What does the SHARK say?

Mariana Raykova
Eran Trommer
Madars Virza
Zero-knowledge on the blockchain
Zero-knowledge on the blockchain

Privacy-preserving cryptocurrencies
Privacy-preserving smart contracts
Proof of regulatory compliance
Blockchain-based sovereign identity
Zero-knowledge on the blockchain

Privacy-preserving cryptocurrencies
Privacy-preserving smart contracts
Proof of regulatory compliance
Blockchain-based sovereign identity

see [ZKProof Standardization – Applications Track]
Zero-knowledge on the blockchain

Privacy-preserving cryptocurrencies
Privacy-preserving smart contracts
Proof of regulatory compliance
Blockchain-based sovereign identity

see [ZKProof Standardization – Applications Track]

Need zero-knowledge non-interactive arguments of knowledge, ideally succinct ones: zk-SNARK.
Which zk-SNARK?
Which zk-SNARK?

- QAP-based

[GGPR13][PGHR13][BCGTV13][DFGK14]
[Groth16][GM17][BG18]...

Fastest verification. Widely used.
Which zk-SNARK?

- QAP-based

  Fastest verification. Widely used.

  Require a Common Reference String (CRS)

[GGPR13][PGHR13][BCGTV13][DFGK14]
[Groth16][GM17][BG18]…
Which zk-SNARK?

• QAP-based

  [GGPR13][PGHR13][BCGTV13][DFGK14]
  [Groth16][GM17][BG18]…

  Fastest verification. Widely used.

  Require a Common Reference String (CRS)

  Structured Reference String (SRS)

ZKProof Standardization
Which zk-SNARK?

• QAP-based

Fastest verification. Widely used.

Require a Structured Reference String (SRS)
Which zk-SNARK?

• QAP-based

Fastest verification. Widely used.
Require a Structured Reference String (SRS)

Generating an SRS:
• Trusted setup
Which zk-SNARK?

- QAP-based

Fastest verification. Widely used.

Require a Structured Reference String (SRS)

Generating an SRS:
- Trusted setup
- MPC + destruction
- Updatable SRS
- Scalable SRS generation ("Powers of Tau")

[GGPR13][PGHR13][BCGTV13][DFGK14]
[Groth16][GM17][BG18]…

[BCGTV15][BGG17]

[GKMMM18, MBKM19]

[BG18]
Avoiding a Structured Reference String

Other zk-SNARKs

• PCP-based (e.g., *libSTARK*)

\[\text{[Micali94][BCGT13][BCS16][BBCGGHPRSTV17][BBHR18]}\]

Asymptotically succinct but large constants.
Avoiding a Structured Reference String

Other zk-SNARKs

- PCP-based (e.g., \textit{libSTARK})
  
  [Micali94][BCGT13][BCS16][BBCGGHPRSTV17][BBHR18]

  Asymptotically succinct but large constants.

Non-succinct ZK:
Avoiding a Structured Reference String

Other zk-SNARKs

- PCP-based (e.g., *libSTARK*)
  
  [Micali94][BCGT13][BCS16][BBCGGHPRSTV17][BBHR18]

  Asymptotically succinct but large constants.

Non-succinct ZK:

- Aurora [BCRSVW19]
- Bulletproofs [BCCGP16][BBBPWM17]
- Hyrax [WTSTW17]
- Ligero [AHIV17]
- ZKBoo(++) [GMO16][CDGORRSZ17]

Slow verification and/or large proofs (as statement grows).
Proof transmission & verification speed
Proof transmission & verification speed
Proof transmission & verification speed
Proof transmission & verification speed
Proof transmission & verification speed
Proof transmission & verification speed
Proof transmission & verification speed
Proof transmission & verification speed

Seeing transactions
Proof transmission & verification speed

Seeing transactions
Proof transmission & verification speed

- Seeing transactions
- Including new transactions in a block
Slow verification
→ miners get a head start by not validating
→ double-spends and chain splits [July 2015 Bitcoin fork]
Seeing transactions

Including new transactions in a block

Slow verification
→ miners get a head start by not validating
→ double-spend and chain splits [July 2015 Bitcoin fork]
Proof transmission & verification speed

- Seeing transactions
- Including new transactions in a block
- Verifying the previous block

Slow verification
→ miners get a head start by not validating
→ double-spends and chain splits [July 2015 Bitcoin fork]
zero-knowledge succinct hybrid argument of knowledge
zero-knowledge
succinct hybrid argument of knowledge

zk-SHARK

Short proof
zero-knowledge

succinct hybrid argument of knowledge

Prudent verification

Slow (comparable to Bulletproofs)

No SRS
zero-knowledge succinct hybrid argument of knowledge

**Prudent verification**
- Slow (comparable to Bulletproofs)
- No SRS

**Optimistic verification**
- Fast (comparable to QAP-based SNARK)
- Relies on SRS
Proof transmission & verification speed
Proof transmission & verification speed
Proof transmission & verification speed

Optimistically verify during propagation.
Proof transmission & verification speed

Optimistically verify during propagation.
Proof transmission & verification speed

Optimistically verify during propagation.
Proof transmission & verification speed

Optimistically verify during propagation.
Proof transmission & verification speed

Optimistically verify during propagation.
Proof transmission & verification speed

Optimistically verify during propagation.
Optimistically verify during propagation.
Optimistically verify during propagation.

Optimistically verify tx for inclusion in block template.

Prudently verify later, during PoW.
Proof transmission & verification speed

Optimistically verify during propagation.

Optimistically verify tx for inclusion in block template.

Prudently verify later, during PoW.
**Proof transmission & verification speed**

- **Optimistically** verify during propagation.
- **Optimistically** verify tx for inclusion in block template.
- **Prudently** verify later, during PoW.
- **Optimistically** verify incoming blocks.
- **Prudently** verify later, during PoW.
What if prudent verification fails?

Prudent verification

Optimistic verification
What if prudent verification fails?

Prudent verification

Optimistic verification
What if prudent verification fails?

Prudent verification

Optimistic verification

 berhasil
What if prudent verification fails?

- Detection
  - Real-time

Prudent verification

Optimistic verification
• Detection
  – Real-time
  – Revoke SRS
  – Such pair of proofs is a fraud proof for compromised SRS!
What if prudent verification fails?

- Detection
  - Real-time
  - Revoke SRS
  - Such pair of proofs is a fraud proof for compromised SRS!

- Recovery

Prudent verification

Optimistic verification
What if prudent verification fails?

- **Detection**
  - Real-time
  - Revoke SRS
  - Such pair of proofs is a *fraud proof* for compromised SRS!

- **Recovery**
  - Soundness: prudently verify

Prudent verification

Optimistic verification
What if prudent verification fails?

• Detection
  – Real-time
  – Revoke SRS
  – Such pair of proofs is a fraud proof for compromised SRS!

