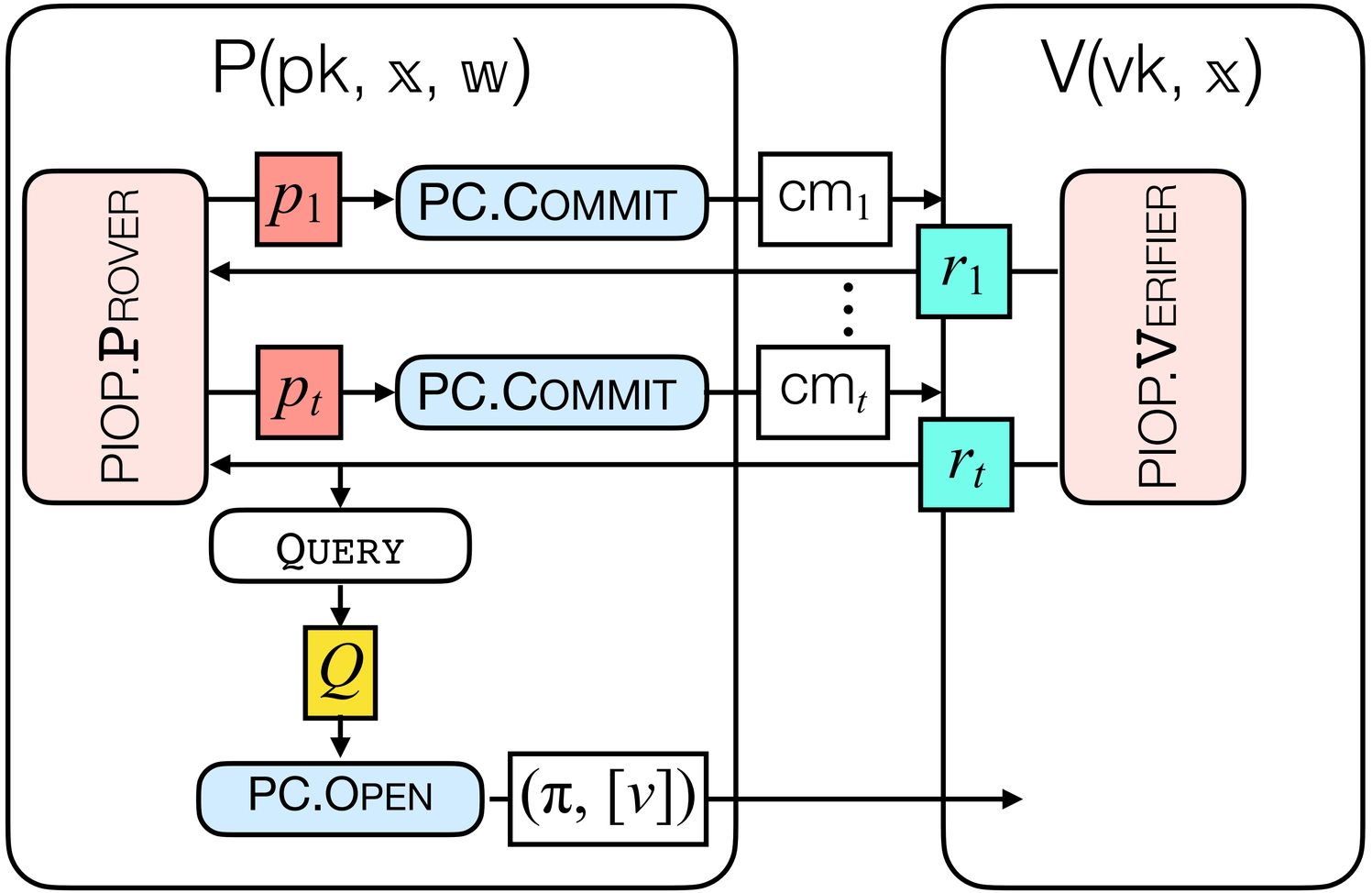


# PIOPs + PC Schemes → SNARK

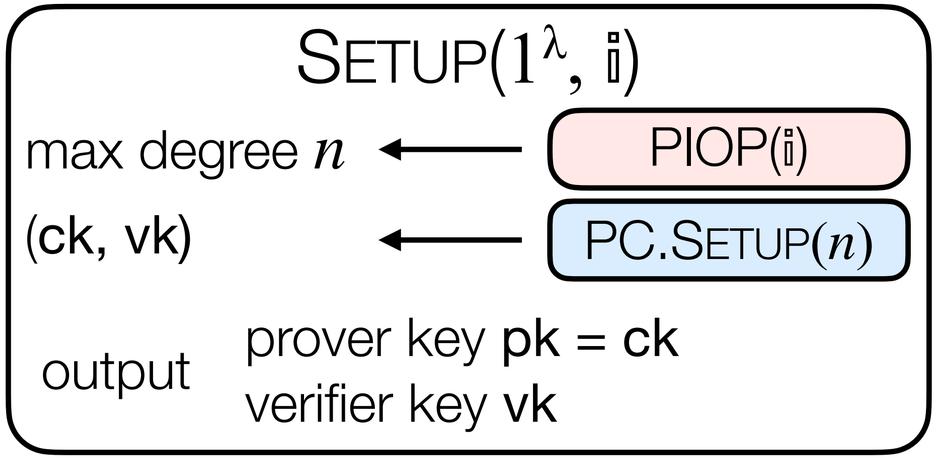
[CHMMVW20, BFS20]

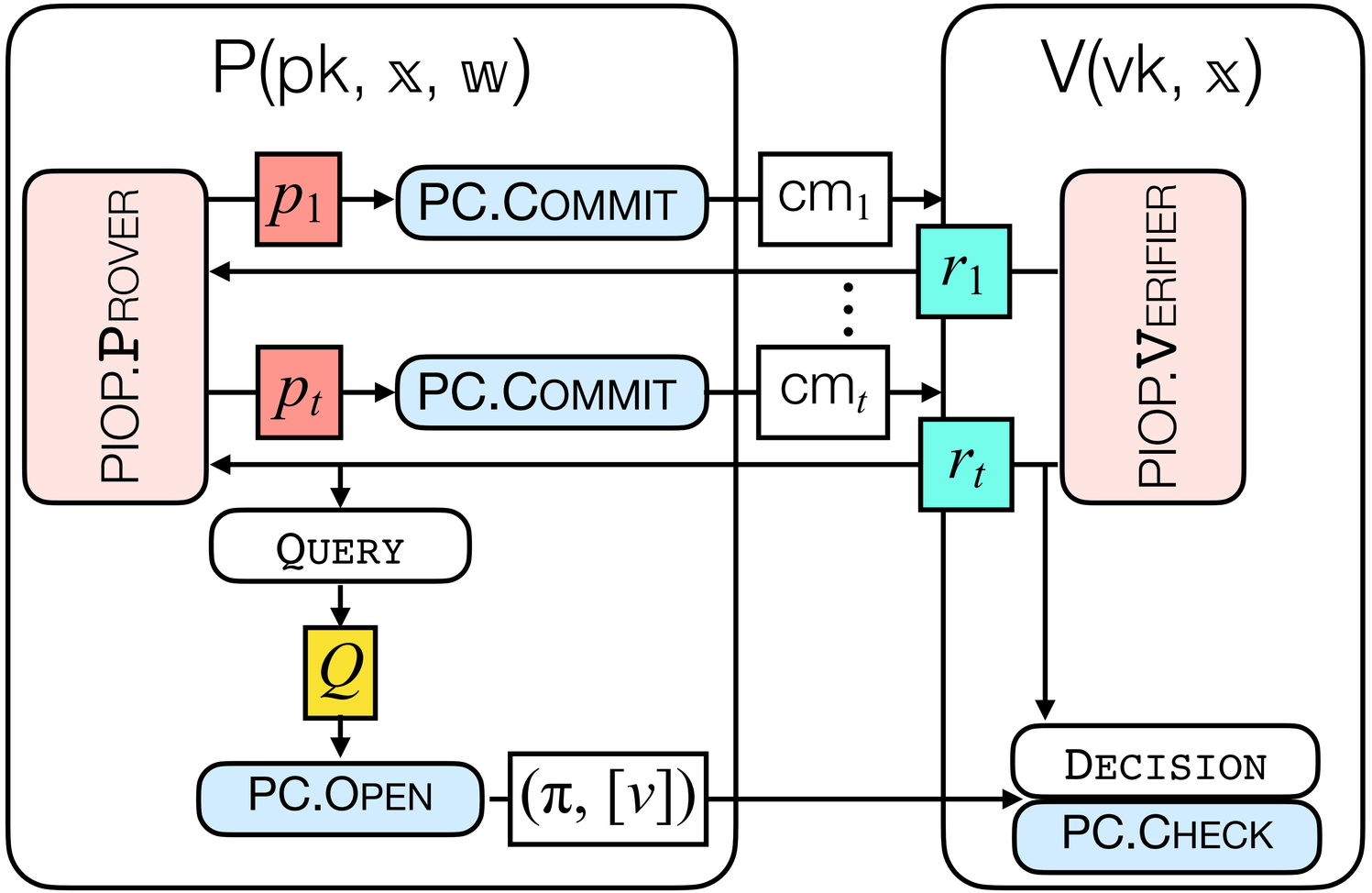


# PIOPs + PC Schemes → SNARK

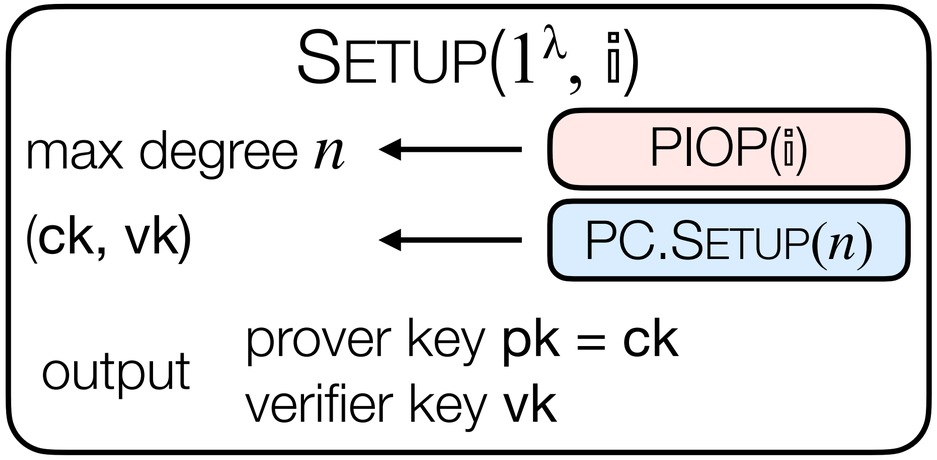
[CHMMVW20, BFS20]

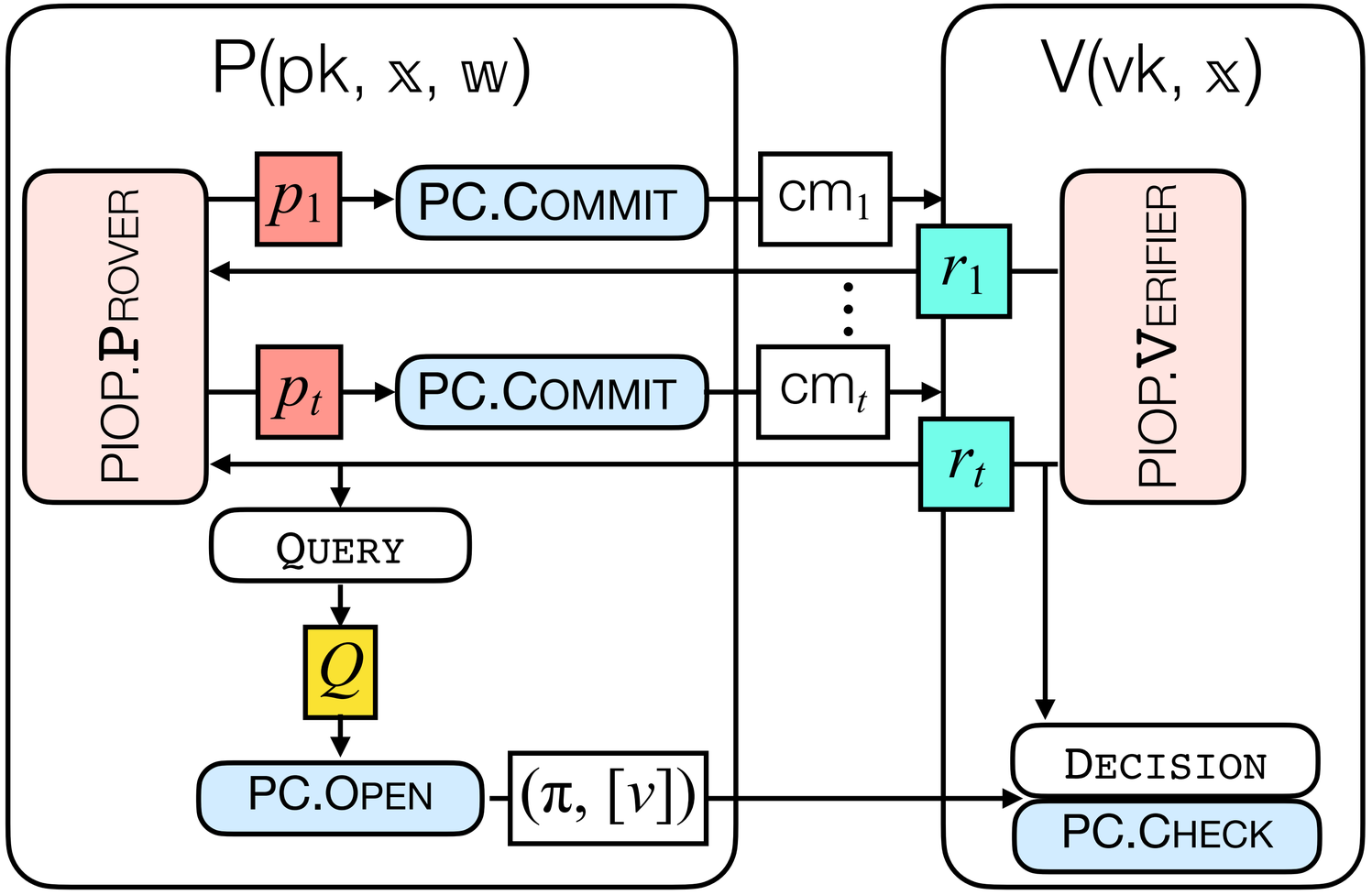


# PIOPs + PC Schemes → SNARK

[CHMMVW20, BFS20]




