Some Recent SNARKs

Saba Eskandarian
Qualifying Exam Talk

What is a zk-SNARK?

Zero-Knowledge

Succinct

Non-interactive

ARguments of

Knowledge

"Come, listen, my men, while I tell you again
The five unmistakable marks
By which you may know, wheresoever you go,
The warranted genuine Snarks."
-Lewis Carroll, The Hunting of the Snark

What is a zk-SNARK?

Zero-Knowledge

Succinct

Non-interactive

ARguments of

Knowledge

Prover

C, x, w, y
s.t.
C(x,w)=y

Verifier

C, x, y

What is a zk-SNARK?

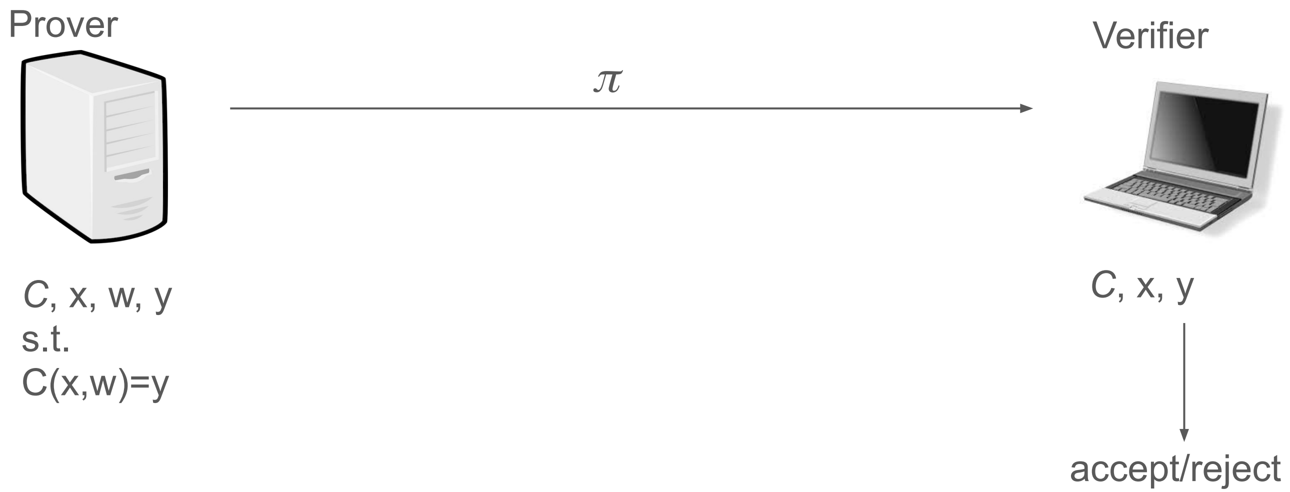
Zero-Knowledge

Succinct

Non-interactive

ARguments of

Knowledge



What is a zk-SNARK?

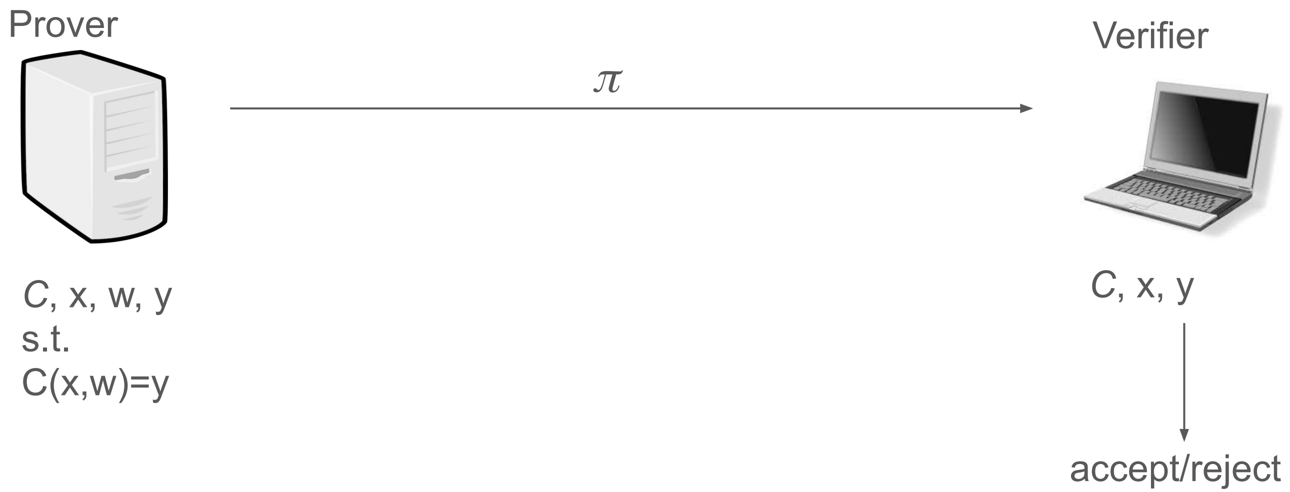
Zero-Knowledge

Succinct

Non-interactive

ARguments of

Knowledge



Correctness: If C(x,w)=y, then the verifier will accept π

What is a zk-SNARK?

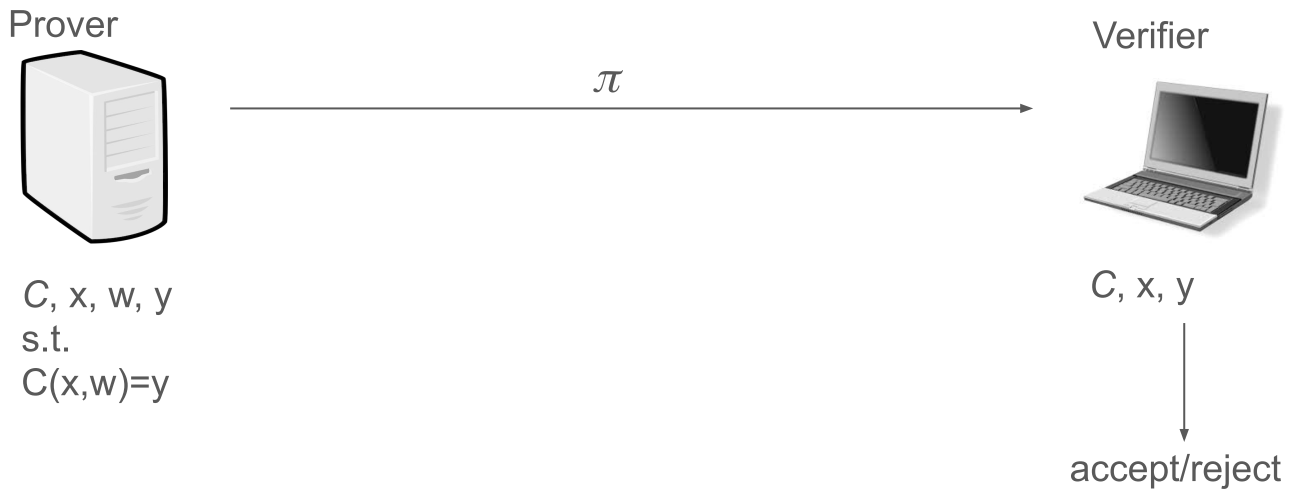
Zero-Knowledge

Succinct

Non-interactive

ARguments of

Knowledge



Zero-Knowledge: π reveals nothing about w

What is a zk-SNARK?