• Recovery
  – Soundness: prudently verify
  – Speed: regenerate SRS + refresh the optimistic proofs.
• Detection
  – Real-time
  – Revoke SRS
  – Such pair of proofs is a fraud proof for compromised SRS!

• Recovery
  – Soundness: prudently verify
  – Speed: regenerate SRS + refresh the optimistic proofs.
What if prudent verification fails?

- **Detection**
  - Real-time
  - Revoke SRS
  - Such pair of proofs is a fraud proof for compromised SRS!

- **Recovery**
  - Soundness: prudently verify
  - Speed: regenerate SRS + refresh the optimistic proofs.

SHARK requirement: anyone can refresh, without original sender.
A generic SHARK construction
A generic SHARK construction

Attempt – parallel composition:
A generic SHARK construction

Attempt – parallel composition:

1. Fix a language $L$ and construct a NIZK for it:
A generic SHARK construction

Attempt – parallel composition:

① Fix a language $L$ and construct a NIZK for it:

$x \rightarrow P_{\text{NIZK}}$

$w$

Not pictured: $G_{\text{NIZK}}$
A generic SHARK construction

Attempt – parallel composition:

1. Fix a language $L$ and construct a NIZK for it:

\[ x \rightarrow P_{\text{NIZK}} \]

\[ w \]

\[ x \in L \]

\[ \pi \]

Not pictured: $G_{\text{NIZK}}$
A generic SHARK construction

Attempt – parallel composition:

① Fix a language $L$ and construct a NIZK for it:

\[
\begin{align*}
\xrightarrow{x \rightarrow} & \quad P_{\text{NIZK}} \\
\xrightarrow{w} & \quad \pi \\
\xrightarrow{x \in L} & \quad V_{\text{NIZK}} \\
\xrightarrow{x} & 
\end{align*}
\]
Attempt – parallel composition:

① Fix a language $L$ and construct a NIZK for it:

Call $\pi$ “prudent proof”
Attempt – *parallel composition*:

① Fix a language $L$ and construct a NIZK for it:

\[ x \rightarrow P_{\text{NIZK}} \xrightarrow{\pi} V_{\text{NIZK}} \]

Call $\pi$ “prudent proof”

② Construct a SNARK for the same $L$:

\[ w \rightarrow x \in L \rightarrow x \rightarrow P_{\text{NIZK}} \xrightarrow{\pi} V_{\text{NIZK}} \]

Not pictured: $G_{\text{NIZK}}$
Attempt – parallel composition:

1. Fix a language $L$ and construct a NIZK for it:

   \[ P_{\text{NIZK}} \rightarrow \pi \rightarrow V_{\text{NIZK}} \]

   \[ x \rightarrow P_{\text{NIZK}} \]

   \[ w \rightarrow \pi \]

   \[ \pi \rightarrow x \in L \]

   \[ x \rightarrow V_{\text{NIZK}} \]

   Call $\pi$ “prudent proof”

2. Construct a SNARK for the same $L$:

   \[ P_{\text{SNARK}} \rightarrow \phi \rightarrow V_{\text{SNARK}} \]

   \[ x \rightarrow P_{\text{SNARK}} \]

   \[ w \rightarrow \phi \]

   \[ \phi \rightarrow x \in L \]

   \[ x \rightarrow V_{\text{SNARK}} \]

Not pictured: $G_{\text{NIZK}}, G_{\text{SNARK}}$
### A generic SHARK construction

**Attempt – parallel composition:**

1. Fix a language $L$ and construct a NIZK for it:

   $x \xleftarrow{} P_{\text{NIZK}} \xrightarrow{} \pi \xrightarrow{} x \in L \xrightarrow{} V_{\text{NIZK}}$

   Call $\pi$ “prudent proof”

2. Construct a SNARK for the same $L$:

   $x \xleftarrow{} P_{\text{SNARK}} \xrightarrow{} \phi \xrightarrow{} x \in L \xrightarrow{} V_{\text{SNARK}}$

Optimistic proofs should be refreshable without $w$

Not pictured: $G_{\text{NIZK}}, G_{\text{SNARK}}$
**A generic SHARK construction**

**Attempt – parallel composition:**

1. Fix a language $L$ and construct a NIZK for it:

   $\begin{align*}
   & x \rightarrow P_{\text{NIZK}} \\
   & \pi \rightarrow V_{\text{NIZK}} \\
   & x \in L
   \end{align*}$

   Call $\pi$ “prudent proof”

2. Construct a SNARK for the same $L$:

   $\begin{align*}
   & x \rightarrow P_{\text{SNARK}} \\
   & \phi \rightarrow V_{\text{SNARK}} \\
   & x \in L
   \end{align*}$

   Optimistic proofs should be refreshable without $w$

   $\Rightarrow \phi$ is **not** a SHARK optimistic proof

Not pictured: $G_{\text{NIZK}}, G_{\text{SNARK}}$
A generic SHARK construction
A generic SHARK construction

Construction:
A generic SHARK construction

Construction:

① Fix a language $L$ and construct a NIZK for it:
Construction:

1. Fix a language $L$ and construct a NIZK for it:
Construction:

1. Fix a language $L$ and construct a NIZK for it:

Call $\pi$ “prudent proof”
Construction:

1. Fix a language $L$ and construct a NIZK for it:

   - $x \rightarrow P_{NIZK}$
   - $\pi \rightarrow V_{NIZK}$
   - $x \in L$

   Call $\pi$ "prudent proof"

2. Construct a SNARK for "$V_{NIZK}$ accepts"

   Not pictured: $G_{NIZK}$
Construction:

1. Fix a language $L$ and construct a NIZK for it:

   - $P_{\text{NIZK}}$ with input $x$, and output $w$.

   - $V_{\text{NIZK}}$ with input $x$, and output $x \in L$.

   - Call $\pi$ “prudent proof”.