+ Fiat—Shamir to get non-interactivity

# Prior PIOP-based SNARKs for R1CS

[CHMMVW20, S21]

# Prior PIOP-based SNARKs for R1CS

[CHMMVW20, S21]

R1CS consists of triples $((A, B, C), x, w)$ such that the following holds for $z = (x, w)$:

$Az \circ Bz = Cz$, or equivalently the following checks are satisfied:

# Prior PIOP-based SNARKs for R1CS

[CHMMVW20, S21]

R1CS consists of triples $((A, B, C), x, w)$ such that the following holds for $z = (x, w)$:

$Az \circ Bz = Cz$, or equivalently the following checks are satisfied:

**Linear checks:**

$z_A = Az,\ z_B = Bz,\ z_C = Cz$

# Prior PIOP-based SNARKs for R1CS

[CHMMVW20, S21]

R1CS consists of triples $((A, B, C), x, w)$ such that the following holds for $z = (x, w)$:

$Az \circ Bz = Cz$, or equivalently the following checks are satisfied:

**Linear checks:**

$$z_A = Az,\ z_B = Bz,\ z_C = Cz$$

**Nonlinear "row" checks:**

$$z_A \circ z_B = z_C$$

# Prior PIOP-based SNARKs for R1CS

[CHMMVW20, S21]

R1CS consists of triples $((A, B, C), x, w)$ such that the following holds for $z = (x, w)$:

$Az \circ Bz = Cz$, or equivalently the following checks are satisfied:

**Linear checks:**

$$z_A = Az,\ z_B = Bz,\ z_C = Cz$$

**Nonlinear "row" checks:**

$$z_A \circ z_B = z_C$$

# Prior PIOP-based SNARKs for R1CS

[CHMMVW20, S21]

R1CS consists of triples $((A, B, C), x, w)$ such that the following holds for $z = (x, w)$:

$Az \circ Bz = Cz$, or equivalently the following checks are satisfied:

**Linear checks:**

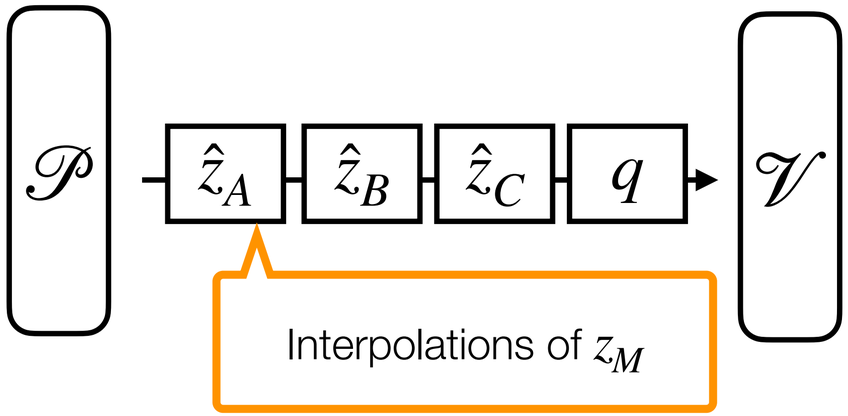
$$z_A = Az,\ z_B = Bz,\ z_C = Cz$$

**Nonlinear "row" checks:**

$$z_A \circ z_B = z_C$$

**Rowcheck subPIOP**

Usually quite cheap!



# Prior PIOP-based SNARKs for R1CS

[CHMMVW20, S21]

R1CS consists of triples $((A, B, C), x, w)$ such that the following holds for $z = (x, w)$:

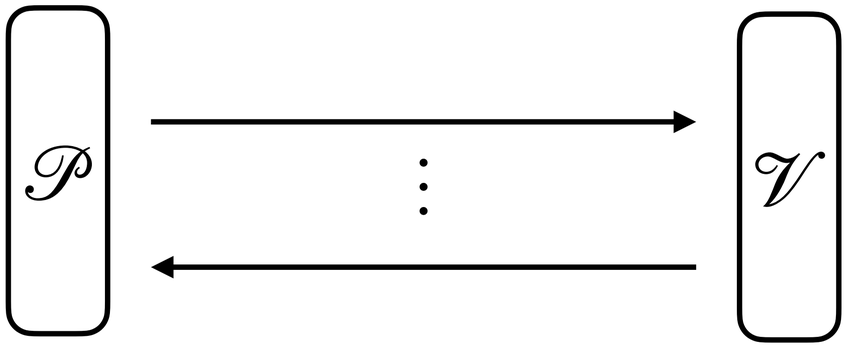
$Az \circ Bz = Cz$, or equivalently the following checks are satisfied:

**Linear checks:**

$z_A = Az,\ z_B = Bz,\ z_C = Cz$

**Lincheck subPIOP**

Usually most expensive part!

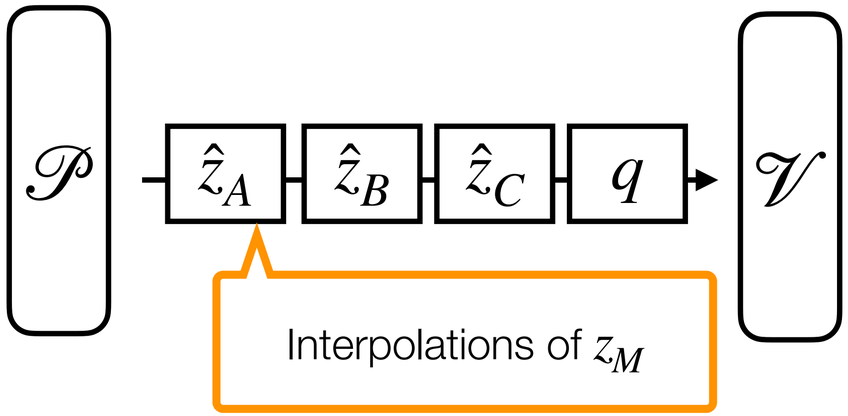



**Nonlinear "row" checks:**

$z_A \circ z_B = z_C$

**Rowcheck subPIOP**

Usually quite cheap!



# Prior PIOP-based SNARKs for R1CS

[CHMMVW20, S21]

R1CS consists of triples $((A, B, C), x, w)$ such that the following holds for $z = (x, w)$:

$Az \circ Bz = Cz$, or equivalently the following checks are satisfied:

**Linear checks:**

$$z_A = Az,\ z_B = Bz,\ z_C = Cz$$

**Lincheck subPIOP**

Usually most expensive part!

Requires numerous commitments, openings, and evaluation proofs

In contrast, circuit-specific SNARKs like Groth16 require no extra group elements

**Nonlinear "row" checks:**

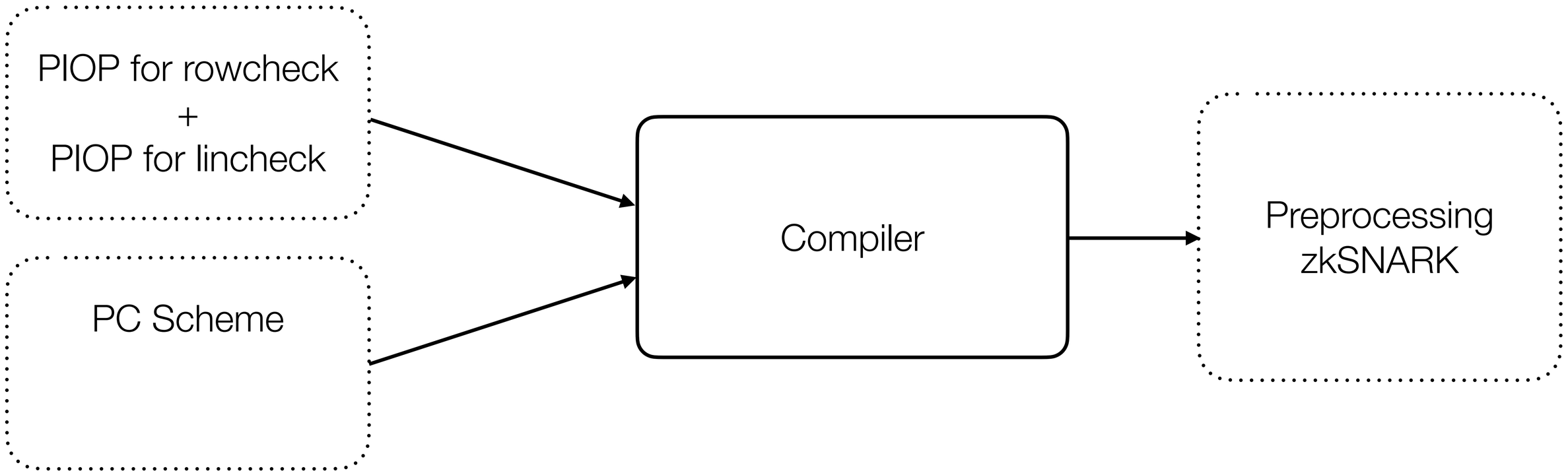
$$z_A \circ z_B = z_C$$

**Rowcheck subPIOP**

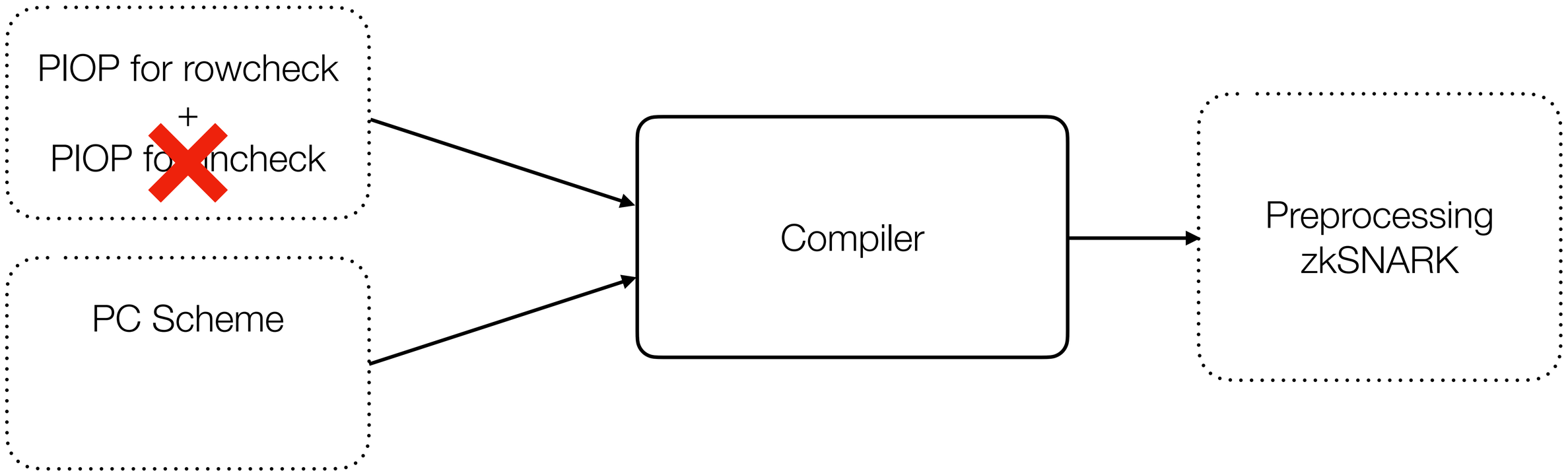
Usually quite cheap!

Compiles to only 4 group elements!