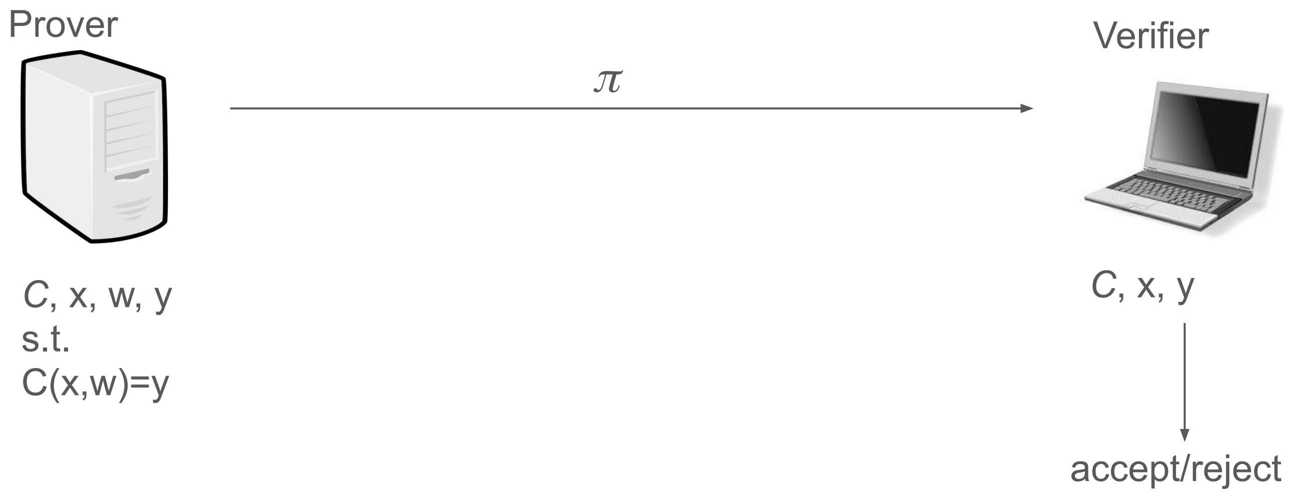
Zero-Knowledge

Succinct

Non-interactive

ARguments of

Knowledge



Succinct: 1. π is shorter than $|C|$

2. Verifier has to do less than $|C|$ work

What is a zk-SNARK?

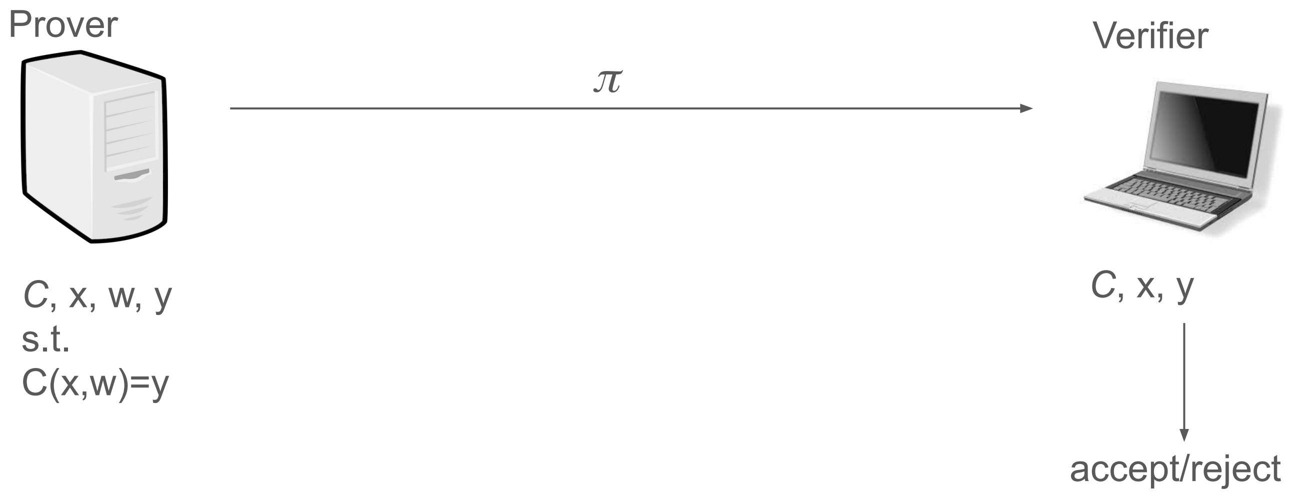
Zero-Knowledge

Succinct

Non-interactive

ARguments of

Knowledge



Non-interactive: π is one message from Prover to Verifier

What is a zk-SNARK?

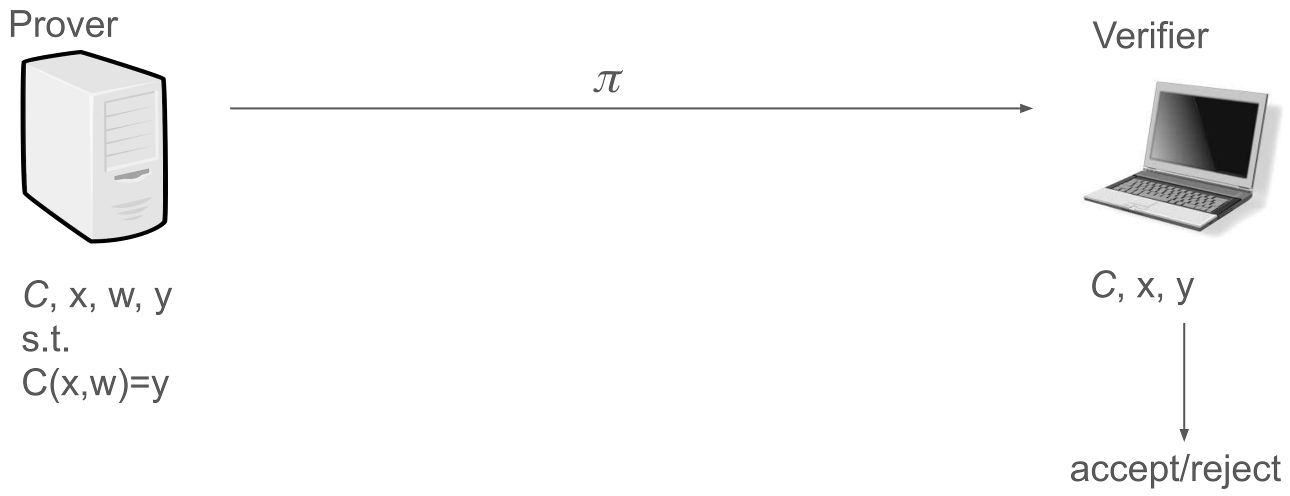
Zero-Knowledge

Succinct

Non-interactive

ARguments of

Knowledge



ARgument: the proof system is only computationally sound -- an unbounded Prover *can* prove false statements

What is a zk-SNARK?

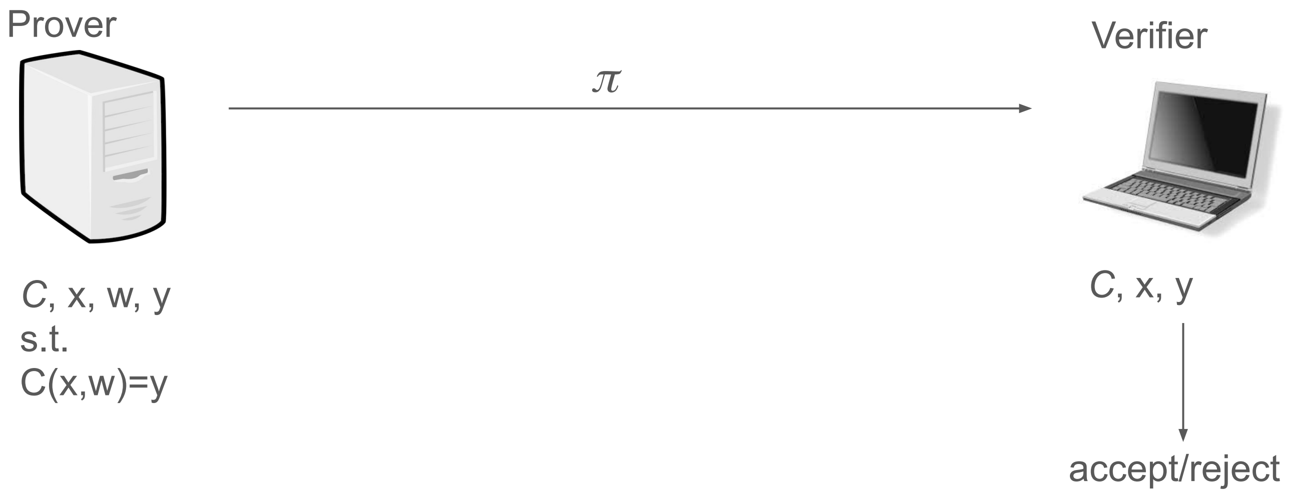
Zero-Knowledge

Succinct

Non-interactive

ARguments of

Knowledge



Soundness: For any unsatisfiable C(x,•)=y, ∀ $\underline{\text{PPT}}$ P*, ∀w

$\Pr[r \in_R \{0, 1\}^*: \langle P^*(w), V(r) \rangle (C, x, y) = \text{reject}] \geq 1 - \varepsilon_s$

What is a zk-SNARK?