2. Construct a SNARK for “$V_{\text{NIZK}}$ accepts”

Not pictured: $G_{\text{NIZK}}$, $G_{\text{SNARK}}$
A generic SHARK construction

Construction:

① Fix a language $L$ and construct a NIZK for it:

$\mathcal{P}_{\text{NIZK}} \xrightarrow{w} x \xrightarrow{\pi} \mathcal{V}_{\text{NIZK}}$ with $x \in L$

Call $\pi$ “prudent proof”

② Construct a SNARK for “$\mathcal{V}_{\text{NIZK}}$ accepts”

Not pictured: $G_{\text{NIZK}}, G_{\text{SNARK}}$
Construction:

① Fix a language $L$ and construct a NIZK for it:

$P_{NIZK}(x, w)$

$\pi \in \{P, \phi\}$

$x \in L$

$V_{NIZK}$ accepts

Call $\pi$ “prudent proof”

② Construct a SNARK for “$V_{NIZK}$ accepts”

$P_{SNARK}$

$V_{SNARK}$

Not pictured: $G_{NIZK}, G_{SNARK}$
Construction:

1. Fix a language $L$ and construct a NIZK for it:

   \[ P_{\text{NIZK}} \xrightarrow{x} V_{\text{NIZK}} \]

   Call $\pi$ “prudent proof”

   \[ x \in L \]

2. Construct a SNARK for “$V_{\text{NIZK}}$ accepts”

   \[ P_{\text{SNARK}} \xrightarrow{x} V_{\text{SNARK}} \]

   Call $\phi$ “optimistic proof”

Not pictured: $G_{\text{NIZK}}, G_{\text{SNARK}}$
Construction:

① Fix a language $L$ and construct a NIZK for it:

$P_{\text{NIZK}} \xrightarrow{x} V_{\text{NIZK}}$  \hspace{1cm} $x \in L$

Call $\pi$ “prudent proof”

② Construct a SNARK for “$V_{\text{NIZK}}$ accepts”

$P_{\text{SNARK}} \xrightarrow{x} V_{\text{SNARK}}$

Call $\phi$ “optimistic proof”

Not pictured: $G_{\text{NIZK}}, G_{\text{SNARK}}$
Challenge: efficient SHARK construction
Challenge: efficient SHARK construction

Generically combining state-of-the-art components:
Challenge: efficient SHARK construction

Generically combining state-of-the-art components:
- Bulletproofs NIZK + Groth16 SNARK
Generically combining state-of-the-art components:

- Bulletproofs NIZK + Groth16 SNARK

$O(|C|)$-size circuit
Generically combining state-of-the-art components:

- Bulletproofs NIZK + Groth16 SNARK

\[ O(\lambda |C|) \text{ group ops in } V_{\text{NIZK}} \]

\[ O(|C|) \text{-size circuit} \]
Generically combining state-of-the-art components:
- Bulletproofs NIZK + Groth16 SNARK

$O(\lambda |C|)$ group ops in $V_{\text{NIZK}}$

$O(|C|)$-size circuit

some speed-up via multiexp
Challenge: efficient SHARK construction

- $O(\lambda^2 |C|)$ group ops in $P_{SNARK}$
- $O(\lambda |C|)$ group ops in $V_{NIZK}$
- $O(|C|)$-size circuit

Generically combining state-of-the-art components:

- Bulletproofs NIZK + Groth16 SNARK

Some speed-up via multiexp
Generically combining state-of-the-art components:

- Bulletproofs NIZK + Groth16 SNARK

Some speed-up via multiexp
Generically combining state-of-the-art components:

- Bulletproofs NIZK + Groth16 SNARK

some speed-up via multiexp
Technical contribution: efficient SHARK construction
Technical contribution: efficient SHARK construction

NIZK for RICS
Technical contribution: efficient SHARK construction

≈ “arithmetic circuits with bilinear gates”

NIZK for R1CS
• a new compilation technique for linear PCPs

\approx "arithmetic circuits with bilinear gates"

NIZK for RICS
Technical contribution: efficient SHARK construction

≈ “arithmetic circuits with bilinear gates”

• a new compilation technique for linear PCPs
• public coin NIZK from LPCPs ⇒ prudent mode
Technical contribution: efficient SHARK construction

• a new compilation technique for linear PCPs
• public coin NIZK from LPCPs ⇒ prudent mode
• an optimized variant of Bulletproofs’ inner product argument

≈ “arithmetic circuits with bilinear gates”
Technical contribution: efficient SHARK construction

NIZK for RICS

\approx \text{“arithmetic circuits with bilinear gates”}

• a new compilation technique for linear PCPs
• public coin NIZK from LPCPs ⇒ prudent mode
• an optimized variant of Bulletproofs’ inner product argument

⇒ \( V_{NIZK} \) has an “algebraic heart”
Technical contribution: efficient SHARK construction

≈ “arithmetic circuits with bilinear gates”

• a new compilation technique for linear PCPs
• public coin NIZK from LPCPs ⇒ prudent mode
• an optimized variant of Bulletproofs’ inner product argument

⇒ $V_{\text{NIZK}}$ has an “algebraic heart”

A special-purpose SNARK
Technical contribution: efficient SHARK construction

\[ \approx \text{“arithmetic circuits with bilinear gates”} \]

- a new compilation technique for linear PCPs
- public coin NIZK from LPCPs $\Rightarrow$ prudent mode
- an optimized variant of Bulletproofs’ inner product argument

$\Rightarrow V_{\text{NIZK}}$ has an “algebraic heart”

A special-purpose SNARK

for “encoded polynomial delegation”, a problem we introduce
Technical contribution: efficient SHARK construction

Our SHARK

\[ \Rightarrow V_{\text{NIZK}} \text{ has an “algebraic heart”} \]

A special-purpose SNARK

for “encoded polynomial delegation”, a problem we introduce

NIZK for R1CS

\approx \text{“arithmetic circuits with bilinear gates”}

- a new compilation technique for linear PCPs
- public coin NIZK from LPCPs \( \Rightarrow \) prudent mode
- an optimized variant of Bulletproofs’ inner product argument

\[ \Rightarrow V_{\text{NIZK}} \text{ has an “algebraic heart”} \]
Linear PCP paradigm
Design a proof system sound against linear provers
Linear PCP paradigm

① Design a proof system sound against linear provers
② Force prover to be linear using a cryptographic encoding
Linear PCP paradigm

1. Design a proof system sound against linear provers
2. Force prover to be linear using a cryptographic encoding
1. Design a proof system sound against linear provers
2. Force prover to be linear using a cryptographic encoding

Diagram:
- $P$ with inputs $x$ and $w$ leading to $\pi$
- $Q$ with a query sampler
Linear PCP paradigm

① Design a proof system sound against linear provers
② Force prover to be linear using a cryptographic encoding
Linear PCP paradigm

1. Design a proof system sound against linear provers
2. Force prover to be linear using a cryptographic encoding
Linear PCP paradigm

① Design a proof system sound against linear provers
② Force prover to be linear using a cryptographic encoding
Design a proof system sound against linear provers

Force prover to be linear using a cryptographic encoding
Design a proof system sound against linear provers

Force prover to be linear using a cryptographic encoding
Linear PCP paradigm

1. Design a proof system sound against linear provers
2. Force prover to be linear using a cryptographic encoding

Can define natural notions of completeness, PoK, ZK.
Our compilation technique
Our compilation technique

$P \quad V$
Our compilation technique

Observe that $P_{LPCP}$ does not need to know queries a priori.

$P$  \hspace{2cm} $V$
Our compilation technique

Observe that $P_{LPCP}$ does not need to know queries a priori.
Our compilation technique

Observe that $P_{\text{LPCP}}$ does not need to know queries a priori.