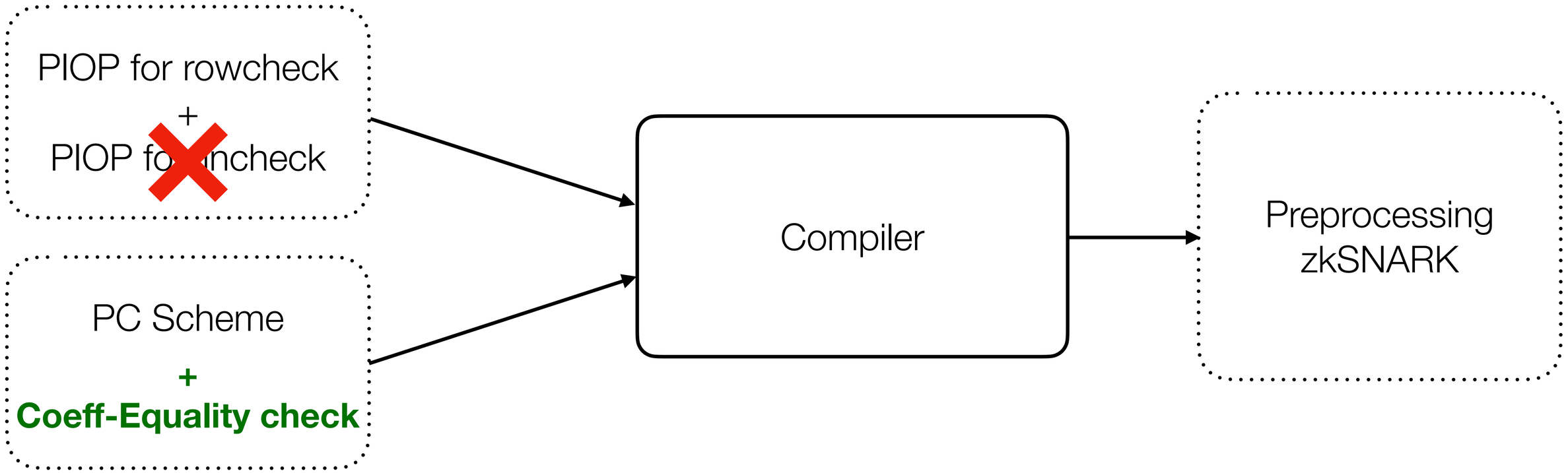
# R1CS SNARKs from PIOPs + EPC Schemes



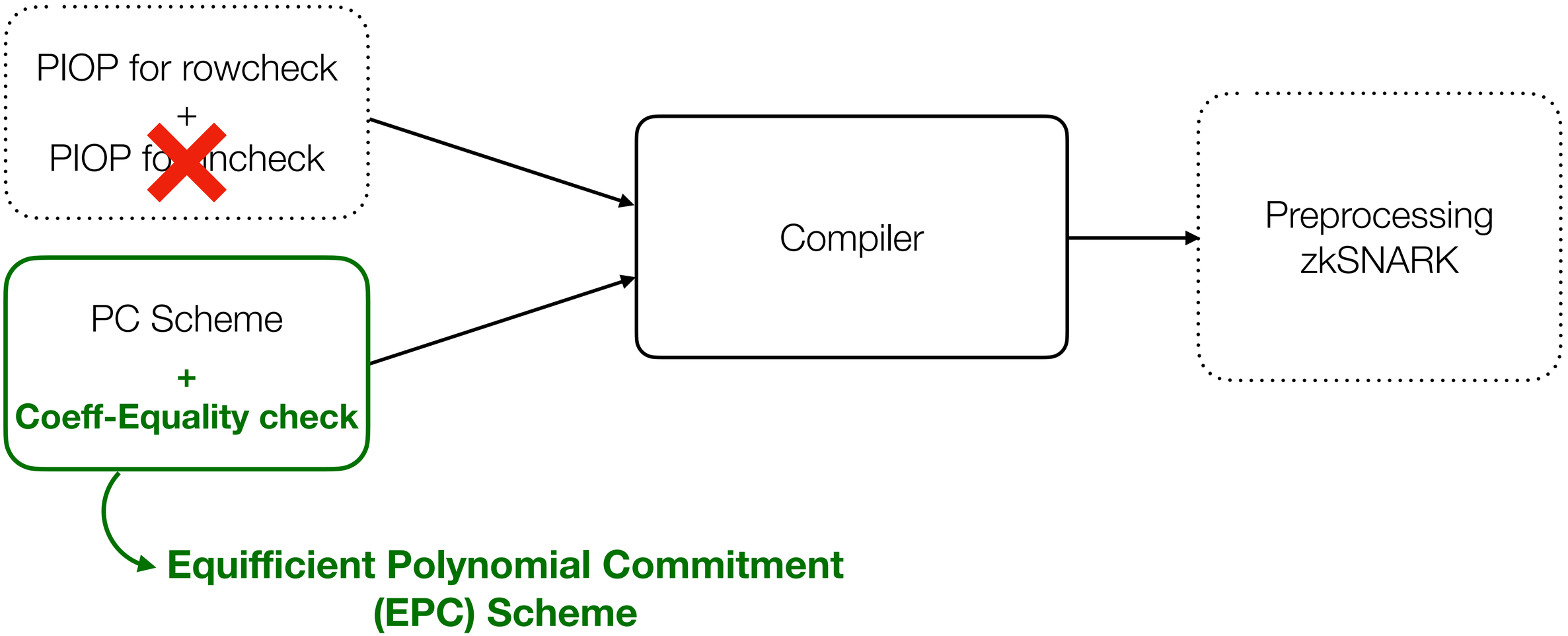
# R1CS SNARKs from PIOPs + EPC Schemes



# R1CS SNARKs from PIOPs + EPC Schemes



# R1CS SNARKs from PIOPs + EPC Schemes



# New Approach for Lincheck

R1CS consists of triples $((A, B, C), x, w)$ such that the following holds for $z = (x, w)$:

$Az \circ Bz = Cz$, or equivalently the following checks are satisfied:

**Linear checks:**

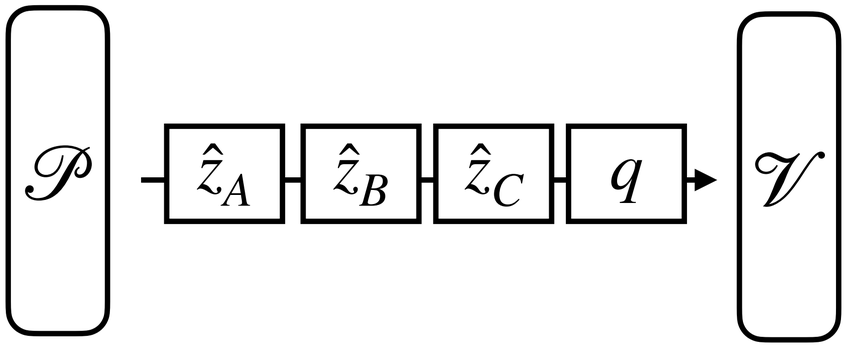
$$z_A = Az,\ z_B = Bz,\ z_C = Cz$$

**Nonlinear "row" checks:**

$$z_A \circ z_B = z_C$$

**Rowcheck subPIOP**

Usually quite cheap!



# New Approach for Lincheck

R1CS consists of triples $((A, B, C), x, w)$ such that the following holds for $z = (x, w)$:

$Az \circ Bz = Cz$, or equivalently the following checks are satisfied:

**Linear checks:**

$$z_A = Az,\ z_B = Bz,\ z_C = Cz$$

**Lincheck via coefficient-equality**

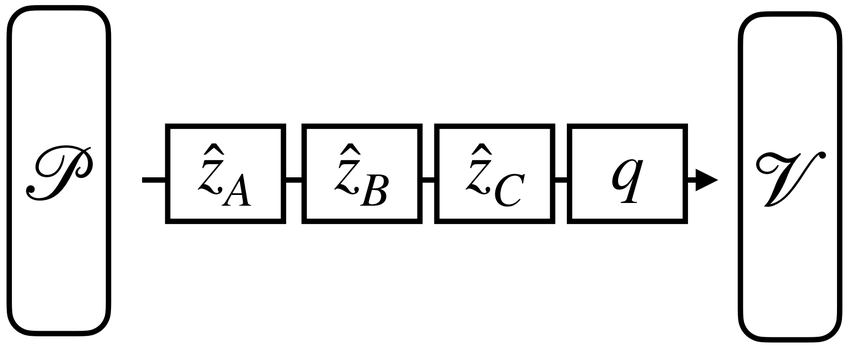
How to enforce?

**Nonlinear "row" checks:**

$$z_A \circ z_B = z_C$$

**Rowcheck subPIOP**

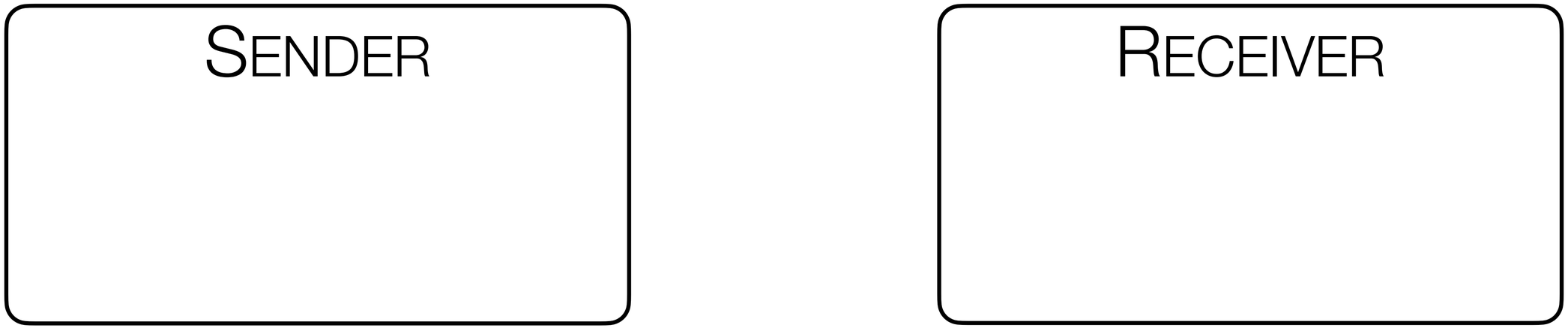
Usually quite cheap!



# New Tool: EPC schemes

# Equifficient PC (EPC) schemes

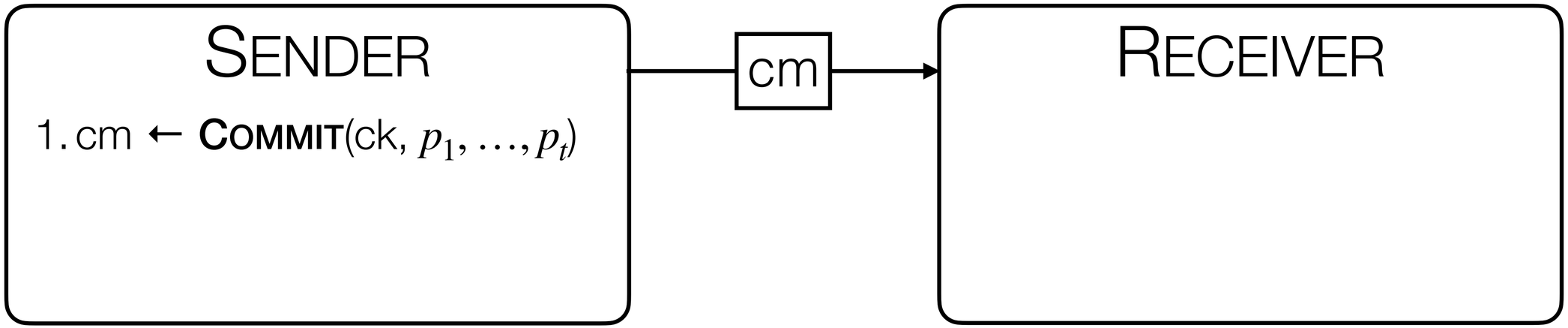
# Equifficient PC (EPC) schemes



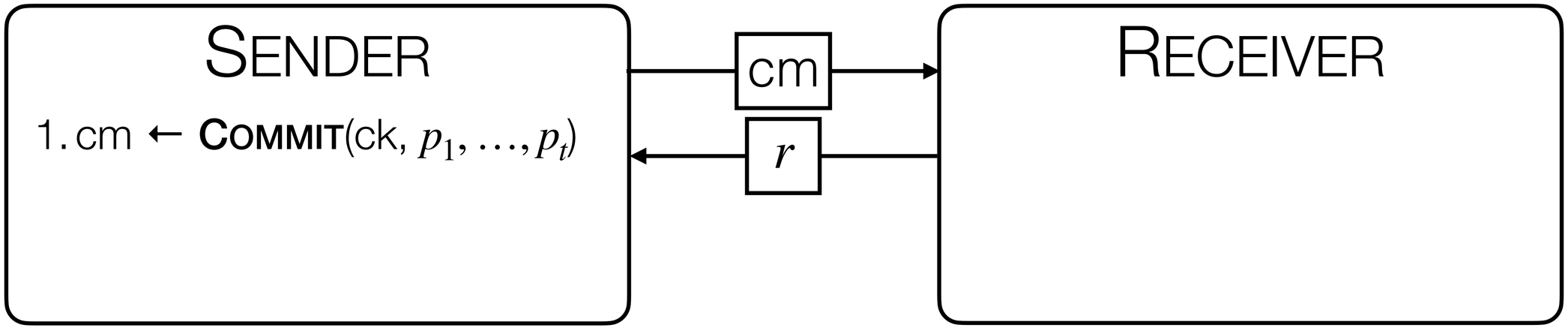
# Equifficient PC (EPC) schemes

| SENDER | RECEIVER |
| --- | --- |
| 1. cm ← **COMMIT**(ck, $p_1, \ldots, p_t$) | |