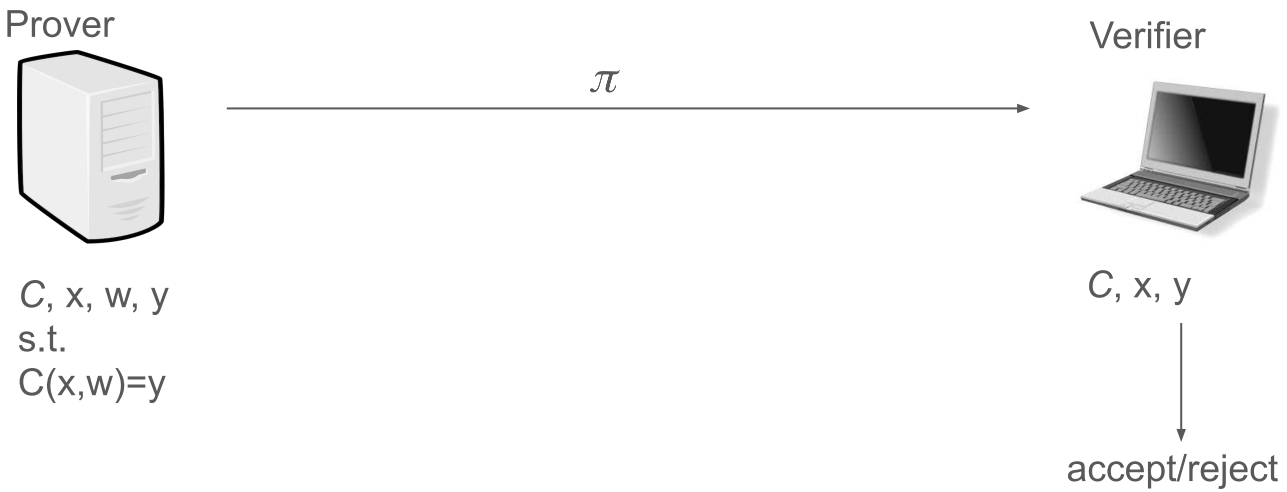
Zero-Knowledge

Succinct

Non-interactive

ARguments of

Knowledge



Knowledge: Prover really *knows* w, i.e., there is an *extractor* algorithm that interacts with Prover and outputs w

Families of Approaches

PCP + Merkle Trees (e.g., CS Proofs)

Linear PCPs, QAPs (e.g., IKO’07, Zaatar, GGPR13, Pinnochio, BCIOP13, BCGTV13, libSNARK)

IOPs/PCIPs (e.g., STARK, Aurora, Ligero, RRR16)

MPC-Based (e.g., ZKBoo, ZKB++, Ligero)

Discrete-log Based (e.g., BCCGP16, Bulletproofs, Hyrax)

Interactive Proofs (e.g., GKR, CMT, Thaler13, Giraffe, Hyrax, Clover, Spartan, vSQL, vRAM, zk-vSQL, Libra, Sonic, AuroraLight)

Families of Approaches

PCP + Merkle Trees (e.g., CS Proofs)

Linear PCPs, QAPs (e.g., IKO'07, Zaatar, GGPR13, Pinnochio, BCIOP13, BCGTV13, libSNARK)

IOPs/PCIPs (e.g., STARK, Aurora, Ligero, RRR16)

MPC-Based (e.g., ZKBoo, ZKB++, Ligero)

Discrete-log Based (e.g., BCCGP16, Bulletproofs, Hyrax)

Interactive Proofs (e.g., GKR, CMT, Thaler13, Giraffe, Hyrax, Clover, Spartan, vSQL, vRAM, zk-vSQL, Libra, Sonic, AuroraLight)

The Scourge of Trusted Setup

Many zk-SNARKs require a trusted setup to provide a CRS/SRS (common/structured reference string) that must be generated honestly

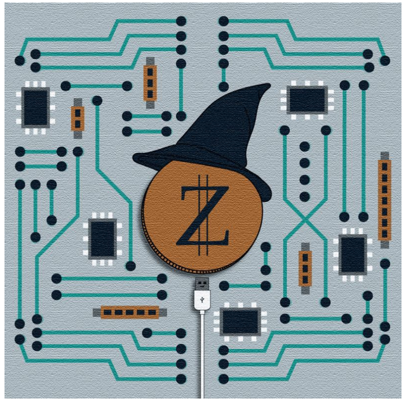
Cryptocurrency companies (and others) do elaborate “ceremonies” to inspire confidence in their CRSs

The Scourge of Trusted Setup

Many zk-SNARKs require a trusted setup to provide a CRS/SRS (common/structured reference string) that must be generated honestly

Cryptocurrency companies (and others) do elaborate "ceremonies" to inspire confidence in their CRSs

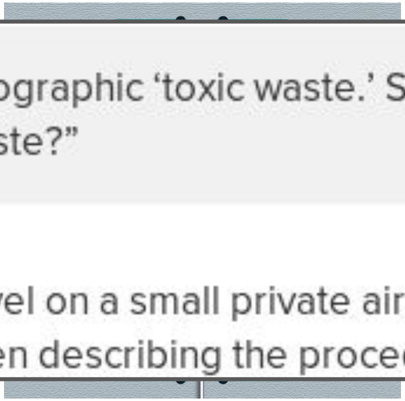
Fun listening: "The Ceremony", RadioLab, July 2017. https://www.wnycstudios.org/story/ceremony

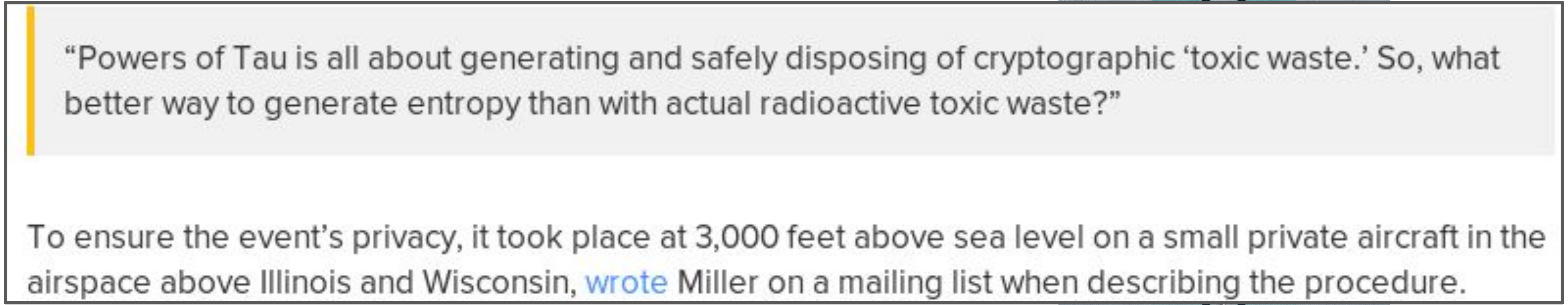


The Scourge of Trusted Setup

Many zk-SNARKs require a trusted setup to provide a CRS/SRS (common/structured reference string) that must be generated honestly

Cryptocurrency companies (and others) do elaborate “ceremonies” to inspire confidence in their CRSs





> “Powers of Tau is all about generating and safely disposing of cryptographic ‘toxic waste.’ So, what better way to generate entropy than with actual radioactive toxic waste?”