1. Run:

\[
P_{\text{LPCP}} \rightarrow \vec{\pi}
\]

$\vec{\pi} := \text{Commit}(\vec{\pi})$
Our compilation technique

Observe that $P_{LPCP}$ does not need to know queries a priori.
Our compilation technique

Observe that $P_{\text{LPCP}}$ does not need to know queries a priori.
Our compilation technique

Observe that $P_{LPCP}$ does not need to know queries a priori.

1. Run:
   $P_{LPCP} \xrightarrow{x, w} \vec{\pi}$

2. Pick coins for $Q_{LPCP}$

3. Run:
   $Q_{LPCP} \xrightarrow{\vec{q}^{(i)}}$
**Our compilation technique**

Observe that \( P_{LPCP} \) does not need to know queries a priori. *Public-coin* so can’t encrypt queries or use functional commitments. [BCIOP13] [LRY16]

1. Run: \( x \rightarrow P_{LPCP} \rightarrow \pi \)
2. Pick coins for \( Q_{LPCP} \)
3. Run: \( \text{state} \rightarrow Q_{LPCP} \rightarrow \tilde{q}^{(i)} \)

\[ cm_{\pi} := \text{Commit}(\pi) \]
Our compilation technique

Observe that $P_{\text{LPCP}}$ does not need to know queries a priori. 

*Public-coin* so can’t encrypt queries or use functional commitments.  

[BCIOP13] [LY16]
Our compilation technique

Observe that $P_{LPCP}$ does not need to know queries a priori. **Public-coin** so can’t encrypt queries or use functional commitments. [BCIO13] [LRY16]

1. Run:

   $x \rightarrow P_{LPCP} \rightarrow \pi$

   $w \rightarrow P_{LPCP} \rightarrow \pi$

3. Run:

   $Q_{LPCP} \rightarrow \tilde{q}^{(i)}$

   state

   $\text{cm}_{\pi} := \text{Commit}(\pi)$

   $\text{a}_i := \langle \tilde{q}^{(i)}, \pi \rangle$

2. Pick coins for $Q_{LPCP}$

4. Run $Q_{LPCP}$ to compute state.
Our compilation technique

Observe that $P_{LPCP}$ does not need to know queries a priori. *Public-coin* so can’t encrypt queries or use functional commitments. [BCIO13] [LRY16]

1. Run:

   $\pi \leftarrow P_{LPCP}(x, w)$

2. Pick coins for $Q_{LPCP}$

3. Run:

   $\tilde{q}^{(i)} \leftarrow Q_{LPCP}($ state $)$

4. Run $Q_{LPCP}$ to compute state.

Check that

$D_{LPCP}(x, a_i)$ accepts.
Our compilation technique

Observe that $P_{LPCP}$ does not need to know queries a priori. \textit{Public-coin} so can’t encrypt queries or use functional commitments. [BCIOP13] [LRY16]

1. Run: $x \xrightarrow{} P_{LPCP} \rightarrow \vec{π}$
2. Pick coins for $Q_{LPCP}$
3. Run: $Q_{LPCP}$
4. Run $Q_{LPCP}$ to compute state.

Check that $D_{LPCP}$ accepts.

$P$

$\vec{π}$

$\vec{q}^{(i)}$

$Q_{LPCP}$

$\text{state}$

$\text{accepts.}$

$\text{acc/rej}$

$\text{commit}$

$\text{Commit}(\vec{π})$

$\text{cm}_π$
Our compilation technique

Observe that $P_{LPCP}$ does not need to know queries a priori. 

Public-coin so can’t encrypt queries or use functional commitments.  

1. Run: $x \xrightarrow{w} P_{LPCP} \rightarrow \vec{\pi}$

2. Pick coins for $Q_{LPCP}$

3. Run: $Q_{LPCP}$

4. Run $Q_{LPCP}$ to compute state.

5. Check that $a_i$’s are consistent with $cm_{\vec{\pi}}$.

Check that $D_{LPCP}$ accepts.

[BCIOP13] [LRY16]
Answer consistency via inner product arguments
Verifier knows: $\mathcal{C}$, $a$’s, and commitment to proof $\mathbf{cm}_{\bar{\pi}} := \text{Commit}(\bar{\pi})$
Answer consistency via inner product arguments

Verifier knows: $\mathcal{A}$‘s, and commitment to proof $\text{cm}_{\vec{\pi}} := \text{Commit}(\vec{\pi})$

Goal: for every query $\vec{q}$ check $a = \langle \vec{q}, \vec{\pi} \rangle$ for a pre-committed $\vec{\pi}$
Answer consistency via inner product arguments

Verifier knows: \( \mathcal{V} \), \( a \)'s, and commitment to proof \( \text{cm}_{\vec{\pi}} := \text{Commit}(\vec{\pi}) \)

Goal: for every query \( \vec{q} \) check \( a = \langle \vec{q}, \vec{\pi} \rangle \) for a pre-committed \( \vec{\pi} \)

Technique: inner-product arguments

[BCCGP16, BBBPWM18]
Verifier knows: coins, a’s, and commitment to proof \( \text{cm}_{\vec{\pi}} := \text{Commit}(\vec{\pi}) \)

Goal: for every query \( \vec{q} \) check \( a = \langle \vec{q}, \vec{\pi} \rangle \) for a pre-committed \( \vec{\pi} \)

**Technique:** inner-product arguments

**Input:** Two vector Pedersen commitments \( \text{cm}_{\vec{u}}, \text{cm}_{\vec{v}}, \) and \( z \in \mathbb{F} \).
Verifier knows: $\mathcal{A}$, $a$’s, and commitment to proof $\text{cm}_{\vec{\pi}} := \text{Commit}(\vec{\pi})$

Goal: for every query $\vec{q}$ check $a = \langle \vec{q}, \vec{\pi} \rangle$ for a pre-committed $\vec{\pi}$

**Technique:** inner-product arguments

**Input:** Two vector Pedersen commitments $\text{cm}_{\vec{u}}$, $\text{cm}_{\vec{v}}$, and $z \in \mathbb{F}$.

**Prove:** The decommitments $\vec{u}, \vec{v} \in \mathbb{F}^n$ have the specified inner-product $z = \langle \vec{u}, \vec{v} \rangle$
Answer consistency via inner product arguments

Verifier knows: $\mathcal{A}$'s, and commitment to proof $\text{cm}_\pi := \text{Commit}(\pi)$

Goal: for every query $\hat{q}$ check $a = \langle \hat{q}, \pi \rangle$ for a pre-committed $\pi$

**Technique:** inner-product arguments [BCCGP16,BBBPWM18]

*Input:* Two vector Pedersen commitments $\text{cm}_\mathbf{u}$, $\text{cm}_\mathbf{v}$, and $z \in \mathbb{F}$.