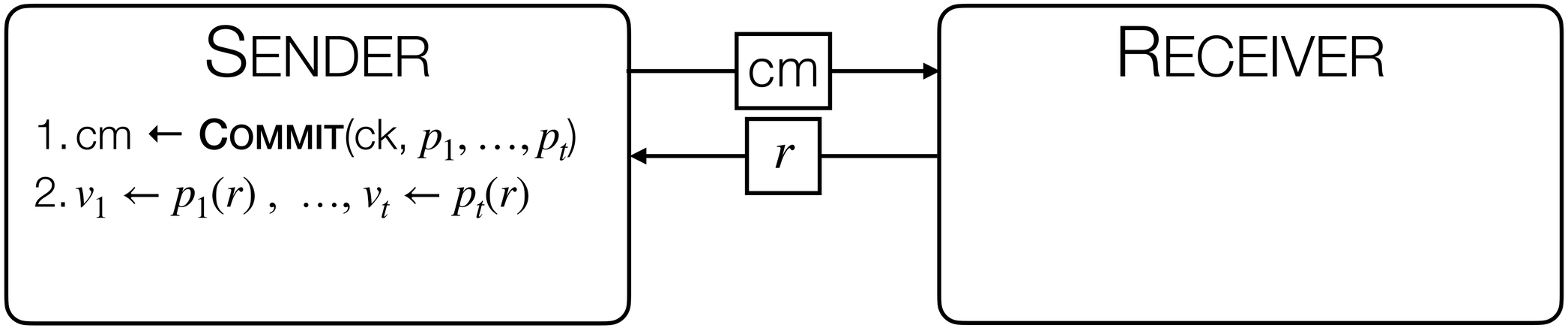
# Equifficient PC (EPC) schemes



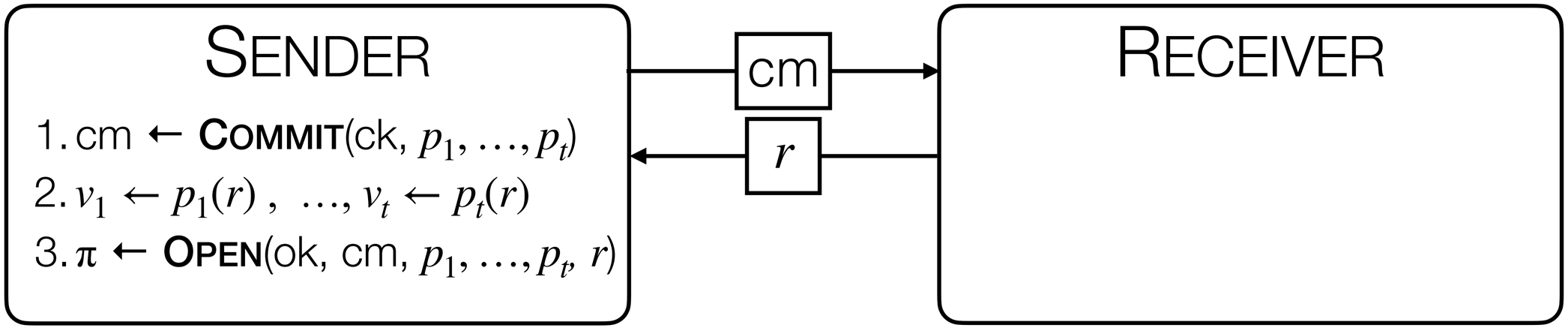
# Equifficient PC (EPC) schemes



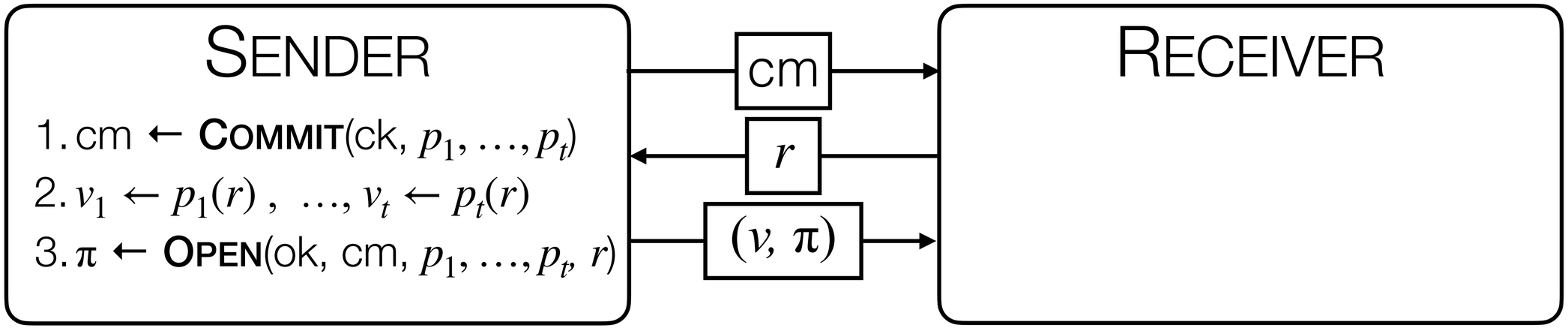
# Equifficient PC (EPC) schemes



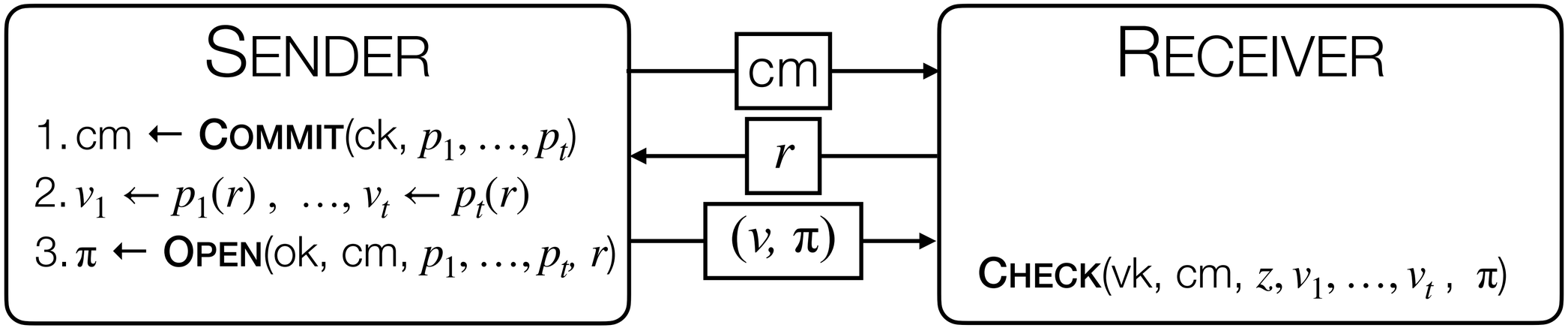
# Equifficient PC (EPC) schemes



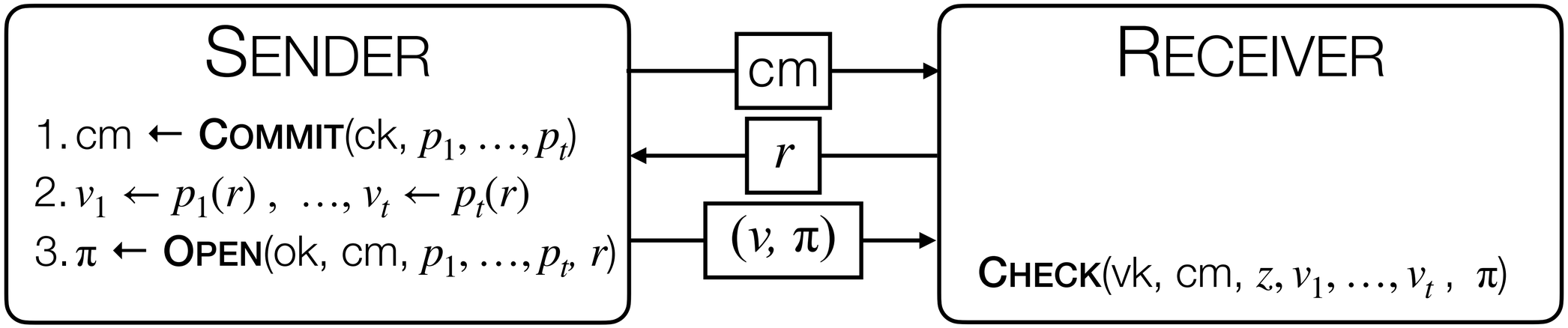
# Equifficient PC (EPC) schemes



# Equifficient PC (EPC) schemes

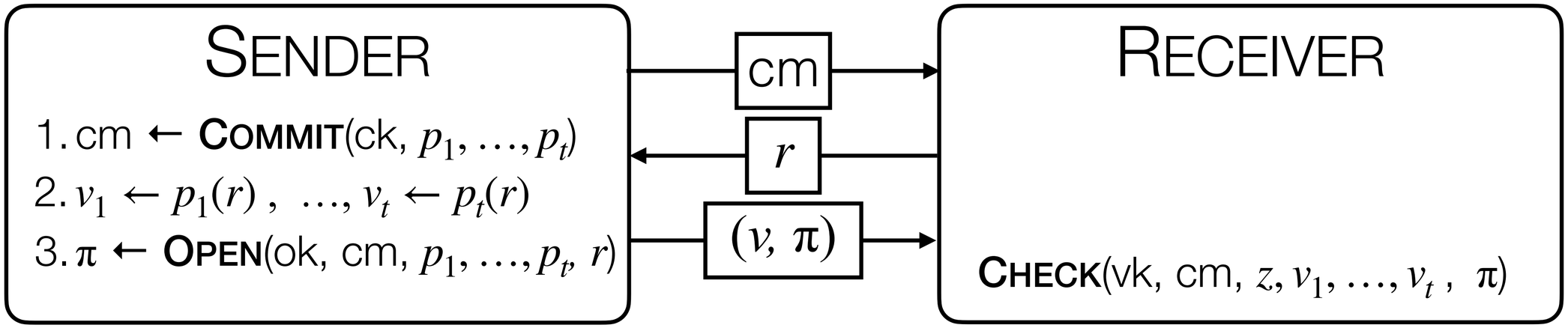


# Equifficient PC (EPC) schemes




**Extractability**

# Equifficient PC (EPC) schemes




## Extractability

If the adversary outputs a commitment & proof that convinces the receiver, then it must know $p_1, \ldots, p_n$ that the following holds:

- **PC Extractability:** $p_1(z) = v_1, \ldots, p_n(z) = v_n$
- **Equifficient constraint satisfaction:** $p_1, \ldots, p_n$ can be represented with equal coefficients

# KZG-based EPC Construction

# KZG-based EPC Construction

**Step 1:** Using regular KZG, commit to the polynomials $\hat{z}_A$, $\hat{z}_B$, and $\hat{z}_C$

# KZG-based EPC Construction

**Step 1:** Using regular KZG, commit to the polynomials $\hat{z}_A$, $\hat{z}_B$, and $\hat{z}_C$

$$\mathsf{ck} = (1 \cdot G,\ \tau \cdot G,\ \tau^2 \cdot G, \ldots,\ \tau^{n-1} \cdot G)$$

# KZG-based EPC Construction

**Step 1:** Using regular KZG, commit to the polynomials $\hat{z}_A$, $\hat{z}_B$, and $\hat{z}_C$

$$\mathsf{ck} = (1 \cdot G,\ \tau \cdot G,\ \tau^2 \cdot G,\ \ldots,\ \tau^{n-1} \cdot G)$$

$$\begin{aligned}
\mathsf{KZG.Commit}(\mathsf{ck}, \hat{z}_A) &\to c_A \\
\mathsf{KZG.Commit}(\mathsf{ck}, \hat{z}_B) &\to c_B \\
\mathsf{KZG.Commit}(\mathsf{ck}, \hat{z}_C) &\to c_C
\end{aligned}$$

where $c_M := \sum_i z_M[i] \cdot \tau^i \cdot G$ for $M \in \{A, B, C\}$

# KZG-based EPC Construction

# KZG-based EPC Construction

**Step 2:** Enforce the coefficient-equality constraint.

# KZG-based EPC Construction

**Step 2:** Enforce the coefficient-equality constraint.

To do this, first we construct committer keys that encode each basis…

$$E := \{\mathscr{A}, \mathscr{B}, \mathscr{C}\}$$

# KZG-based EPC Construction

**Step 2:** Enforce the coefficient-equality constraint.

To do this, first we construct committer keys that encode each basis…

$$E := \{\mathscr{A}, \mathscr{B}, \mathscr{C}\}$$

$$\mathscr{A} = \{\hat{a}_1, \ldots, \hat{a}_n\} \iff \mathsf{ck}_A = \left[\hat{a}_1(\tau)G,\ \hat{a}_2(\tau)G,\ \hat{a}_3(\tau)G,\ \ldots,\ \hat{a}_n(\tau)G\right]$$

$$\mathscr{B} = \{\hat{b}_1, \ldots, \hat{b}_n\} \iff \mathsf{ck}_B = \left[\hat{b}_1(\tau)G,\ \hat{b}_2(\tau)G,\ \hat{b}_3(\tau)G,\ \ldots,\ \hat{b}_n(\tau)G\right]$$

$$\mathscr{C} = \{\hat{c}_1, \ldots, \hat{c}_n\} \iff \mathsf{ck}_C = \left[\hat{c}_1(\tau)G,\ \hat{c}_2(\tau)G,\ \hat{c}_3(\tau)G,\ \ldots,\ \hat{c}_n(\tau)G\right]$$

# KZG-based EPC Construction

# KZG-based EPC Construction

**Step 3:** Next, we take a random linear combination of these committer keys to get the “consistency” committer key!

# KZG-based EPC Construction

**Step 3:** Next, we take a random linear combination of these committer keys to get the "consistency" committer key!

$$
\begin{aligned}
\mathsf{ck}^* = \quad & \alpha \cdot \mathsf{ck}_A + \\
& \beta \cdot \mathsf{ck}_B + \\
& \gamma \cdot \mathsf{ck}_C
\end{aligned}
=
\begin{aligned}
& \alpha \cdot \left[ \hat{a}_1(\tau)\, G, \quad \hat{a}_2(\tau)\, G, \ \ldots \quad \hat{a}_n(\tau)\, G \right] + \\
& \beta \cdot \left[ \hat{b}_1(\tau)\, G, \quad \hat{b}_2(\tau)\, G, \ \ldots \quad \hat{b}_n(\tau)\, G \right] + \\
& \gamma \cdot \left[ \hat{c}_1(\tau)\, G, \quad \hat{c}_2(\tau)\, G, \ \ldots \quad \hat{c}_n(\tau)\, G \right]
\end{aligned}
$$

# KZG-based EPC Construction

**Step 3:** Next, we take a random linear combination of these committer keys to get the "consistency" committer key!

$$
\begin{aligned}
\mathsf{ck}^* = \quad & \alpha \cdot \mathsf{ck}_A + \\
& \beta \cdot \mathsf{ck}_B + \\
& \gamma \cdot \mathsf{ck}_C
\end{aligned}
=
\begin{aligned}
& \alpha \cdot \left[ \hat{a}_1(\tau)\, G, \quad \hat{a}_2(\tau)\, G, \ \ldots \quad \hat{a}_n(\tau)\, G \right] + \\
& \beta \cdot \left[ \hat{b}_1(\tau)\, G, \quad \hat{b}_2(\tau)\, G, \ \ldots \quad \hat{b}_n(\tau)\, G \right] + \\
& \gamma \cdot \left[ \hat{c}_1(\tau)\, G, \quad \hat{c}_2(\tau)\, G, \ \ldots \quad \hat{c}_n(\tau)\, G \right]
\end{aligned}
$$

Now, commit to $z$, which recall is the equal-coefficient representation of $\hat{z}_A$, $\hat{z}_B$, and $\hat{z}_C$

# KZG-based EPC Construction

**Step 3:** Next, we take a random linear combination of these committer keys to get the "consistency" committer key!

$$\begin{aligned} \mathsf{ck}^* = \quad & \alpha \cdot \mathsf{ck}_A + \\ & \beta \cdot \mathsf{ck}_B + \\ & \gamma \cdot \mathsf{ck}_C \end{aligned} = \begin{aligned} & \alpha \cdot \left[ \hat{a}_1(\tau)\, G, \quad \hat{a}_2(\tau)\, G, \ \ldots \quad \hat{a}_n(\tau)\, G \right] + \\ & \beta \cdot \left[ \hat{b}_1(\tau)\, G, \quad \hat{b}_2(\tau)\, G, \ \ldots \quad \hat{b}_n(\tau)\, G \right] + \\ & \gamma \cdot \left[ \hat{c}_1(\tau)\, G, \quad \hat{c}_2(\tau)\, G, \ \ldots \quad \hat{c}_n(\tau)\, G \right] \end{aligned}$$

Now, commit to $z$, which recall is the equal-coefficient representation of $\hat{z}_A$, $\hat{z}_B$, and $\hat{z}_C$

$$\begin{aligned} c^* = \langle z, \mathsf{ck}^* \rangle = \quad & \alpha \cdot \left( z_1 \cdot \hat{a}_1(\tau) + z_2 \cdot \hat{a}_2(\tau) + \ldots + z_n \cdot \hat{a}_n(\tau) \right) \cdot G + \\ & \beta \cdot \left( z_1 \cdot \hat{b}_1(\tau) + z_2 \cdot \hat{b}_2(\tau) + \ldots + z_n \cdot \hat{b}_n(\tau) \right) \cdot G + \\ & \gamma \cdot \left( z_1 \cdot \hat{c}_1(\tau) + z_2 \cdot \hat{c}_2(\tau) + \ldots + z_n \cdot \hat{c}_n(\tau) \right) \cdot G \end{aligned}$$

# KZG-based EPC Construction

**Step 3:** Next, we take a random linear combination of these committer keys to get the "consistency" committer key!

$$
\begin{aligned}
\mathsf{ck}^* = \quad & \alpha \cdot \mathsf{ck}_A + \\
& \beta \cdot \mathsf{ck}_B + \\
& \gamma \cdot \mathsf{ck}_C
\end{aligned}
=
\begin{aligned}
& \alpha \cdot \left[ \hat{a}_1(\tau)\, G, \quad \hat{a}_2(\tau)\, G, \ \ldots \quad \hat{a}_n(\tau)\, G \right] + \\
& \beta \cdot \left[ \hat{b}_1(\tau)\, G, \quad \hat{b}_2(\tau)\, G, \ \ldots \quad \hat{b}_n(\tau)\, G \right] + \\
& \gamma \cdot \left[ \hat{c}_1(\tau)\, G, \quad \hat{c}_2(\tau)\, G, \ \ldots \quad \hat{c}_n(\tau)\, G \right]
\end{aligned}
$$