To ensure the event’s privacy, it took place at 3,000 feet above sea level on a small private aircraft in the airspace above Illinois and Wisconsin, wrote Miller on a mailing list when describing the procedure.

Coindesk, Jan 2018

This Talk: Minimizing Trusted Setup

Spartan: Efficient and general-purpose zkSNARKs without trusted setup

Srinath Setty (2019/550)

Almost completely eliminates trusted setup, but succinct means root, not log

This Talk: Minimizing Trusted Setup

Spartan: Efficient and general-purpose zkSNARKs without trusted setup

Srinath Setty (2019/550)

Almost completely eliminates trusted setup, but succinct means root, not log

Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings

Mary Maller, Sean Bowe, Markulf Kohlweiss, Sarah Meiklejohn (2019/099)

AuroraLight: Improved prover efficiency and SRS size in a Sonic-like system

Ariel Gabizon (2019/601)

Trusted setup can be updated later to increase confidence/security

A Common Template

1. Convert program to arithmetic circuit
2. Convert arithmetic circuit to polynomial
3. Build an argument to prove something about the polynomial
4. Add on other features generically

zk-SNARK

A Common Template

1. Convert program to arithmetic circuit
2. **Convert arithmetic circuit to polynomial**
3. **Build an argument to prove something about the polynomial**
4. Add on other features generically

~~zk~~-S~~N~~ARK

Tool: Polynomial Commitments

Commitments

$c \leftarrow \text{Commit}(m, r)$

$b \leftarrow \text{Verify}(c, m, r)$

Security Properties:

Binding: given commitment Commit(m, r), can’t decommit to $m' \neq m$

Hiding: given commitment Commit(m, r), can’t find m

Tool: Polynomial Commitments

Polynomial Commitments

F $\leftarrow$ Commit(pp, f(•))

(y, π) $\leftarrow$ Eval(pp, f(•), x)

b $\leftarrow$ Verify(pp, F, x, y, π)

Relevant Security Properties:

Correctness: In honest execution, y = f(x) and Verify(pp, F, x, y, π) = 1

Evaluation Binding: Can’t verify two different values f(x)=y, f(x)=y’ for y≠y’

Hiding: Can’t learn anything about f(•) at unqueried points

Tool: Polynomial Commitments

A (naive) polynomial commitment scheme:

1. Use any additively homomorphic (standard) commitment to commit to coefficients

Means we can add committed values to get commitment to their sum

$$5x^2 + 3x + 7$$

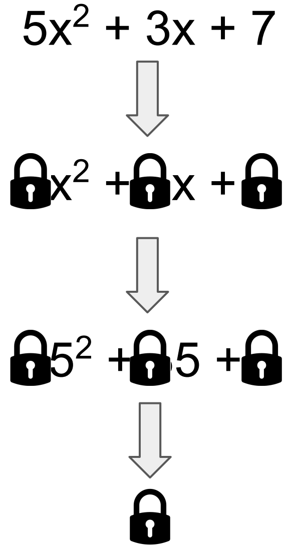
⇩

🔒x^2 + 🔒x + 🔒

Tool: Polynomial Commitments

A (naive) polynomial commitment scheme:

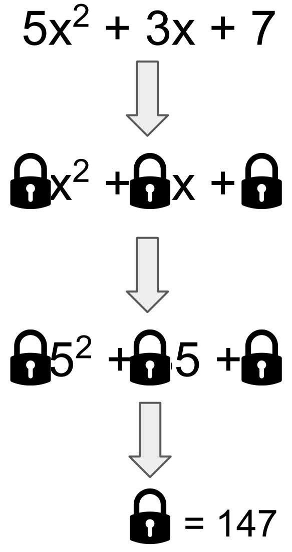
1. Use any additively homomorphic (standard) commitment to commit to coefficients
2. Use additive homomorphism to evaluate at given point



Tool: Polynomial Commitments

A (naive) polynomial commitment scheme:

1. Use any additively homomorphic (standard) commitment to commit to coefficients
2. Use additive homomorphism to evaluate at given point
3. Decommit to evaluated value



Tool: Polynomial Commitments

A (naive) polynomial commitment scheme:

1. Use any additively homomorphic (standard) commitment to commit to coefficients
2. Use additive homomorphism to evaluate at given point
3. Decommit to evaluated value

This scheme requires commitments *linear* in the size of the polynomial

The constructions in this talk rely on more efficient polynomial commitments

Spartan

Spartan

Pros: Removes trusted setup almost entirely

Verifier needs to read C once and never again

Con: Proof size and verifier time ~$\sqrt{(|C|)}$

Spartan

Pros: Removes trusted setup almost entirely

Verifier needs to read *C* once and never again

Con: Proof size and verifier time ~$\sqrt{(|C|)}$

Succinctness and setup requirements come from choice of polynomial commitment scheme that works on *multilinear* polynomials (from Hyrax [WTsTW17])

Spartan

Pros: Removes trusted setup almost entirely

Verifier needs to read *C* once and never again

Con: Proof size and verifier time ~$\sqrt{(|C|)}$

Succinctness and setup requirements come from choice of polynomial commitment scheme that works on *multilinear* polynomials (from Hyrax [WTsTW17])

Based on prior work (Clover [BTVW14]) using MIPs for verifiable computation

Spartan

Pros: Removes trusted setup almost entirely

Verifier needs to read C once and never again

Con: Proof size and verifier time $\sim\sqrt{(|C|)}$

Succinctness and setup requirements come from choice of polynomial commitment scheme that works on *multilinear* polynomials (from Hyrax [WTsTW17])

Based on prior work (Clover [BTVW14]) using MIPs for verifiable computation

Main additional tool: Sumcheck protocol [LFKN90]

Tool: Sumcheck Protocol

Goal: Prover wants to convince Verifier that an *m*-variate polynomial G:$\mathbf{F}^m \rightarrow \mathbf{F}$, where the maximum degree of any variable is *l*, sums to a value H over the m-dimensional binary hypercube, i.e.,

$$H = \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_m \in \{0,1\}} G(x_1, x_2, \ldots, x_m)$$

Approach: verifier checks sum one variable at a time

Spartan: Circuit to Polynomial (“front-end”)

Goal: Convert arithmetic circuit into a low-degree polynomial in variables $x_1,...,x_m$ such that the sum over the m-dimensional boolean hypercube is 0 if and only if the original circuit *C*(x,w)=y is satisfied by the prover’s value of w.

Then we can use sumcheck to prove this fact.