*Prove:* The decommitments $\mathbf{u}, \mathbf{v} \in \mathbb{F}^n$ have the specified inner-product $z = \langle \mathbf{u}, \mathbf{v} \rangle$

So the verifier:
Answer consistency via inner product arguments

Verifier knows: \( m_i \), \( A \)'s, and commitment to proof \( \text{cm}_{\pi} := \text{Commit}(\pi) \)

Goal: for every query \( \hat{q} \) check \( a = \langle \hat{q}, \pi \rangle \) for a pre-committed \( \pi \)

Technique: inner-product arguments [BCCGP16,BBBPWM18]

Input: Two vector Pedersen commitments \( \text{cm}_u, \text{cm}_v \), and \( z \in \mathbb{F} \).

Prove: The decommitments \( \hat{u}, \hat{v} \in \mathbb{F}^n \) have the specified inner-product \( z = \langle \hat{u}, \hat{v} \rangle \)

So the verifier: ● computes commitments to queries: \( \text{cm}_{\hat{q}} := \text{Commit}(\hat{q}) \)
**Answer consistency via inner product arguments**

Verifier knows: \( \mathcal{C}, a \)'s, and commitment to proof \( \text{cm}_\pi := \text{Commit}(\pi) \)

Goal: for every query \( \tilde{q} \) check \( a = \langle \tilde{q}, \pi \rangle \) for a pre-committed \( \pi \)

**Technique: inner-product arguments**

*Input:* Two vector Pedersen commitments \( \text{cm}_\tilde{u}, \text{cm}_\tilde{v} \), and \( z \in \mathbb{F} \).

*Prove:* The decommitments \( \tilde{u}, \tilde{v} \in \mathbb{F}^n \) have the specified inner-product \( z = \langle \tilde{u}, \tilde{v} \rangle \)

So the verifier:

- computes commitments to queries: \( \text{cm}_{\tilde{q}} := \text{Commit}(\tilde{q}) \)
- engages in IP arguments for \( (\text{cm}_{\tilde{q}}, \text{cm}_\pi, a) \)
Verifier knows: a’s, and commitment to proof \( \text{Commit}(\vec{\pi}) \)

Goal: for every query \( \vec{q} \) check \( a = \langle \vec{q}, \vec{\pi} \rangle \) for a pre-committed \( \vec{\pi} \)

**Technique:** inner-product arguments

**Input:** Two vector Pedersen commitments \( \text{cm}_{\vec{u}}, \text{cm}_{\vec{v}}, \) and \( z \in \mathbb{F} \).

**Prove:** The decommitments \( \vec{u}, \vec{v} \in \mathbb{F}^n \) have the specified inner-product \( z = \langle \vec{u}, \vec{v} \rangle \)

So the verifier:
- computes commitments to queries: \( \text{cm}_{\vec{q}} := \text{Commit}(\vec{q}) \)
- engages in IP arguments for \( (\text{cm}_{\vec{q}}, \text{cm}_{\vec{\pi}}, a) \)

**Result:** NIZK from linear PCPs!
Answer consistency via inner product arguments

Verifier knows: $\mathcal{C}$, $a$'s, and commitment to proof $\text{cm}_{\vec{\pi}} := \text{Commit}(\vec{\pi})$

Goal: for every query $\vec{q}$ check $a = \langle \vec{q}, \vec{\pi} \rangle$ for a pre-committed $\vec{\pi}$

Technique: **inner-product arguments**

*Input:* Two vector Pedersen commitments $\text{cm}_u$, $\text{cm}_v$, and $z \in \mathbb{F}$.

*Prove:* The decommitments $\vec{u}, \vec{v} \in \mathbb{F}^n$ have the specified inner-product $z = \langle \vec{u}, \vec{v} \rangle$

So the verifier:

- computes commitments to queries: $\text{cm}_{\vec{q}} := \text{Commit}(\vec{q})$
- engages in IP arguments for $(\text{cm}_{\vec{q}}, \text{cm}_{\vec{\pi}}, a)$

**Result:** NIZK from linear PCPs!
The heart of our NIZK verifier
The heart of our NIZK verifier
The heart of our NIZK verifier

\[ V_{\text{NIZK}} \]

- check that LPCP decision predicate accepts (cheap)
The heart of our NIZK verifier

\[ V_{NIZK} \]

- check that LPCP decision predicate accepts (cheap)
- compute Pedersen commitment to each LPCP query \( \tilde{q} \) (costly)
The heart of our NIZK verifier

\[ V_{\text{NIZK}} \]

- check that LPCP decision predicate accepts (cheap)
- compute Pedersen commitment to each LPCP query \( \tilde{q} \) (costly)
- check that each inner product argument verifier accepts (costly)
The heart of our NIZK verifier

\[ V_{\text{NIZK}} \]

- check that LPCP decision predicate accepts (cheap)
- compute Pedersen commitment to each LPCP query \( \tilde{q} \) (costly)
- check that each inner product argument verifier accepts (costly)

(Not pictured: Fiat-Shamir transform, …)
The heart of our NIZK verifier

- check that LPCP decision predicate accepts (cheap)
- compute Pedersen commitment to each LPCP query $\tilde{q}$ (costly)
- check that each inner product argument verifier accepts (costly)

We will make $P_{SNARK}$ do both

(Not pictured: Fiat-Shamir transform, …)
The heart of our NIZK verifier

\[ V_{\text{NIZK}} \]

- check that LPCP decision predicate accepts (cheap)
- compute Pedersen commitment to each LPCP query \( \tilde{q} \) (costly)
- check that each inner product argument verifier accepts (costly)

A new building block: “encoded polynomial delegation”
we will make \( P_{\text{SNARK}} \) do both

(Not pictured: Fiat-Shamir transform, \ldots)
Outsourcing vector commitment computation
Goal: compute $\text{cm}_{\vec{q}} := \text{Commit}(\vec{q})$
Goal: compute $\mathbf{cm}_\tilde{q} := \text{Commit}(\tilde{q})$
Outsourcing vector commitment computation

Goal: compute $\text{cm}_{\tilde{q}} := \text{Commit}(\tilde{q})$

Most efficient linear PCP: quadratic arithmetic programs of [GGPR12]
Outsourcing vector commitment computation

Goal: compute $\text{cm}_{\tilde{q}} := \text{Commit}(\tilde{q})$

Most efficient linear PCP: quadratic arithmetic programs of [GGPR12]

Each query $\tilde{q}$ has nice algebraic structure:
Goal: compute $\mathbf{cm}_\tilde{q} := \text{Commit}(\tilde{q})$

Most efficient linear PCP: quadratic arithmetic programs of \cite{GGPR12}

Each query $\tilde{q}$ has nice algebraic structure:
Outsourcing vector commitment computation

Goal: compute $\mathbf{cm}_{\tilde{q}} := \text{Commit}(\tilde{q})$

Most efficient linear PCP: quadratic arithmetic programs of [GGPR12]