Now, commit to $z$, which recall is the equal-coefficient representation of $\hat{z}_A$, $\hat{z}_B$, and $\hat{z}_C$

$$
\begin{aligned}
c^* = \langle z, \mathsf{ck}^* \rangle = \quad & \alpha \cdot \left( z_1 \cdot \hat{a}_1(\tau) + z_2 \cdot \hat{a}_2(\tau) + \ldots + z_n \cdot \hat{a}_n(\tau) \right) \cdot G + \\
& \beta \cdot \left( z_1 \cdot \hat{b}_1(\tau) + z_2 \cdot \hat{b}_2(\tau) + \ldots + z_n \cdot \hat{b}_n(\tau) \right) \cdot G + \\
& \gamma \cdot \left( z_1 \cdot \hat{c}_1(\tau) + z_2 \cdot \hat{c}_2(\tau) + \ldots + z_n \cdot \hat{c}_n(\tau) \right) \cdot G
\end{aligned}
$$

# KZG-based EPC Construction

# KZG-based EPC Construction

$$\text{Commit}(\text{ck}, (z_A(X), z_B(X), z_C(X))) \rightarrow c_A, c_B, c_C \, , \, c^*$$

$$c_A = z_A(\tau) \cdot G, \quad c_B = z_B(\tau) \cdot G \, , c_C = z_C(\tau) \cdot G$$

**Consistency Commitment**

$$c^* = ( \alpha \cdot \hat{z}_A(\tau) + \beta \cdot \hat{z}_B(\tau) + \gamma \cdot \hat{z}_C(\tau) ) \cdot G$$

# KZG-based EPC Construction

Equifficient

$$\text{Commit}(\text{ck}, (z_A(X), z_B(X), z_C(X))) \rightarrow c_A, c_B, c_C \, , c^*$$

$$c_A = z_A(\tau) \cdot G, \quad c_B = z_B(\tau) \cdot G \, , c_C = z_C(\tau) \cdot G$$

**Consistency Commitment**

$$c^* = ( \alpha \cdot \hat{z}_A(\tau) + \beta \cdot \hat{z}_B(\tau) + \gamma \cdot \hat{z}_C(\tau) ) \cdot G$$

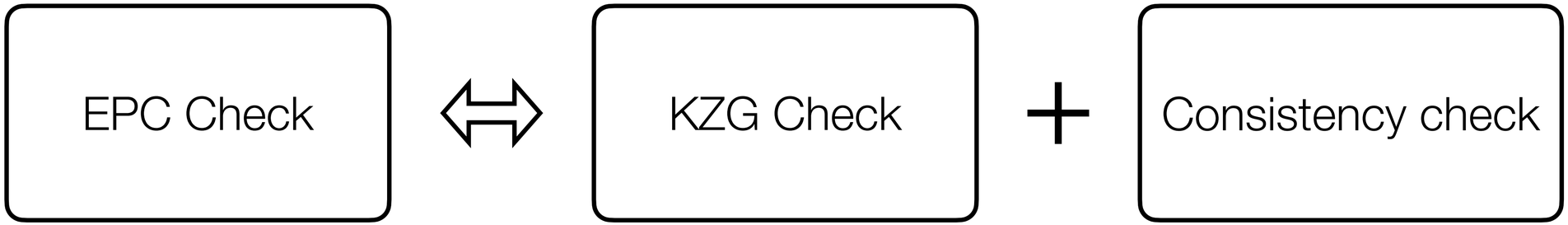
# KZG-based EPC Construction

# KZG-based EPC Construction

**Step 4:** Now, in the EPC check, do regular KZG verifications for each of $c_A, c_B$ and $c_C$ plus a consistency check using our new commitment $c*$

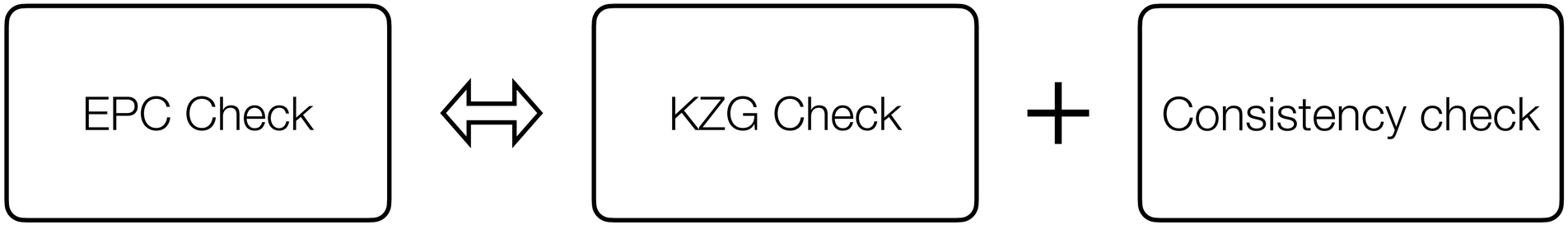
# KZG-based EPC Construction

**Step 4:** Now, in the EPC check, do regular KZG verifications for each of $c_A, c_B$ and $c_C$ plus a consistency check using our new commitment $c*$



# KZG-based EPC Construction

**Step 4:** Now, in the EPC check, do regular KZG verifications for each of $c_A, c_B$ and $c_C$ plus a consistency check using our new commitment $c*$




KZG Check

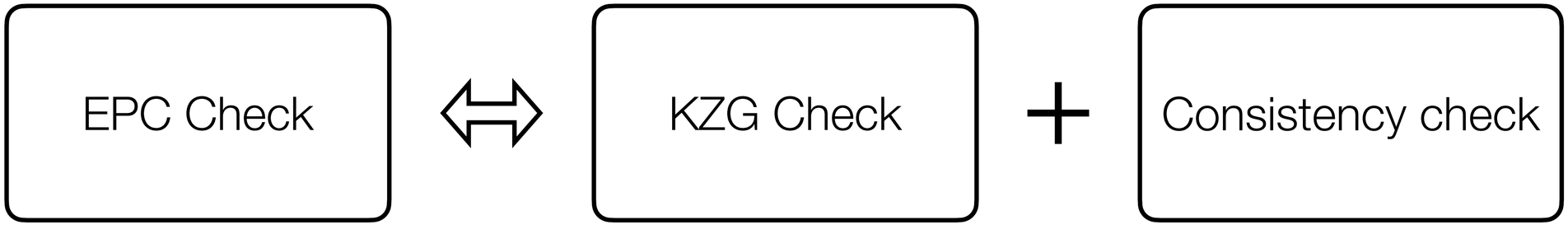
Pass/fail← **KZG.CHECK**(vk, $c_A, v_A, \pi_A$ )

Pass/fail← **KZG.CHECK**(vk, $c_B, v_B, \pi_B$)

Pass/fail ← **KZG.CHECK**(vk, $c_C, v_C, \pi_C$)

# KZG-based EPC Construction

**Step 4:** Now, in the EPC check, do regular KZG verifications for each of $c_A, c_B$ and $c_C$ plus a consistency check using our new commitment $c*$




KZG Check

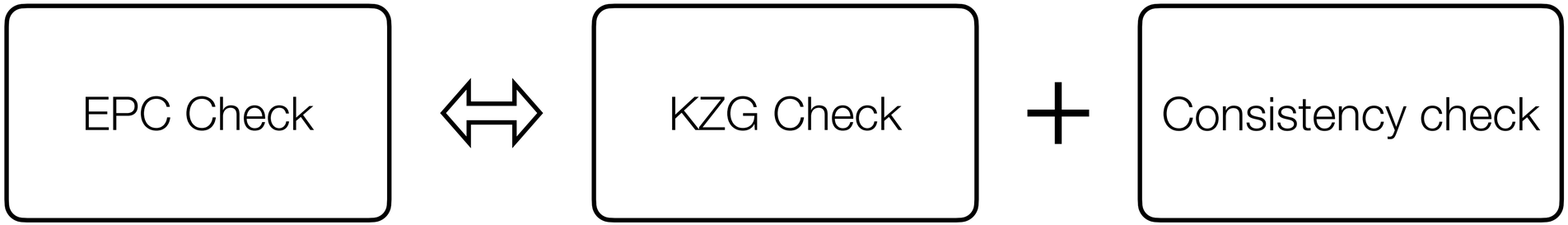
Pass/fail ← **KZG.CHECK**(vk, $c_A, v_A, \pi_A$ )

Pass/fail ← **KZG.CHECK**(vk, $c_B, v_B, \pi_B$)

Pass/fail ← **KZG.CHECK**(vk, $c_C, v_C, \pi_C$)

# KZG-based EPC Construction

**Step 4:** Now, in the EPC check, do regular KZG verifications for each of $c_A, c_B$ and $c_C$ plus a consistency check using our new commitment $c*$




KZG Check

Pass/fail ← **KZG.CHECK**(vk, $c_A, v_A, \pi_A$ )

Pass/fail ← **KZG.CHECK**(vk, $c_B, v_B, \pi_B$)

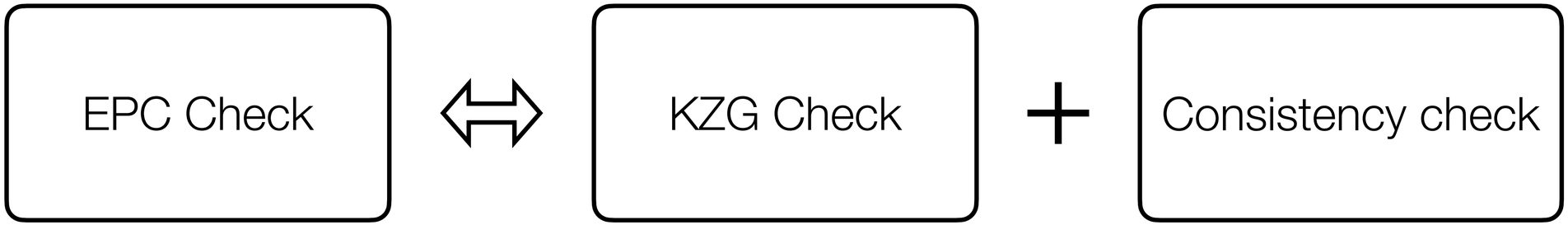
Pass/fail ← **KZG.CHECK**(vk, $c_C, v_C, \pi_C$)

Consistency check

$$e(c^*, H) \stackrel{?}{=} \prod_{M \in \{A,B,C\}} e(c_M, \mathsf{vk}_M)$$

# KZG-based EPC Construction

**Step 4:** Now, in the EPC check, do regular KZG verifications for each of $c_A, c_B$ and $c_C$ plus a consistency check using our new commitment $c*$




KZG Check

Pass/fail ← **KZG.CHECK**(vk, $c_A, v_A, \pi_A$ )

Pass/fail ← **KZG.CHECK**(vk, $c_B, v_B, \pi_B$)

Pass/fail ← **KZG.CHECK**(vk, $c_C, v_C, \pi_C$)

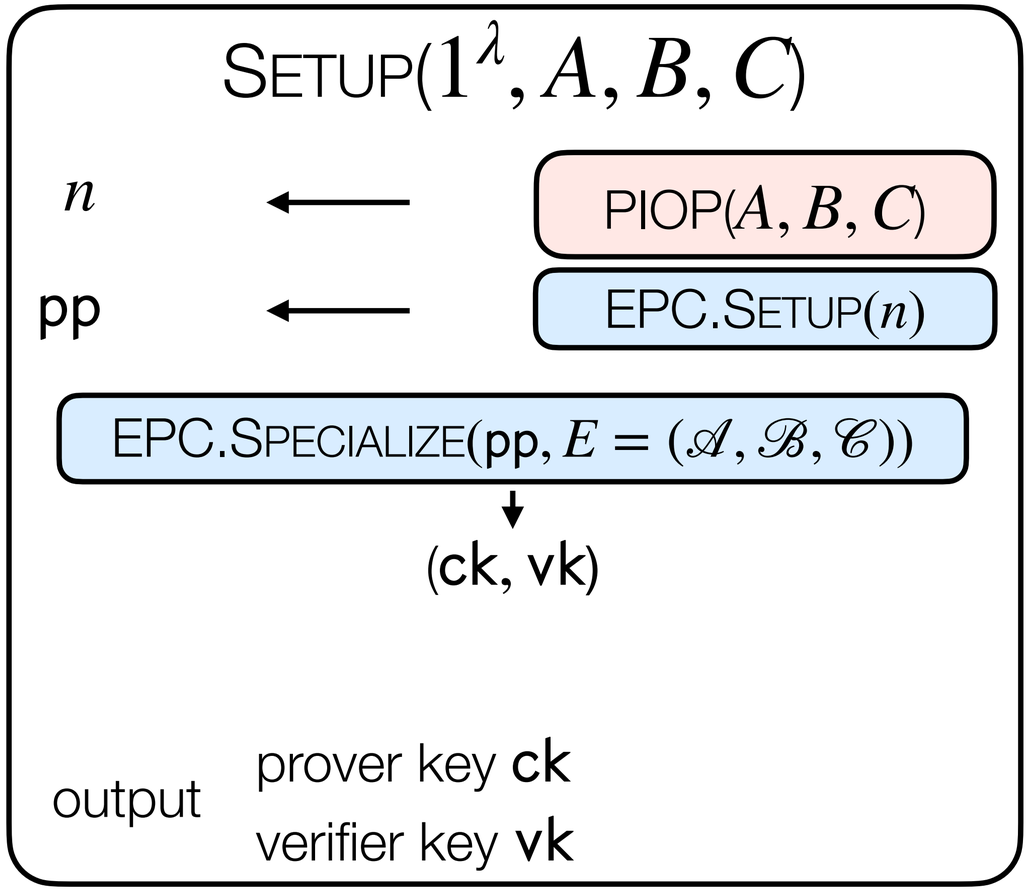
These proofs are computed by KZG.Open, which we omit for simplicity!