Spartan: Circuit to Polynomial (“front-end”)

Steps

1. Write the arithmetic circuit as a function
2. Convert the function to a low-degree polynomial
3. Give the polynomial the property that its sum over the boolean hypercube is 0
4. Put polynomial into format suitable for sumcheck

Spartan: Circuit to Polynomial (“front-end”)

1. Write the arithmetic circuit as a function

Label each gate in *C* with an *s*-bit identifier, *s*=*log*|*C*|

Spartan: Circuit to Polynomial (“front-end”)

1. Write the arithmetic circuit as a function

Label each gate in *C* with an *s*-bit identifier, *s*=*log*|*C*|
Define the following functions that capture the structure of the circuit

$$\mathrm{add}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ adds the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases} \qquad \mathrm{mul}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ multiplies the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases}$$

Spartan: Circuit to Polynomial (“front-end”)

1. Write the arithmetic circuit as a function

Label each gate in *C* with an *s*-bit identifier, *s=log|C|*
Define the following functions that capture the structure of the circuit

$$\mathrm{add}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ adds the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{mul}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ multiplies the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{io}(a,b,c) = \begin{cases} 1 & \text{if } a \text{ is a public input gate, and } b = c = 0 \\ 1 & \text{if } a \text{ is an output gate, and } b \text{ and } c \text{ are in-neighbors of } a \\ 0 & \text{otherwise} \end{cases}$$

$$I_{x,y}(a) = \begin{cases} x_a & \text{if } a \text{ is a public input gate } (x_a \text{ refers to an element of } x \text{ assigned to } a) \\ y_a & \text{if } a \text{ is an output gate} \\ 0 & \text{otherwise} \end{cases}$$

Spartan: Circuit to Polynomial ("front-end")

1. Write the arithmetic circuit as a function

$$\operatorname{add}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ adds the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases}$$

$$\operatorname{mul}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ multiplies the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases}$$

$$\operatorname{io}(a,b,c) = \begin{cases} 1 & \text{if } a \text{ is a public input gate, and } b = c = 0 \\ 1 & \text{if } a \text{ is an output gate, and } b \text{ and } c \text{ are in-neighbors of } a \\ 0 & \text{otherwise} \end{cases}$$

$$I_{x,y}(a) = \begin{cases} x_a & \text{if } a \text{ is a public input gate } (x_a \text{ refers to an element of } x \text{ assigned to } a) \\ y_a & \text{if } a \text{ is an output gate} \\ 0 & \text{otherwise} \end{cases}$$

Next, define *Z* as a function that, given a gate label *g*, outputs the value of the output of that gate in the computation of *C*(x,w).

Spartan: Circuit to Polynomial (“front-end”)

1. Write the arithmetic circuit as a function

$$\mathrm{add}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ adds the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{mul}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ multiplies the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{io}(a,b,c) = \begin{cases} 1 & \text{if } a \text{ is a public input gate, and } b = c = 0 \\ 1 & \text{if } a \text{ is an output gate, and } b \text{ and } c \text{ are in-neighbors of } a \\ 0 & \text{otherwise} \end{cases}$$

$$I_{x,y}(a) = \begin{cases} x_a & \text{if } a \text{ is a public input gate } (x_a \text{ refers to an element of } x \text{ assigned to } a) \\ y_a & \text{if } a \text{ is an output gate} \\ 0 & \text{otherwise} \end{cases}$$

Next, define *Z* as a function that, given a gate label *g*, outputs the value of the output of that gate in the computation of *C*(x,w).

$$F_{x,y}(a,b,c) = \mathrm{io}(a,b,c)\cdot(I_{x,y}(a) - Z(a)) + \mathrm{add}(a,b,c)\cdot(Z(a) - (Z(b)+Z(c))) + \mathrm{mul}(a,b,c)\cdot(Z(a) - Z(b)\cdot Z(c))$$

Key property: $F_{x,y}(a,b,c) = 0$ for all $(a,b,c) \in \{0,1\}^{3s}$ iff *Z* is a *satisfying assignment*, i.e., it represents a correct evaluation of C as described by F.

Spartan: Circuit to Polynomial (“front-end”)

1. Write the arithmetic circuit as a function

$$\mathrm{add}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ adds the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases} \qquad \mathrm{mul}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ multiplies the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{io}(a,b,c) = \begin{cases} 1 & \text{if } a \text{ is a public input gate, and } b = c = 0 \\ 1 & \text{if } a \text{ is an output gate, and } b \text{ and } c \text{ are in-neighbors of } a \\ 0 & \text{otherwise} \end{cases} \qquad I_{x,y}(a) = \begin{cases} x_a & \text{if } a \text{ is a public input gate } (x_a \text{ refers to an element of } x \text{ assigned to } a) \\ y_a & \text{if } a \text{ is an output gate} \\ 0 & \text{otherwise} \end{cases}$$

Next, define *Z* as a function that, given a gate label *g*, outputs the value of the output of that gate in the computation of *C*(x,w).

$$F_{x,y}(a,b,c) = \mathrm{io}(a,b,c)\cdot(I_{x,y}(a) - Z(a)) + \mathrm{add}(a,b,c)\cdot(Z(a) - (Z(b)+Z(c))) + \mathrm{mul}(a,b,c)\cdot(Z(a) - Z(b)\cdot Z(c))$$

Key property: $F_{x,y}(a,b,c) = 0$ for all $(a,b,c) \in \{0,1\}^{3s}$ iff *Z* is a *satisfying assignment*, i.e., it represents a correct evaluation of C as described by F.

Spartan: Circuit to Polynomial ("front-end")

1. Write the arithmetic circuit as a function

$$\mathrm{add}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ adds the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases} \qquad \mathrm{mul}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ multiplies the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{io}(a,b,c) = \begin{cases} 1 & \text{if } a \text{ is a public input gate, and } b = c = 0 \\ 1 & \text{if } a \text{ is an output gate, and } b \text{ and } c \text{ are in-neighbors of } a \\ 0 & \text{otherwise} \end{cases} \qquad I_{x,y}(a) = \begin{cases} x_a & \text{if } a \text{ is a public input gate } (x_a \text{ refers to an element of } x \text{ assigned to } a) \\ y_a & \text{if } a \text{ is an output gate} \\ 0 & \text{otherwise} \end{cases}$$

Next, define *Z* as a function that, given a gate label *g*, outputs the value of the output of that gate in the computation of *C*(x,w).

$$F_{x,y}(a,b,c) = \mathrm{io}(a,b,c)\cdot(I_{x,y}(a) - Z(a)) + \mathrm{add}(a,b,c)\cdot(Z(a) - (Z(b)+Z(c))) + \mathrm{mul}(a,b,c)\cdot(Z(a) - Z(b)\cdot Z(c))$$

Key property: $F_{x,y}(a,b,c) = 0$ for all $(a,b,c) \in \{0,1\}^{3s}$ iff *Z* is a *satisfying assignment*, i.e., it represents a correct evaluation of C as described by F.

Spartan: Circuit to Polynomial (“front-end”)

1. Write the arithmetic circuit as a function

$$\text{add}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ adds the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases}$$

$$\text{mul}(a,b,c) = \begin{cases} 1 & \text{if gate } a \text{ multiplies the outputs of gates } b \text{ and } c \\ 0 & \text{otherwise} \end{cases}$$

$$\text{io}(a,b,c) = \begin{cases} 1 & \text{if } a \text{ is a public input gate, and } b = c = 0 \\ 1 & \text{if } a \text{ is an output gate, and } b \text{ and } c \text{ are in-neighbors of } a \\ 0 & \text{otherwise} \end{cases}$$

$$I_{x,y}(a) = \begin{cases} x_a & \text{if } a \text{ is a public input gate } (x_a \text{ refers to an element of } x \text{ assigned to } a) \\ y_a & \text{if } a \text{ is an output gate} \\ 0 & \text{otherwise} \end{cases}$$

Next, define *Z* as a function that, given a gate label *g*, outputs the value of the output of that gate in the computation of *C*(x,w).

$$F_{x,y}(a,b,c) \boxed{= \text{io}(a,b,c) \cdot (I_{x,y}(a) - Z(a))} + \text{add}(a,b,c) \cdot (Z(a) - (Z(b) + Z(c))) + \text{mul}(a,b,c) \cdot (Z(a) - Z(b) \cdot Z(c))$$

Key property: $F_{x,y}(a,b,c) = 0$ for all $(a,b,c) \in \{0,1\}^{3s}$ iff *Z* is a *satisfying assignment*, i.e., it represents a correct evaluation of C as described by F.