Each query $\tilde{q}$ has nice algebraic structure:

$$\tilde{q} = (p_1(\tau), p_2(\tau), ..., p_n(\tau))$$
Goal: compute $c_{\text{m}q} := \text{Commit}(\tilde{q})$

Most efficient linear PCP: quadratic arithmetic programs of [GGPR12]

Each query $\tilde{q}$ has nice algebraic structure:

$$
\tilde{q} = (p_1(\tau), p_2(\tau), \ldots, p_n(\tau))
$$

$$
p_i(\tau) = p_{i,0} + p_{i,1}\tau + \cdots + p_{i,d}\tau^d
$$
Goal: compute $\mathbf{cm}_{\tilde{q}} := \text{Commit}(\tilde{q})$

Most efficient linear PCP: quadratic arithmetic programs of [GGPR12]

Each query $\tilde{q}$ has nice algebraic structure:

$$\tilde{q} = (p_1(\tau), p_2(\tau), ..., p_n(\tau))$$
$$= (p_{1,0}, p_{2,0}, ..., p_{n,0})$$

$$p_i(\tau) = p_{i,0} + p_{i,1} \tau + \cdots + p_{i,d} \tau^d$$
Goal: compute $\mathbf{cm}_\tilde{q} := \text{Commit}(\tilde{q})$

Most efficient linear PCP: quadratic arithmetic programs of [GGPR12]

Each query $\tilde{q}$ has nice algebraic structure:

$$\tilde{q} = (p_1(\tau), p_2(\tau), ..., p_n(\tau))$$

$$= (p_{1,0}, p_{2,0}, ..., p_{n,0}) + \tau \cdot (p_{1,1}, p_{2,1}, ..., p_{n,1})$$

$$p_i(\tau) = p_{i,0} + p_{i,1}\tau + \cdots + p_{i,d}\tau^d$$
Goal: compute $\mathbf{cm}_{\mathbf{q}} := \text{Commit}(\mathbf{\hat{q}})$

Most efficient linear PCP: quadratic arithmetic programs of [GGPR12]

Each query $\mathbf{\hat{q}}$ has nice algebraic structure:

$$\mathbf{\hat{q}} = (p_1(\tau), p_2(\tau), ..., p_n(\tau))$$

$$= (p_{1,0}, p_{2,0}, ..., p_{n,0}) + \tau \cdot (p_{1,1}, p_{2,1}, ..., p_{n,1}) + \cdots + \tau^d \cdot (p_{1,d}, p_{2,d}, ..., p_{n,d})$$

$$p_i(\tau) = p_{i,0} + p_{i,1}\tau + \cdots + p_{i,d}\tau^d$$
Outsourcing vector commitment computation

Goal: compute $\text{cm}_{\tilde{q}} := \text{Commit}(\tilde{q})$

Most efficient linear PCP: quadratic arithmetic programs of [GGPR12]

Each query $\tilde{q}$ has nice algebraic structure:

$$\tilde{q} = (p_1(\tau), p_2(\tau), \ldots, p_n(\tau))$$

$$\text{cm}_{\tilde{q}} = \text{Commit}(p_1(\tau), p_2(\tau), \ldots, p_n(\tau))$$

$$\tau = \text{\textit{\$}}$$

$$\tilde{q} = (p_1, 0, p_2, 0, \ldots, p_n, 0) + \tau \cdot (p_1, 1, p_2, 1, \ldots, p_n, 1) + \ldots + \tau^d \cdot (p_1, d, p_2, d, \ldots, p_n, d)$$

$$p_i(\tau) = p_{i,0} + p_{i,1} \tau + \ldots + p_{i,d} \tau^d$$
Goal: compute $\mathbf{cm}_\tilde{q} := \text{Commit}(\tilde{q})$

Most efficient linear PCP: quadratic arithmetic programs of [GGPR12]

Each query $\tilde{q}$ has nice algebraic structure:

$$\tilde{q} = (p_1(\tau), p_2(\tau), \ldots, p_n(\tau)) = (p_{1,0}, p_{2,0}, \ldots, p_{n,0}) + \tau \cdot (p_{1,1}, p_{2,1}, \ldots, p_{n,1}) + \ldots + \tau^d \cdot (p_{1,d}, p_{2,d}, \ldots, p_{n,d})$$

$$p_i(\tau) = p_{i,0} + p_{i,1} \tau + \ldots + p_{i,d} \tau^d$$

$$\mathbf{cm}_\tilde{q} = \text{Commit}(p_1(\tau), p_2(\tau), \ldots, p_n(\tau)) = \text{Commit}(p_{1,0}, p_{2,0}, \ldots, p_{n,0})$$
Goal: compute \( \text{cm}_q := \text{Commit}(\hat{q}) \)

Most efficient linear PCP: quadratic arithmetic programs of [GGPR12]

Each query \( \hat{q} \) has nice algebraic structure:

\[
\hat{q} = (p_1(\tau), p_2(\tau), ..., p_n(\tau)) = (p_{1,0}, p_{2,0}, ..., p_{n,0}) + \tau \cdot (p_{1,1}, p_{2,1}, ..., p_{n,1}) + \cdots + \tau^d \cdot (p_{1,d}, p_{2,d}, ..., p_{n,d})
\]

\[
p_i(\tau) = p_{i,0} + p_{i,1} \tau + \cdots + p_{i,d} \tau^d
\]
Goal: compute $\mathbf{cm}_{\tilde{q}} := \text{Commit}(\tilde{q})$

Most efficient linear PCP: quadratic arithmetic programs of [GGPR12]

Each query $\tilde{q}$ has nice algebraic structure:

$$\tilde{q} = (p_1(\tau), p_2(\tau), \ldots, p_n(\tau))$$

$$\quad = (p_{1,0}, p_{2,0}, \ldots, p_{n,0}) + \tau \cdot (p_{1,1}, p_{2,1}, \ldots, p_{n,1}) + \cdots + \tau^d \cdot (p_{1,d}, p_{2,d}, \ldots, p_{n,d})$$