Consistency check

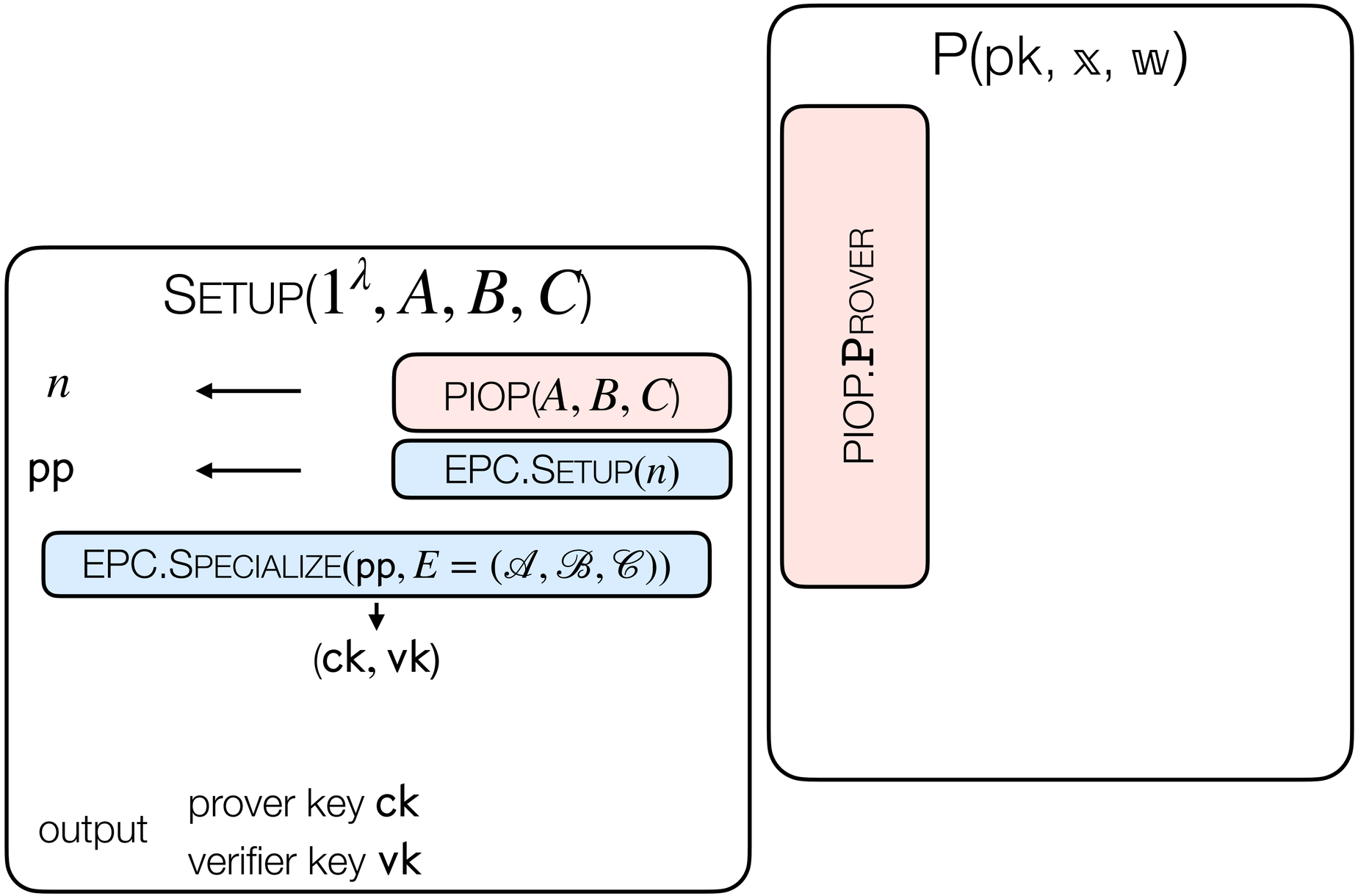
$$e(c^*, H) \stackrel{?}{=} \prod_{M \in \{A,B,C\}} e(c_M, \mathsf{vk}_M)$$