Spartan: Circuit to Polynomial (“front-end”)

2. Convert the function to a low-degree polynomial

Idea: replace each function in $F_{x,y}$ with its *multilinear extension*

Spartan: Circuit to Polynomial ("front-end")

2. Convert the function to a low-degree polynomial

Idea: replace each function in $F_{x,y}$ with its *multilinear extension*, e.g.,

$$\begin{aligned}\widetilde{Z}(x_1,\ldots,x_m) &= \sum_{e\in\{0,1\}^m} Z(e)\cdot\prod_{i=1}^{m}(x_i\cdot e_i+(1-x_i)\cdot(1-e_i))\\ &= \sum_{e\in\{0,1\}^m} Z(e)\cdot\chi_e\\ &= \langle(Z(0),\ldots,Z(2^m-1)),(\chi_0,\ldots,\chi_{2^m-1})\rangle\end{aligned}$$

Where $Z:\{0,1\}^m\rightarrow\mathbf{F}$, the multilinear extension is a polynomial in m variables ($\mathbf{F}^m\rightarrow\mathbf{F}$) that agrees with Z on $\{0,1\}^m$.

Spartan: Circuit to Polynomial (“front-end”)

2. Convert the function to a low-degree polynomial

Idea: replace each function in $F_{x,y}$ with its *multilinear extension*, e.g.,

$$\begin{aligned}\widetilde{Z}(x_1,\ldots,x_m) &= \sum_{e\in\{0,1\}^m} Z(e)\cdot\prod_{i=1}^{m}\left(x_i\cdot e_i+(1-x_i)\cdot(1-e_i)\right)\\ &= \sum_{e\in\{0,1\}^m} Z(e)\cdot\chi_e\\ &= \langle (Z(0),\ldots,Z(2^m-1)),(\chi_0,\ldots,\chi_{2^m-1})\rangle\end{aligned}$$

Evaluates to *Z(e)* when each value x_i matches the corresponding bit of *e*

Linear in each x_i

Where $Z:\{0,1\}^m\rightarrow\mathbf{F}$, the multilinear extension is a polynomial in m variables ($\mathbf{F}^m\rightarrow\mathbf{F}$) that agrees with Z on $\{0,1\}^m$.

Spartan: Circuit to Polynomial ("front-end")

2. Convert the function to a low-degree polynomial

Idea: replace each function in $F_{x,y}$ with its *multilinear extension*, e.g.,

$$\begin{aligned}\widetilde{Z}(x_1,\ldots,x_m) &= \sum_{e\in\{0,1\}^m} Z(e)\cdot\prod_{i=1}^{m}(x_i\cdot e_i+(1-x_i)\cdot(1-e_i))\\ &= \sum_{e\in\{0,1\}^m} Z(e)\cdot\chi_e\\ &= \langle(Z(0),\ldots,Z(2^m-1)),(\chi_0,\ldots,\chi_{2^m-1})\rangle\end{aligned}$$

In our setting *m=3s* because each of *a,b,c* is *s* bits long

Values of $x_1 \ldots x_m$ select entries of Z because χ_e is 1 iff $e_i = x_i$ for $i \in [m]$

Where $Z:\{0,1\}^m \rightarrow \mathbf{F}$, the multilinear extension is a polynomial in m variables ($\mathbf{F}^m \rightarrow \mathbf{F}$) that agrees with Z on $\{0,1\}^m$.

Spartan: Circuit to Polynomial ("front-end")

2. Convert the function to a low-degree polynomial

Idea: replace each function in $F_{x,y}$ with its *multilinear extension*

$$F_{x,y}(a,b,c) = \mathrm{io}(a,b,c)\cdot(I_{x,y}(a) - Z(a)) + \mathrm{add}(a,b,c)\cdot(Z(a) - (Z(b)+Z(c))) + \mathrm{mul}(a,b,c)\cdot(Z(a) - Z(b)\cdot Z(c))$$

Spartan: Circuit to Polynomial (“front-end”)

2. Convert the function to a low-degree polynomial

Idea: replace each function in $F_{x,y}$ with its *multilinear extension*

$$F_{x,y}(a,b,c) = \mathrm{io}(a,b,c)\cdot(I_{x,y}(a) - Z(a)) + \mathrm{add}(a,b,c)\cdot(Z(a) - (Z(b)+Z(c))) + \mathrm{mul}(a,b,c)\cdot(Z(a) - Z(b)\cdot Z(c))$$

$$\Downarrow$$

$$\begin{aligned}\widetilde{F}_{x,y}(u_1,u_2,u_3) = {} & \widetilde{\mathrm{io}}(u_1,u_2,u_3)\cdot(\widetilde{I}_{x,y}(u_1) - \widetilde{Z}(u_1)) + \\ & \widetilde{\mathrm{add}}(u_1,u_2,u_3)\cdot(\widetilde{Z}(u_1) - (\widetilde{Z}(u_2) + \widetilde{Z}(u_3))) + \\ & \widetilde{\mathrm{mul}}(u_1,u_2,u_3)\cdot(\widetilde{Z}(u_2) - \widetilde{Z}(u_2)\cdot\widetilde{Z}(u_3))\end{aligned}$$

Each u_i is a vector of s field elements.

Spartan: Circuit to Polynomial (“front-end”)

3. Give the polynomial the property that its sum over the boolean hypercube is 0

Problem with running sumcheck directly on what we have: nonzero terms could cancel out and result in sumcheck accepting unsatisfying assignment.

$$Q_{x,y} = \sum_{u \in \{0,1\}^{3s}} \widetilde{F}_{x,y}(u)$$

Spartan: Circuit to Polynomial (“front-end”)

3. Give the polynomial the property that its sum over the boolean hypercube is 0

Problem with running sumcheck directly on what we have: nonzero terms could cancel out and result in sumcheck accepting unsatisfying assignment.

Solution: Multiply each item summed by a different power of an input *t*

$$Q_{x,y}(t) = \sum_{u \in \{0,1\}^{3s}} \widetilde{F}_{x,y}(u) \cdot \boxed{t^{\sum_{i=0}^{3s-1} u_i \cdot 2^i}}$$

$t \in \boldsymbol{F}$, u_i is ith bit of u

Spartan: Circuit to Polynomial (“front-end”)

3. Give the polynomial the property that its sum over the boolean hypercube is 0

Problem with running sumcheck directly on what we have: nonzero terms could cancel out and result in sumcheck accepting unsatisfying assignment.

Solution: Multiply each item summed by a different power of an input *t*

$$Q_{x,y}(t) = \sum_{u \in \{0,1\}^{3s}} \widetilde{F}_{x,y}(u) \cdot \boxed{t^{\sum_{i=0}^{3s-1} u_i \cdot 2^i}}$$

Now *Q* is a zero polynomial iff *F* evaluates to 0 at every point in the boolean hypercube $u \in \{0,1\}^{3s}$.

$t \in \boldsymbol{F}$, u_i is ith bit of u

Spartan: Circuit to Polynomial ("front-end")

4. Put polynomial into format suitable for sumcheck

Our argument system will run sumcheck over low degree polynomials, but our function is not low degree in *u* as written.

u_i is in the exponent

$$Q_{x,y}(t) = \sum_{u \in \{0,1\}^{3s}} \widetilde{F}_{x,y}(u) \cdot t^{\sum_{i=0}^{3s-1} u_i \cdot 2^i}$$

Spartan: Circuit to Polynomial ("front-end")

4. Put polynomial into format suitable for sumcheck

Our argument system will run sumcheck over low degree polynomials, but our function is not low degree in *u* as written.

Observation: $$\tau^{\sum_{i=0}^{3s-1} u_i \cdot 2^i} = \prod_{i=0}^{3s-1} S_\tau(u,i)$$

$$S_\tau(u,i) = \begin{cases} \tau^{2^i} & \text{if } u_i = 1 \\ 1 & \text{otherwise} \end{cases} = 1 + (\tau^{2^i} - 1) \cdot u_i$$

Spartan: Circuit to Polynomial (“front-end”)

4. Put polynomial into format suitable for sumcheck

Our argument system will run sumcheck over low degree polynomials, but our function is not low degree in *u* as written.