$p_i(\tau) = p_{i,0} + p_{i,1}\tau + \cdots + p_{i,d}\tau^d$

$$\mathbf{cm}_{\tilde{q}} = \text{Commit}(p_1(\tau), p_2(\tau), \ldots, p_n(\tau))$$

$$\quad = \text{Commit}(p_{1,0}, p_{2,0}, \ldots, p_{n,0}) + \tau \cdot \text{Commit}(p_{1,1}, p_{2,1}, \ldots, p_{n,1}) + \cdots + \tau^d \cdot \text{Commit}(p_{1,d}, p_{2,d}, \ldots, p_{n,d})$$
Encoded polynomial delegation
Goal: outsource computation of
\[ cm_d = \text{Com}(p_1(\tau), p_2(\tau), ..., p_n(\tau)) \]
\[ = \text{Com}(p_{1,0}, p_{2,0}, ..., p_{n,0}) + \]
\[ \tau \cdot \text{Com}(p_{1,1}, p_{2,1}, ..., p_{n,1}) + \]
\[ \tau^d \cdot \text{Com}(p_{1,d}, p_{2,d}, ..., p_{n,d}) \]
Goal: outsource computation of
\[ \text{cm}_q = \text{Com}(p_1(\tau), p_2(\tau), ..., p_n(\tau)) \]
\[ = \text{Com}(p_{1,0}, p_{2,0}, ..., p_{n,0}) + \tau \cdot \text{Com}(p_{1,1}, p_{2,1}, ..., p_{n,1}) + \ldots \]
\[ \tau^d \cdot \text{Com}(p_{1,d}, p_{2,d}, ..., p_{n,d}) \]

\[ \text{Com}(\_\_\_\_\_)'s \text{ are fully determined by } L! \]
Encrypted polynomial delegation

Goal: outsource computation of
\[ \text{cm}_{\vec{d}} = \text{Com}(p_1(\tau), p_2(\tau), \ldots, p_n(\tau)) \]
\[ = \text{Com}(p_{1,0}, p_{2,0}, \ldots, p_{n,0}) + \tau \cdot \text{Com}(p_{1,1}, p_{2,1}, \ldots, p_{n,1}) + \ldots + \tau^d \cdot \text{Com}(p_{1,d}, p_{2,d}, \ldots, p_{n,d}) \]

Com(\boxed{\ldots})'s are fully determined by \( L! \)

Fixed parameters \( U_0, U_1, \ldots, U_d \in \mathbb{G} \)
Goal: outsource computation of \(\mathbf{cm}_{\vec{q}} = \text{Com}(p_1(\tau), p_2(\tau), \ldots, p_n(\tau))\)

\[
\begin{align*}
\text{Com}(p_1(\tau), p_2(\tau), \ldots, p_n(\tau)) &= \text{Com}(p_{1,0}, p_{2,0}, \ldots, p_{n,0}) + \\
\tau \cdot \text{Com}(p_{1,1}, p_{2,1}, \ldots, p_{n,1}) + \\
\tau^d \cdot \text{Com}(p_{1,d}, p_{2,d}, \ldots, p_{n,d}) &
\end{align*}
\]

For outsourcing \(\mathbf{cm}_{\vec{q}}\) set

\[U_k = \text{Com}(p_{1,k}, p_{2,k}, \ldots, p_{n,k})\]
**Encoded polynomial delegation**

Goal: outsource computation of 
\[ \text{cm}_q = \text{Com}(p_1(\tau), p_2(\tau), ..., p_n(\tau)) \]
\[ = \text{Com}(p_{1,0}, p_{2,0}, ..., p_{n,0}) + \tau \cdot \text{Com}(p_{1,1}, p_{2,1}, ..., p_{n,1}) + \ldots + \tau^d \cdot \text{Com}(p_{1,d}, p_{2,d}, ..., p_{n,d}) \]

Com( )'s are fully determined by \( L \)!

Fixed parameters \( U_0, U_1, ..., U_d \in \mathbb{G} \)

Goal: given input \( \tau \in \mathbb{F} \),

For outsourcing \( \text{cm}_q \) set
\[ U_k = \text{Com}(p_{1,k}, p_{2,k}, ..., p_{n,k}) \]
Encoded polynomial delegation

Goal: outsource computation of
\[ \text{cm}_{\vec{q}} = \text{Com}(p_1(\tau), p_2(\tau), \ldots, p_n(\tau)) \]
\[ = \text{Com}(p_{1,0}, p_{2,0}, \ldots, p_{n,0}) + \tau \cdot \text{Com}(p_{1,1}, p_{2,1}, \ldots, p_{n,1}) + \ldots + \tau^d \cdot \text{Com}(p_{1,d}, p_{2,d}, \ldots, p_{n,d}) \]

Com(\text{\textcolor{red}{red}})'s are fully determined by \( L \)!

Fixed parameters \( U_0, U_1, \ldots, U_d \in \mathbb{G} \)

Goal: given input \( \tau \in \mathbb{F} \),
outsource this computation:
\[ U := U_0 + \tau \cdot U_1 + \ldots + \tau^d \cdot U_d \cdot \]

For outsourcing \( \text{cm}_{\vec{q}} \) set
\[ U_k = \text{Com}(p_{1,k}, p_{2,k}, \ldots, p_{n,k}) \]
Encoded polynomial delegation

Goal: outsource computation of $\mathbf{cm}_\vec{q} = \text{Com}(p_1(\tau), p_2(\tau), ..., p_n(\tau))$

$= \text{Com}(p_{1,0}, p_{2,0}, ..., p_{n,0}) + \tau \cdot \text{Com}(p_{1,1}, p_{2,1}, ..., p_{n,1}) + \ldots + \tau^d \cdot \text{Com}(p_{1,d}, p_{2,d}, ..., p_{n,d})$

Com(□)’s are fully determined by $L$!

Encoded polynomial delegation

Fixed parameters $U_0, U_1, ..., U_d \in \mathbb{G}$

Goal: given input $\tau \in \mathbb{F}$, outsource this computation:

$U := U_0 + \tau \cdot U_1 + \ldots + \tau^d \cdot U_d$.

For outsourcing $\mathbf{cm}_\vec{q}$ set

$U_k = \text{Com}(p_{1,k}, p_{2,k}, ..., p_{n,k})$
Goal: outsource computation of
\[ \text{cm}_{\vec{q}} = \text{Com}(p_1(\tau), p_2(\tau), \ldots, p_n(\tau)) \]
\[ = \text{Com}(p_{1,0}, p_{2,0}, \ldots, p_{n,0}) + \tau \cdot \text{Com}(p_{1,1}, p_{2,1}, \ldots, p_{n,1}) + \ldots + \tau^d \cdot \text{Com}(p_{1,d}, p_{2,d}, \ldots, p_{n,d}) \]

Com(\[\square\]]’s are fully determined by \(L\)!