# Our SNARK Construction

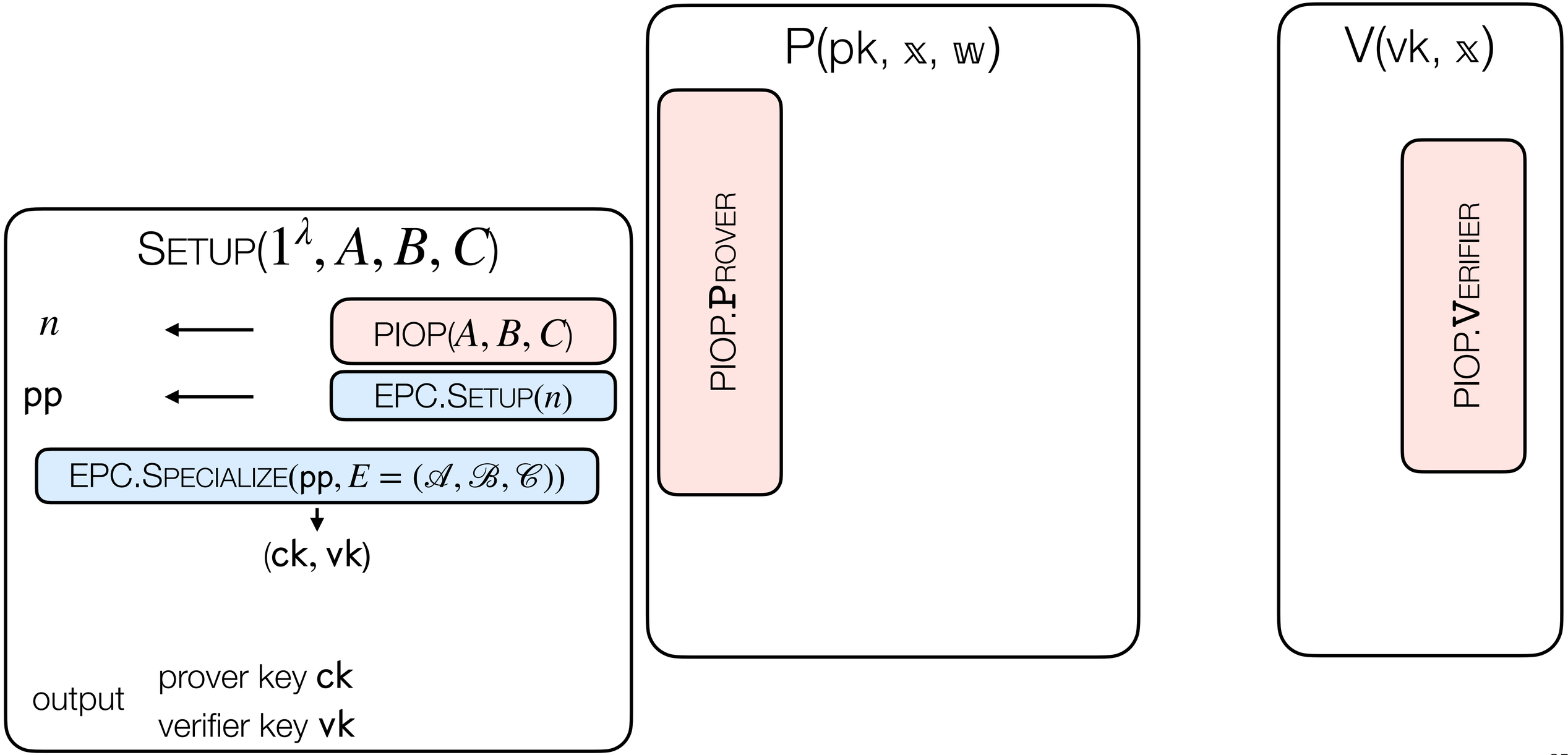
# PIOPs + EPC Schemes → SNARK



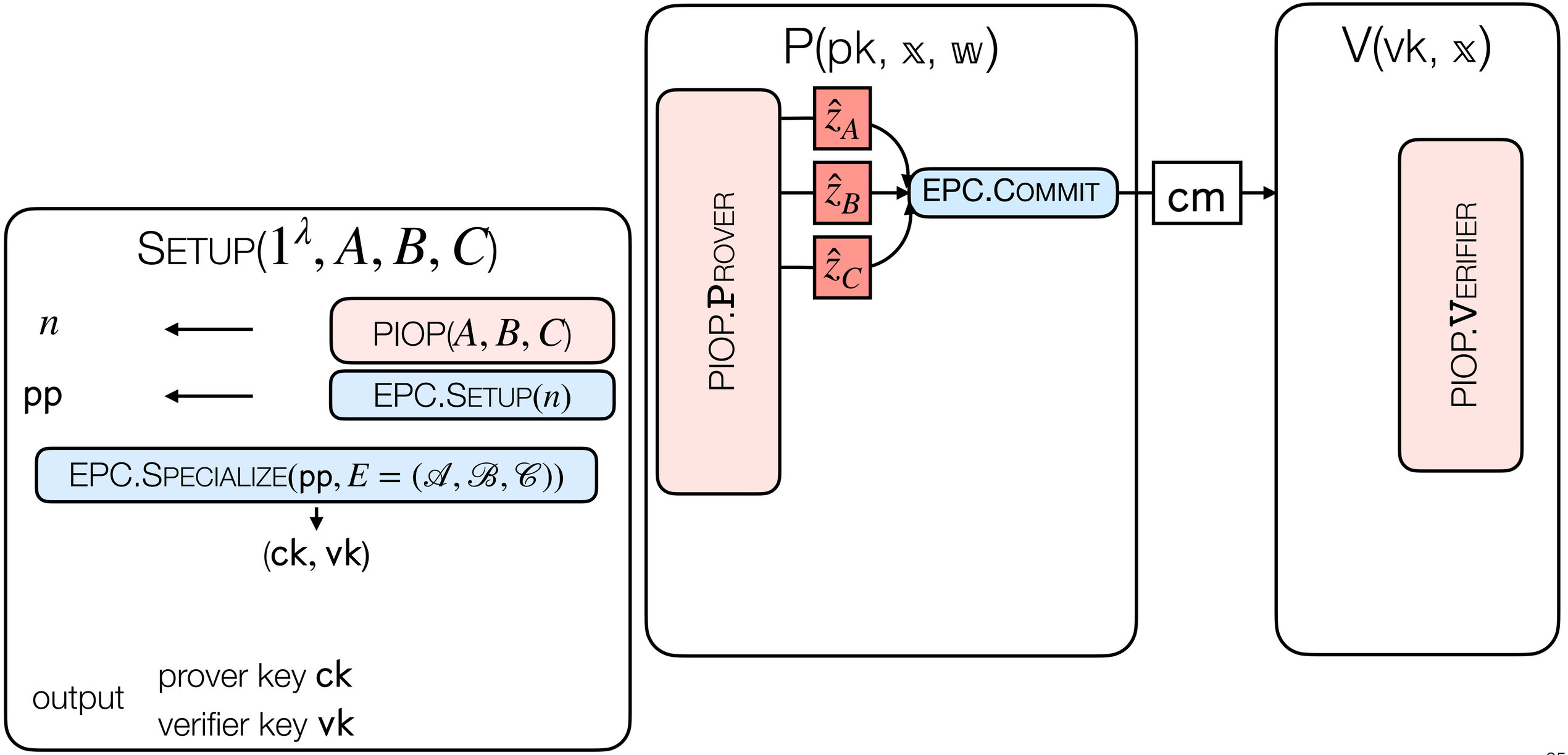
# PIOPs + EPC Schemes → SNARK



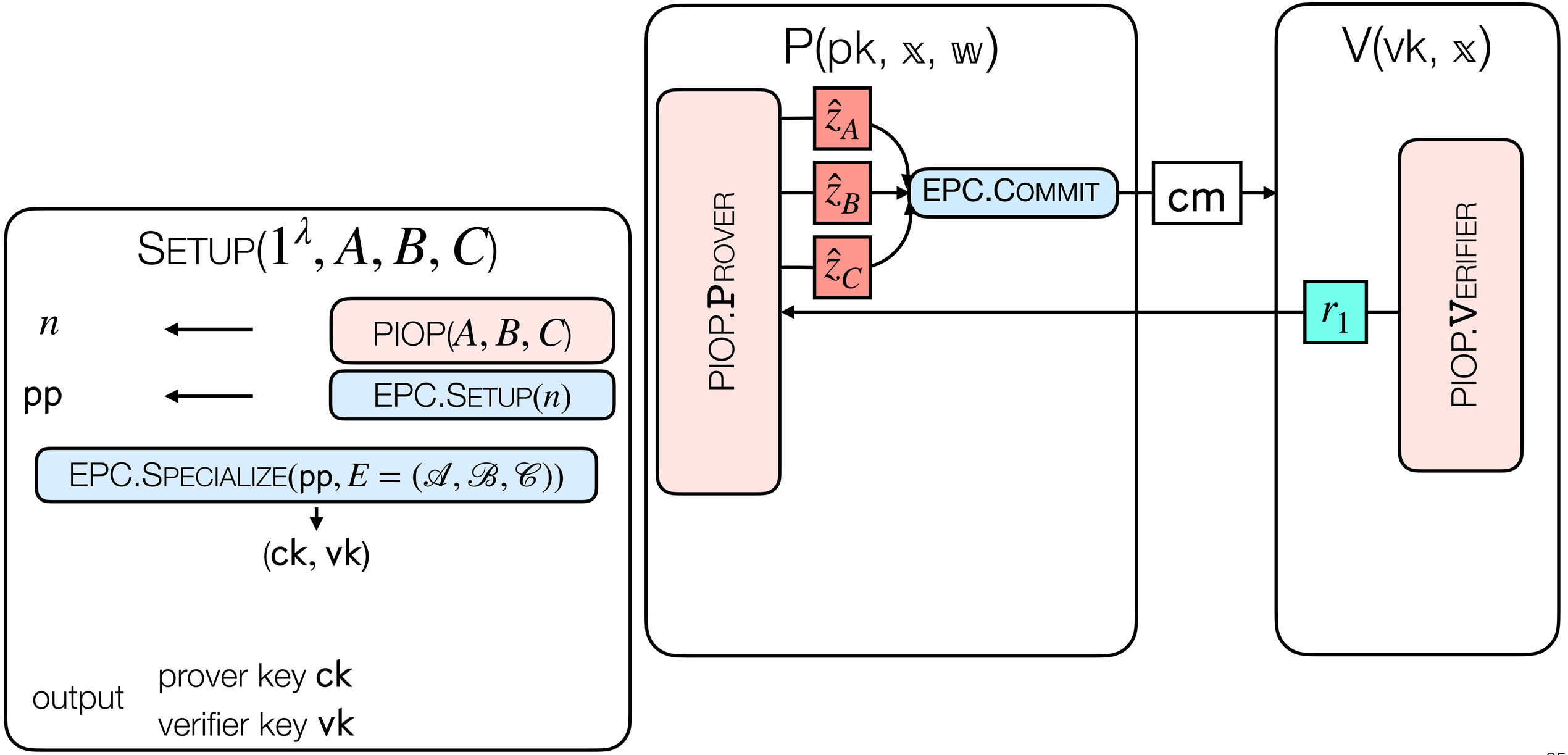
# PIOPs + EPC Schemes → SNARK



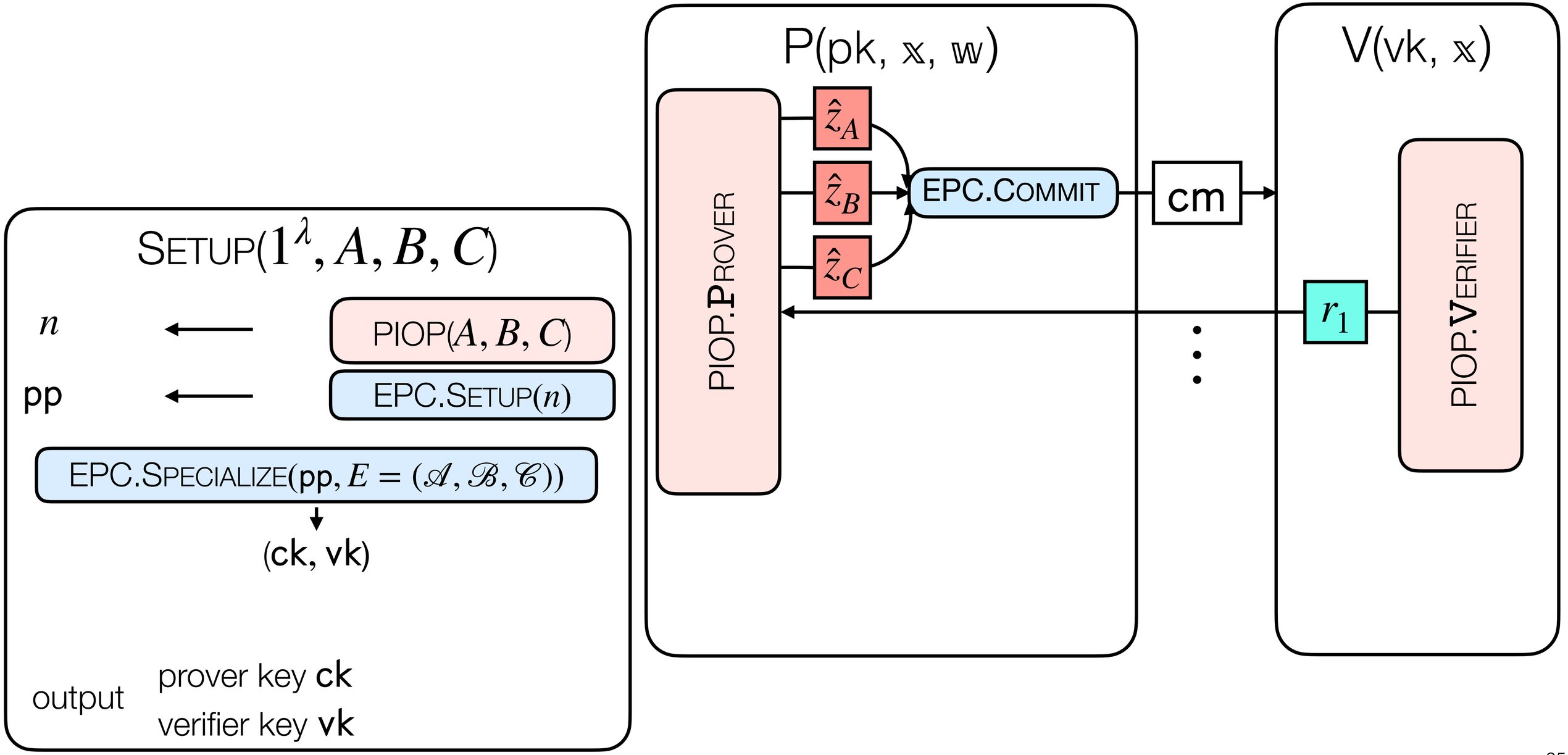
# PIOPs + EPC Schemes → SNARK



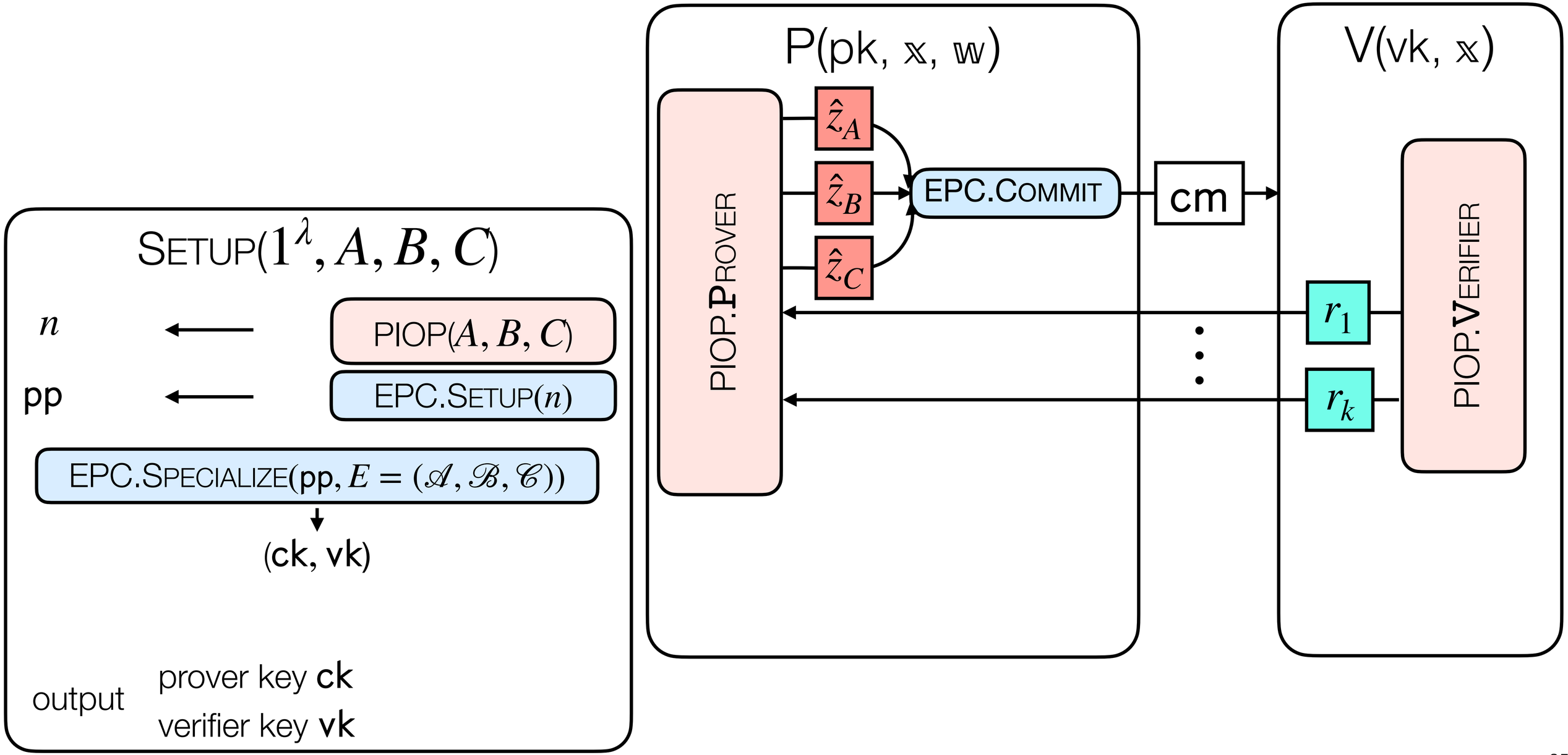
# PIOPs + EPC Schemes → SNARK



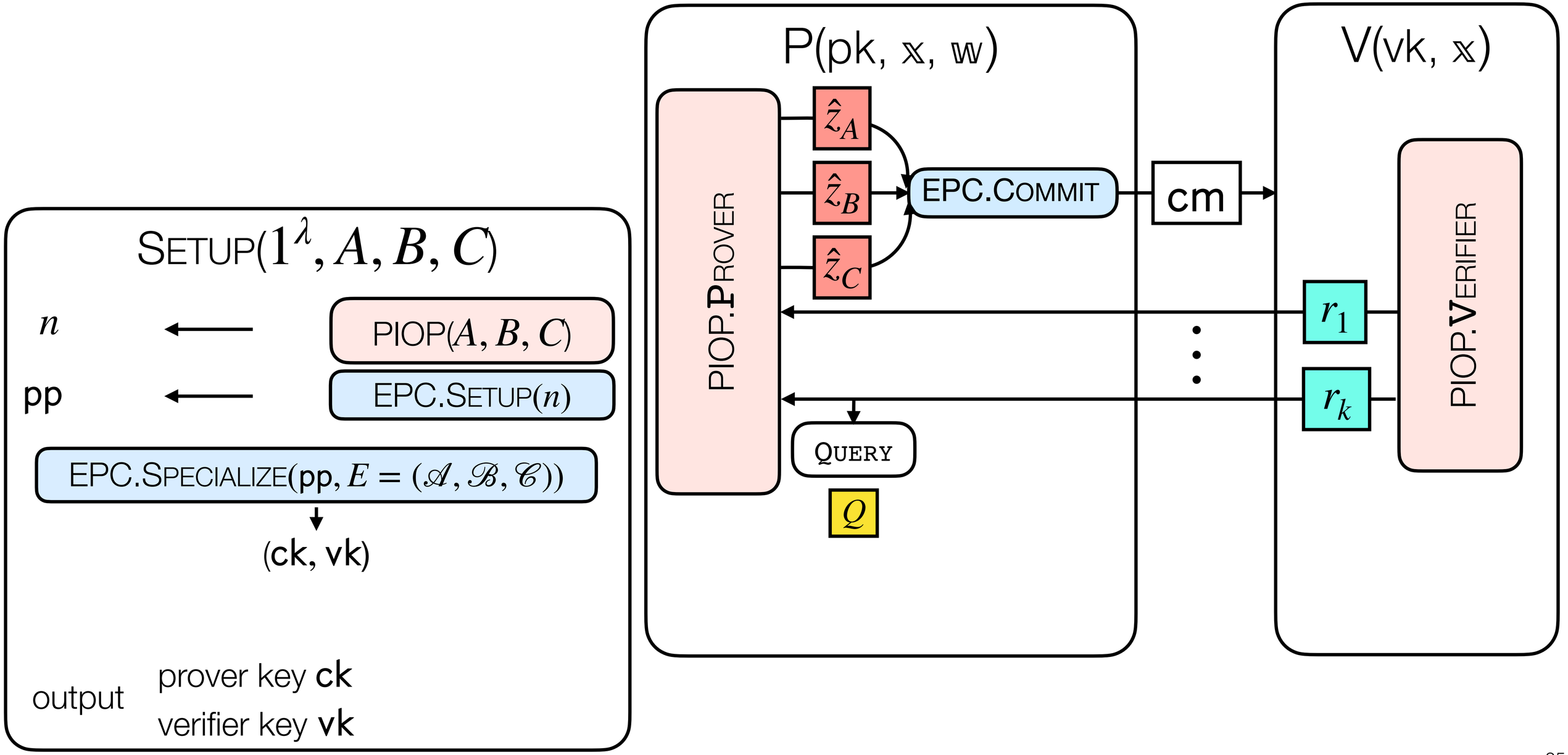
# PIOPs + EPC Schemes → SNARK



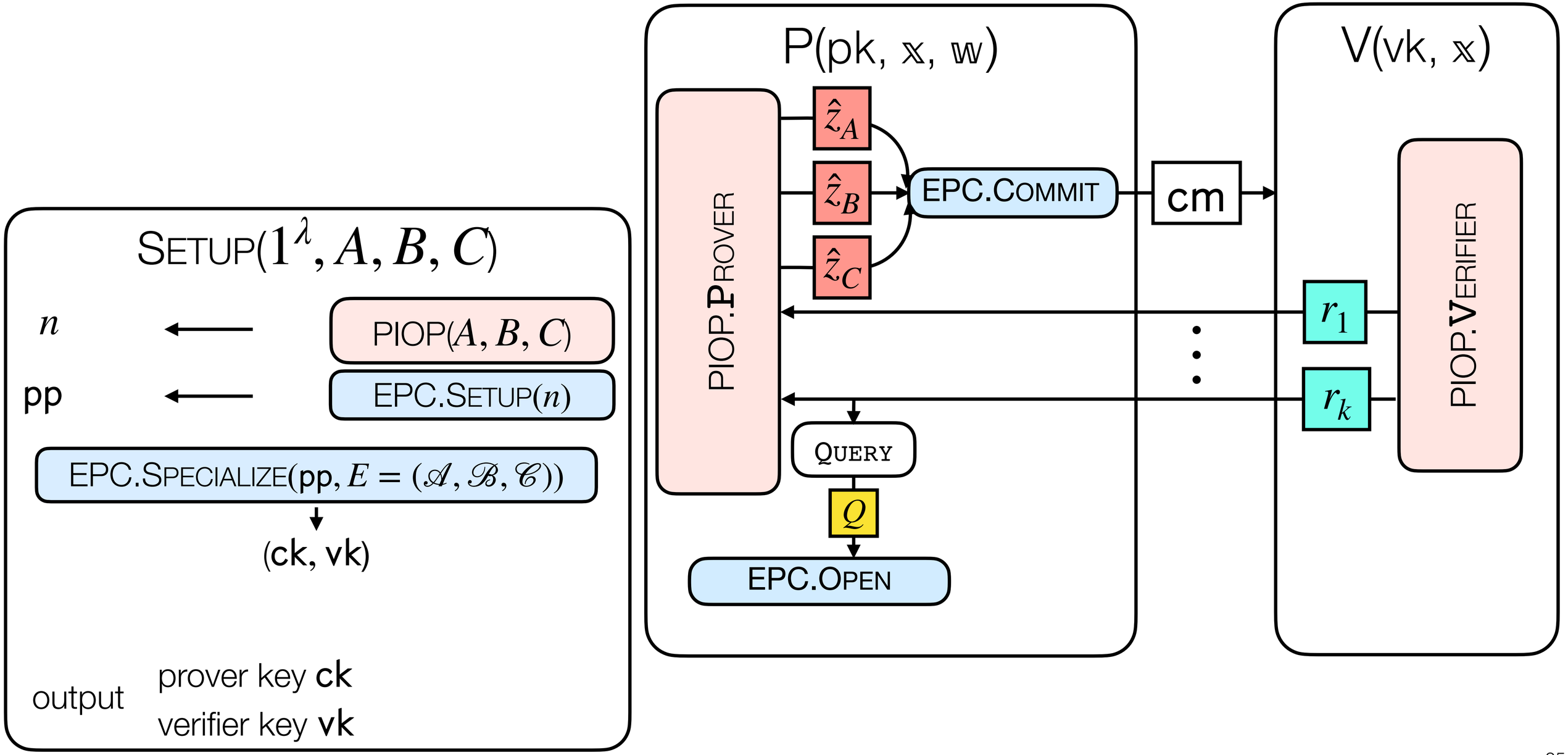
# PIOPs + EPC Schemes → SNARK



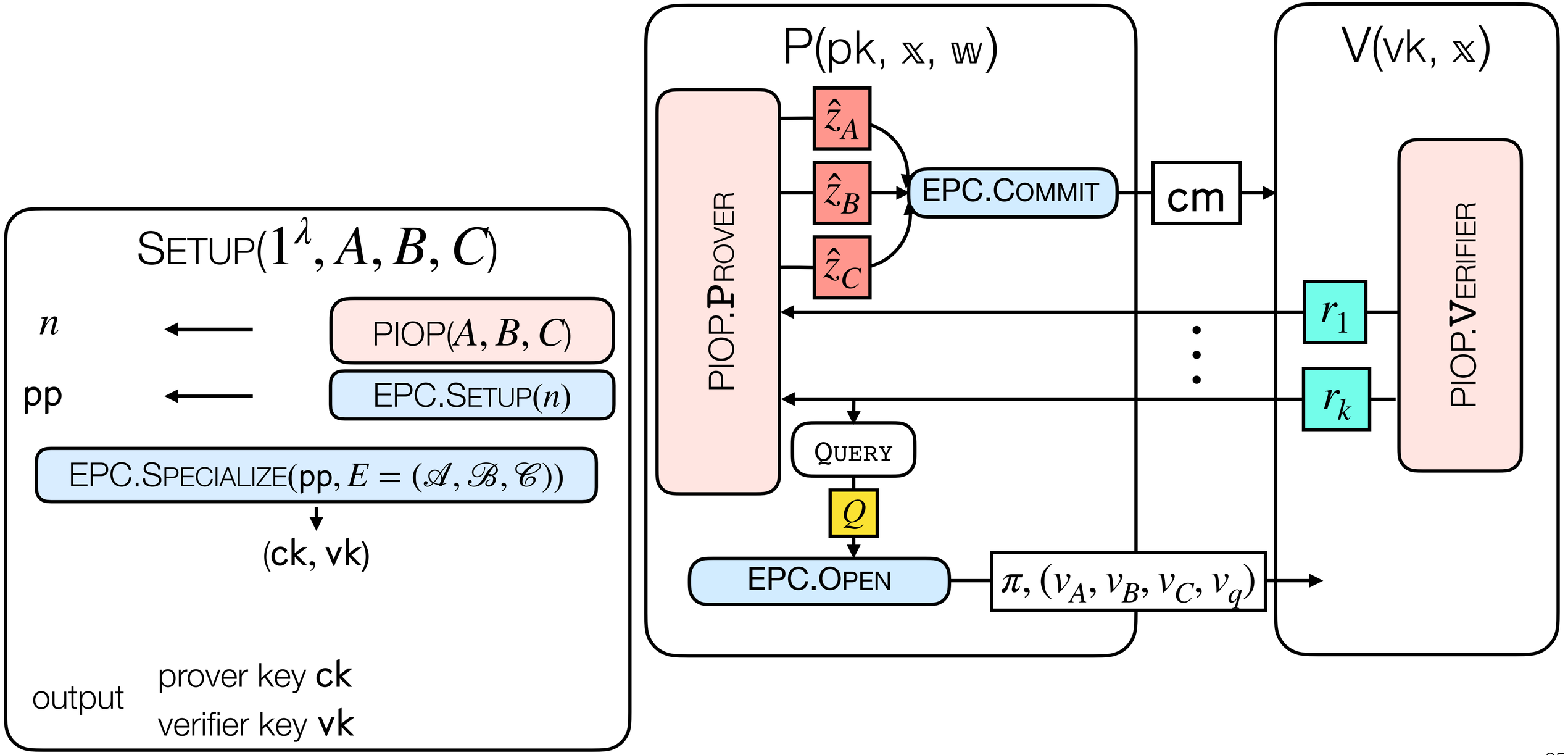
# PIOPs + EPC Schemes → SNARK



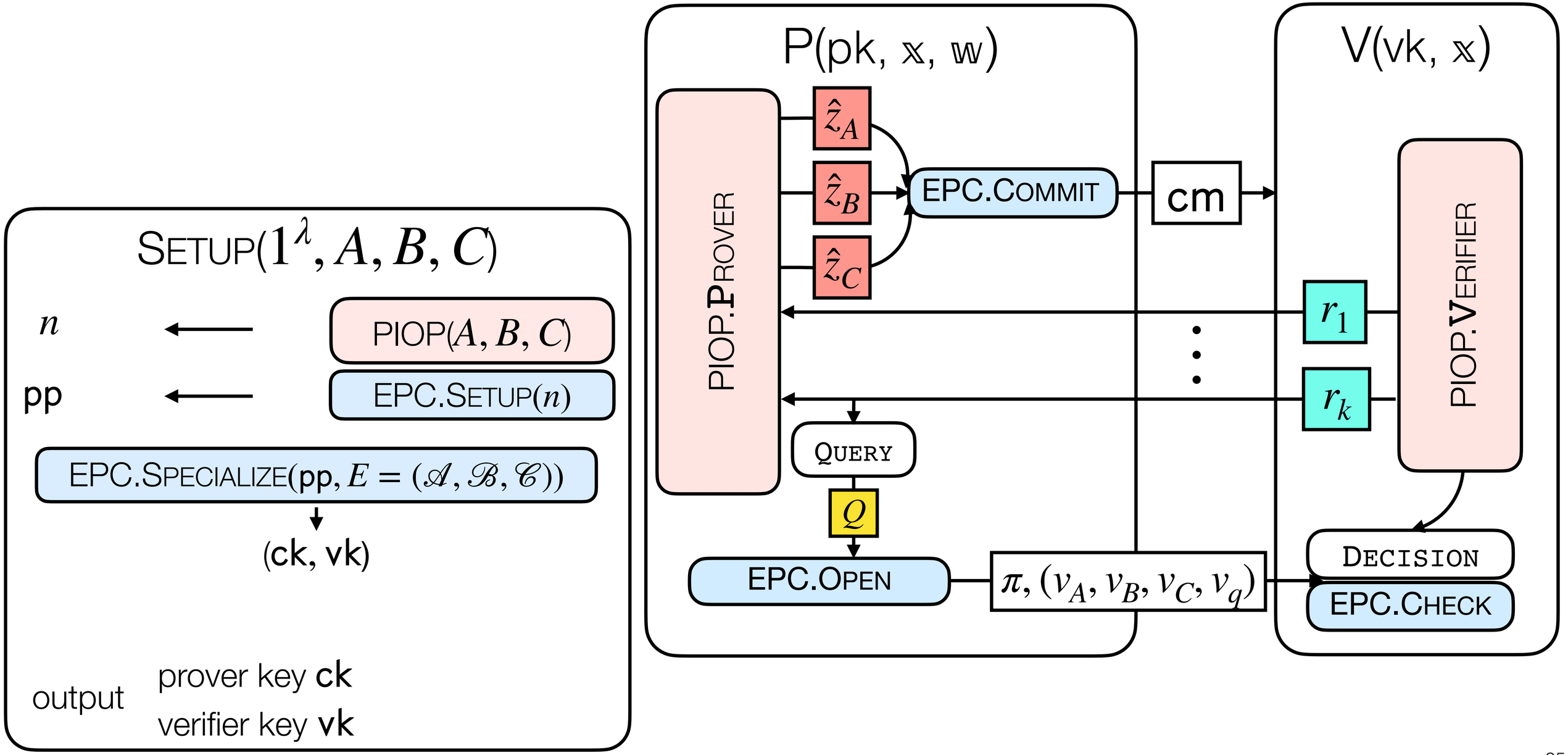
# PIOPs + EPC Schemes → SNARK



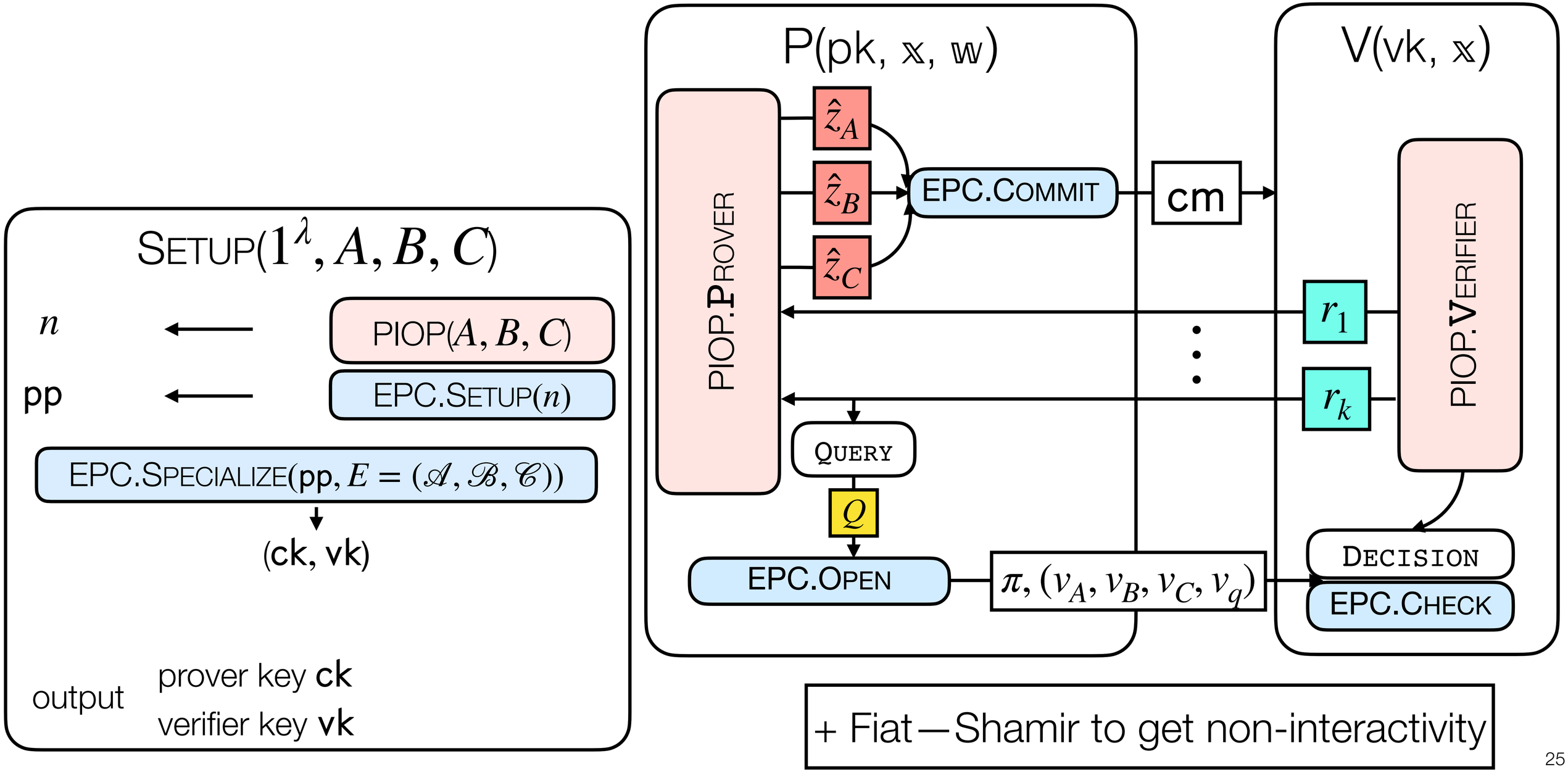
# PIOPs + EPC Schemes → SNARK



# PIOPs + EPC Schemes → SNARK



# PIOPs + EPC Schemes → SNARK



# Diff of Pari and Garuda

| Pari | Garuda |
| --- | --- |
| Square R1CS (SR1CS) | Generalized R1CS (GR1CS) |
| Univariate EPC (Batched) | Multivariate EPC (Non-Batched) |
| Univariate Rowcheck PIOP | Multivariate Rowcheck PIOP Using sumcheck protocol |