Observation: $$\tau^{\sum_{i=0}^{3s-1} u_i \cdot 2^i} = \prod_{i=0}^{3s-1} S_\tau(u,i)$$

$$S_\tau(u,i) = \begin{cases} \tau^{2^i} & \text{if } u_i = 1 \\ 1 & \text{otherwise} \end{cases} = 1 + (\tau^{2^i} - 1) \cdot u_i$$

$$Q_{x,y}(\tau) = \sum_{u \in \{0,1\}^{3s}} \widetilde{F}_{x,y}(u) \cdot \tau^{\sum_{i=0}^{3s-1} u_i \cdot 2^i} = \sum_{u \in \{0,1\}^{3s}} \widetilde{F}_{x,y}(u) \cdot \prod_{i=0}^{3s-1} S_\tau(u,i) = \sum_{u \in \{0,1\}^{3s}} G_{x,y,\tau}(u)$$

Spartan: Circuit to Polynomial ("front-end")

4. Put polynomial into format suitable for sumcheck

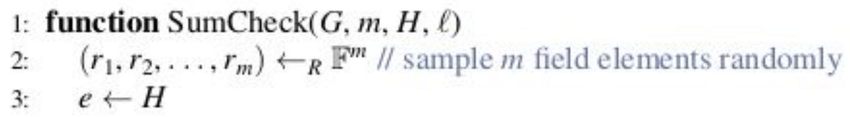
$$Q_{x,y}(\tau) = \sum_{u \in \{0,1\}^{3s}} \widetilde{F}_{x,y}(u) \cdot \tau^{\sum_{i=0}^{3s-1} u_i \cdot 2^i} = \sum_{u \in \{0,1\}^{3s}} \widetilde{F}_{x,y}(u) \cdot \prod_{i=0}^{3s-1} S_\tau(u,i) = \sum_{u \in \{0,1\}^{3s}} G_{x,y,\tau}(u)$$

Can now plug in to sumcheck protocol!

Number of variables: *$3s = 3log|C|$*

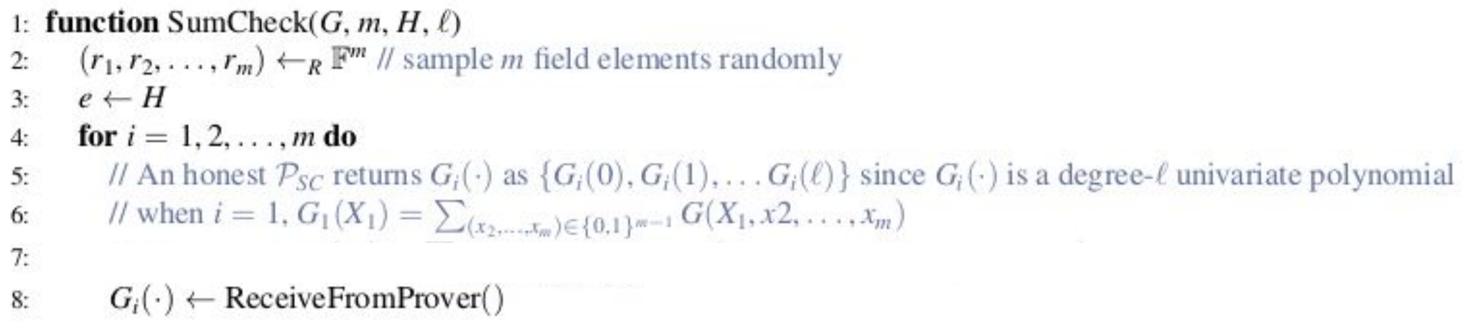
Degree: 3

Tool: Sumcheck Protocol



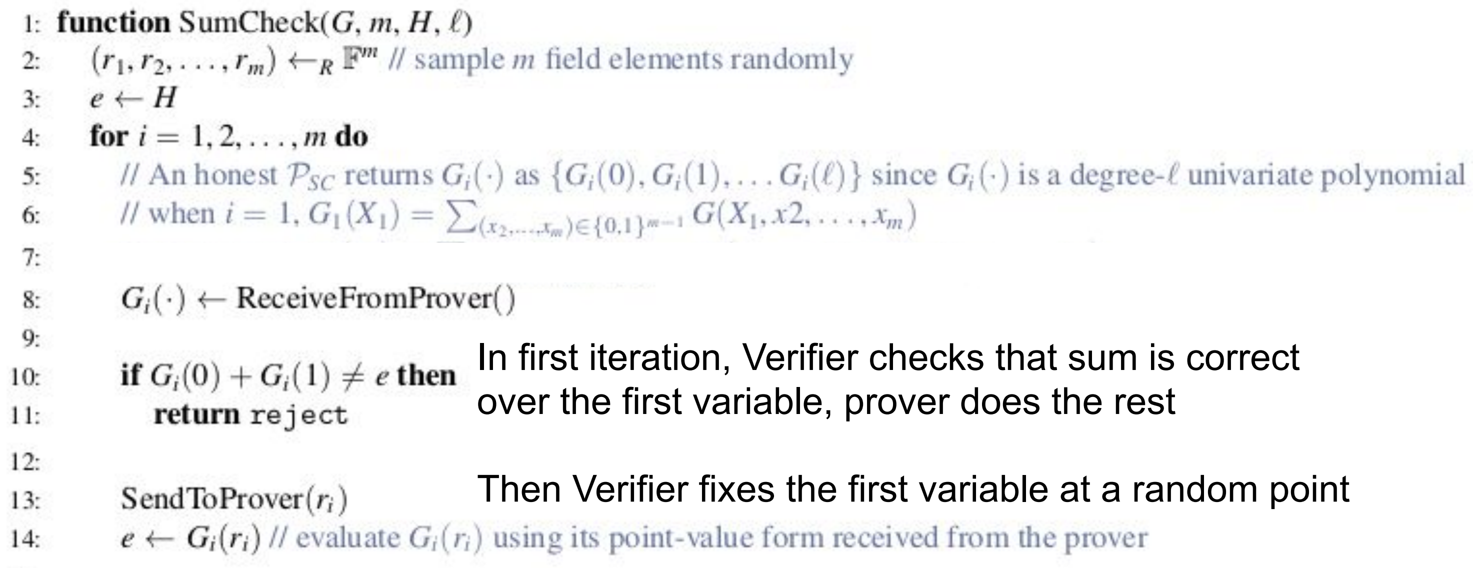
```
function SumCheck($G, m, H, \ell$)
    $(r_1, r_2, \ldots, r_m) \leftarrow_R \mathbb{F}^m$ // sample $m$ field elements randomly
    $e \leftarrow H$
```

Tool: Sumcheck Protocol



```
function SumCheck($G, m, H, \ell$)
    $(r_1, r_2, \ldots, r_m) \leftarrow_R \mathbb{F}^m$ // sample $m$ field elements randomly
    $e \leftarrow H$
    for $i = 1, 2, \ldots, m$ do
        // An honest $\mathcal{P}_{SC}$ returns $G_i(\cdot)$ as $\{G_i(0), G_i(1), \ldots G_i(\ell)\}$ since $G_i(\cdot)$ is a degree-$\ell$ univariate polynomial
        // when $i = 1$, $G_1(X_1) = \sum_{(x_2,\ldots,x_m)\in\{0,1\}^{m-1}} G(X_1, x2, \ldots, x_m)$

        $G_i(\cdot) \leftarrow$ ReceiveFromProver()
```