\(\text{cm}_{\vec{q}}\)’s are fully determined by \(L\)!

\[\tau \cdot \text{Com}(p_{1,1}, p_{2,1}, \ldots, p_{n,1}) + \ldots + \tau^d \cdot \text{Com}(p_{1,d}, p_{2,d}, \ldots, p_{n,d}) \]

Encoded polynomial delegation

\[\text{Com}(p_{1,0}, p_{2,0}, \ldots, p_{n,0}) + \tau \cdot \text{Com}(p_{1,1}, p_{2,1}, \ldots, p_{n,1}) + \ldots + \tau^d \cdot \text{Com}(p_{1,d}, p_{2,d}, \ldots, p_{n,d}) \]

\[U := U_0 + \tau \cdot U_1 + \ldots + \tau^d \cdot U_d \]

For outsourcing \(\text{cm}_{\vec{q}}\) set
\[U_k = \text{Com}(p_{1,k}, p_{2,k}, \ldots, p_{n,k})\]

**New building block:** SNARK for encoded polynomial delegation in pairing groups
Encoded polynomial delegation

Goal: outsource computation of
\[ \text{cm}_{\vec{q}} = \text{Com}(p_1(\tau), p_2(\tau), ..., p_n(\tau)) \]
\[ = \text{Com}(p_{1,0}, p_{2,0}, ..., p_{n,0}) + \tau \cdot \text{Com}(p_{1,1}, p_{2,1}, ..., p_{n,1}) + \ ...
\]
\[ \tau^d \cdot \text{Com}(p_{1,d}, p_{2,d}, ..., p_{n,d}) \]

Com(\[\square\])’s are fully determined by \( L \)!

New building block: SNARK for encoded polynomial delegation in pairing groups + multilinear variant for our optimized IP argument.

Encoded polynomial delegation

Fixed parameters \( U_0, U_1, ..., U_d \in \mathbb{G} \)

Goal: given input \( \tau \in \mathbb{F} \), outsource this computation:
\[ U := U_0 + \tau \cdot U_1 + \cdots + \tau^d \cdot U_d. \]

For outsourcing \( \text{cm}_{\vec{q}} \) set
\[ U_k = \text{Com}(p_{1,k}, p_{2,k}, ..., p_{n,k}) \]
**Encoded polynomial delegation**

**Goal:** outsource computation of

\[ \text{cm}_{\vec{q}} = \text{Com}(p_1(\tau), p_2(\tau), ..., p_n(\tau)) \]

\[ = \text{Com}(p_{1,0}, p_{2,0}, ..., p_{n,0}) + \tau \cdot \text{Com}(p_{1,1}, p_{2,1}, ..., p_{n,1}) + \cdots + \tau^d \cdot \text{Com}(p_{1,d}, p_{2,d}, ..., p_{n,d}) \]

Com( )’s are fully determined by \( L \)!

**New building block:** SNARK for encoded polynomial delegation in pairing groups + multilinear variant for our optimized IP argument.

**Fixed parameters** \( U_0, U_1, ..., U_d \in \mathbb{G} \)

**Goal:** given input \( \tau \in \mathbb{F} \),

outsource this computation:

\[ U := U_0 + \tau \cdot U_1 + \cdots + \tau^d \cdot U_d \]

**For outsourcing cm_{\vec{q}} set**

\[ U_k = \text{Com}(p_{1,k}, p_{2,k}, ..., p_{n,k}) \]

**SHARK optimistic proofs**
Putting things together
Putting things together

$P_{\text{NIZK}} \xrightarrow{x} V_{\text{NIZK}}$

$\xrightarrow{w} P_{\text{SNARK}} \xrightarrow{\pi} V_{\text{SNARK}}$

$x \xrightarrow{\phi}$
Putting things together

$P_{NIZK}$

$V_{NIZK}$

$P_{SNARK}$

$V_{SNARK}$

encoded polynomial delegation

$x$

$\pi$

$\phi$

$w$

$\omega$

$\omega$
Putting things together

\[ \text{encoded polynomial delegation} \]

\[ x \]

\[ P_{\text{NIZK}} \rightarrow \pi \rightarrow V_{\text{NIZK}} \]

\[ w \]

\[ x \]

\[ P_{\text{SNARK}} \rightarrow \phi \rightarrow V_{\text{SNARK}} \]
Comparison with generic instantiation
Comparison with generic instantiation

\[ O(\lambda^2 |C|) \text{ group} \]

\[ \text{ops in } P_{SNARK} \]

\[ O(\lambda |C|) \text{ group} \]

\[ \text{ops in } V_{NIZK} \]

\[ O(|C|)-\text{size circuit} \]
Comparison with generic instantiation

- $O(\lambda^2 |C|)$ group ops in $P_{SNARK}$
- $O(\lambda |C|)$ group ops in $V_{NIZK}$
- $O(|C|)$-size circuit

Generic instantiation
Comparison with generic instantiation

\(O(\lambda^2|C|)\) group
ops in \(P_{\text{SNARK}}\)

\(O(\lambda|C|)\) group
ops in \(V_{\text{NIZK}}\)

\(O(|C|)\)-size circuit

Generic instantiation

This work
Comparison with generic instantiation

- $O(\lambda^2 |C|)$ group ops in $P_{SNARK}$
- $O(\lambda |C|)$ group ops in $V_{NIZK}$
- $O(|C|)$-size circuit

- $O(\lambda |C|)$ group ops in $V_{NIZK}$
- $O(|C|)$-size circuit

Generic instantiation

This work
Comparison with generic instantiation

$O(\lambda^2 |C|)$ group
ops in $P_{SNARK}$

$O(\lambda |C|)$ group
ops in $V_{NIZK}$

$O(|C|)$-size circuit

This work

$O(\lambda |C|)$ group
ops in $V_{NIZK}$, $P_{SNARK}$

$O(|C|)$-size circuit

Generic instantiation
Comparison with generic instantiation

Generic instantiation

\[ O(\lambda^2 |C|) \] group
ops in \( P_{SNARK} \)

\[ O(\lambda |C|) \] group
ops in \( V_{NIZK} \)

\[ O(|C|) \]-size circuit

This work

\[ O(\lambda |C|) \] group
ops in \( V_{NIZK}, P_{SNARK} \)

\[ O(|C|) \]-size circuit

+ competitive proof size and verification time
Comparison with generic instantiation

Generic instantiation

- $O(\lambda^2 |C|)$ group ops in $P_{SNARK}$
- $O(\lambda |C|)$ group ops in $V_{NIZK}$
- $O(|C|)$-size circuit

This work

- $O(\lambda |C|)$ group ops in $V_{NIZK}$, $P_{SNARK}$
- $O(|C|)$-size circuit
- + competitive proof size and verification time
  - e.g. 5 batchable pairings and $6 \times G_1$ for an optimistic proof
Swimming with SHARKs
Swimming with SHARKs

- New primitive: private-coin setup needed for performance but not for soundness or ZK
Swimming with SHARKs

- New primitive: private-coin setup needed for performance but not for soundness or ZK

- Compromised setup can be quickly detected and easily replaced
Swimming with SHARKs

- New primitive: private-coin setup needed for performance but not for soundness or ZK
- Compromised setup can be quickly detected and easily replaced
- Speed competitive with best current zk-SNARKs
Swimming with SHARKs

- New primitive: private-coin setup needed for performance but not for soundness or ZK

- Compromised setup can be quickly detected and easily replaced

- Speed competitive with best current zk-SNARKs

- New building blocks along the way:
  - Optimized inner product argument
  - SNARK for encoded polynomial delegation