# Diff of Pari and Garuda

| Pari | Garuda |
| --- | --- |
| Square R1CS (SR1CS) | Generalized R1CS (GR1CS) |
| Univariate EPC (Batched) | Multivariate EPC (Non-Batched) |
| Univariate Rowcheck PIOP | Multivariate Rowcheck PIOP Using sumcheck protocol |

Support for any custom gates, e.g. Lookups

# Diff of Pari and Garuda

| Pari | Garuda |
| --- | --- |
| Square R1CS (SR1CS) | Generalized R1CS (GR1CS) |
| Univariate EPC (Batched) | Multivariate EPC (Non-Batched) |
| Univariate Rowcheck PIOP | Multivariate Rowcheck PIOP Using sumcheck protocol |

Support for any custom gates, e.g. Lookups

Free addition gates

# Diff of Pari and Garuda

| Pari | Garuda |
| --- | --- |
| Square R1CS (SR1CS) | Generalized R1CS (GR1CS) |
| Univariate EPC (Batched) | Multivariate EPC (Non-Batched) |
| Univariate Rowcheck PIOP | Multivariate Rowcheck PIOP Using sumcheck protocol |

Support for any custom gates, e.g. Lookups

Free addition gates

Linear-time prover

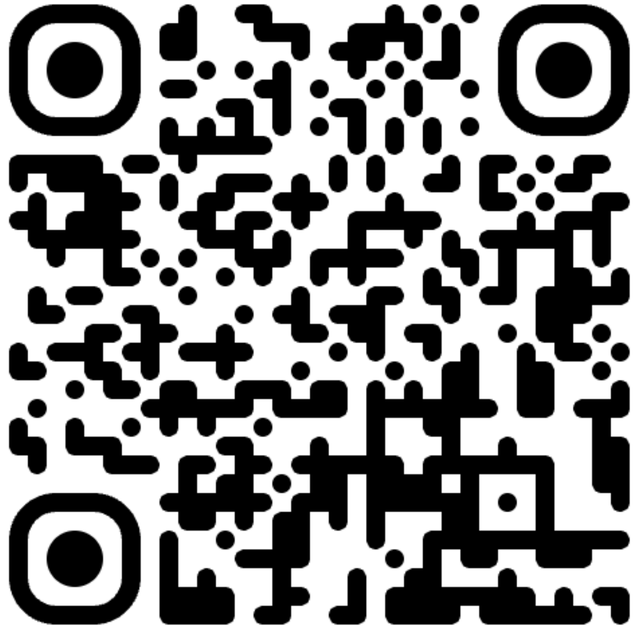
# Thanks!

Paper



*iacr.org/2024/1245*

Code



*github.com/alireza-shirzad/garuda-pari*