Tool: Sumcheck Protocol

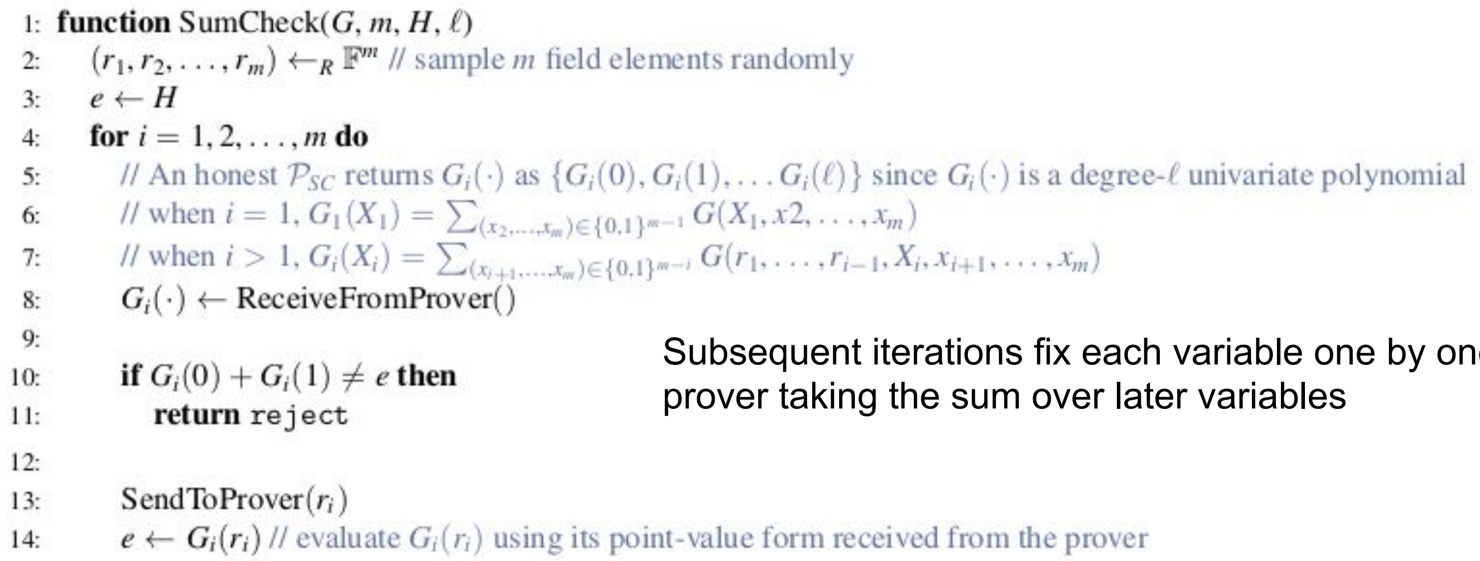


```
function SumCheck($G, m, H, \ell$)
    $(r_1, r_2, \ldots, r_m) \leftarrow_R \mathbb{F}^m$ // sample $m$ field elements randomly
    $e \leftarrow H$
    for $i = 1, 2, \ldots, m$ do
        // An honest $\mathcal{P}_{SC}$ returns $G_i(\cdot)$ as $\{G_i(0), G_i(1), \ldots G_i(\ell)\}$ since $G_i(\cdot)$ is a degree-$\ell$ univariate polynomial
        // when $i = 1$, $G_1(X_1) = \sum_{(x_2,\ldots,x_m)\in\{0,1\}^{m-1}} G(X_1, x2, \ldots, x_m)$

        $G_i(\cdot) \leftarrow$ ReceiveFromProver()

        if $G_i(0) + G_i(1) \neq e$ then
            return reject

        SendToProver($r_i$)
        $e \leftarrow G_i(r_i)$ // evaluate $G_i(r_i)$ using its point-value form received from the prover
```

In first iteration, Verifier checks that sum is correct over the first variable, prover does the rest

Then Verifier fixes the first variable at a random point

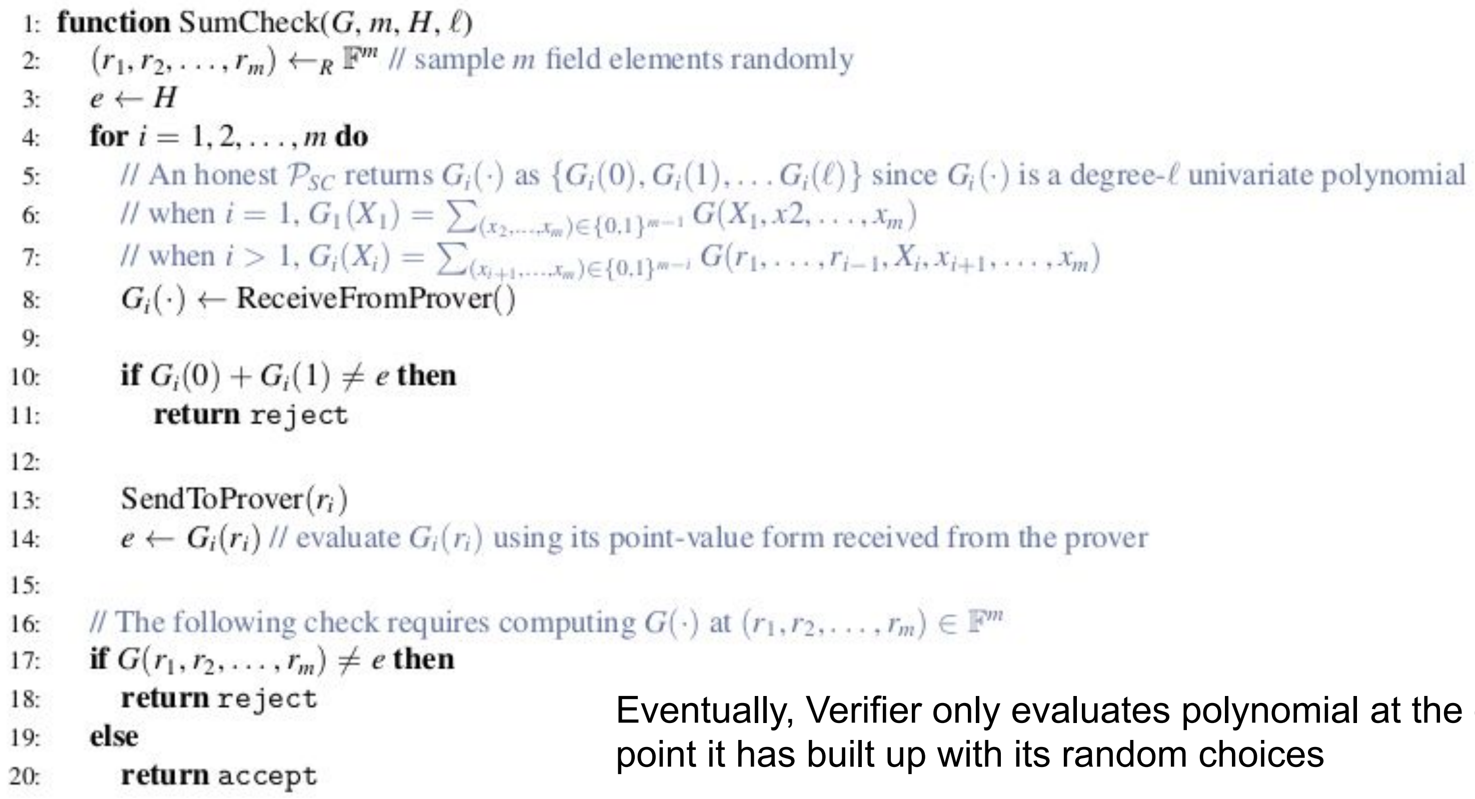
Tool: Sumcheck Protocol



```
function SumCheck($G, m, H, \ell$)
    $(r_1, r_2, \ldots, r_m) \leftarrow_R \mathbb{F}^m$ // sample $m$ field elements randomly
    $e \leftarrow H$
    for $i = 1, 2, \ldots, m$ do
        // An honest $\mathcal{P}_{SC}$ returns $G_i(\cdot)$ as $\{G_i(0), G_i(1), \ldots G_i(\ell)\}$ since $G_i(\cdot)$ is a degree-$\ell$ univariate polynomial
        // when $i = 1$, $G_1(X_1) = \sum_{(x_2,\ldots,x_m)\in\{0,1\}^{m-1}} G(X_1, x2, \ldots, x_m)$
        // when $i > 1$, $G_i(X_i) = \sum_{(x_{i+1},\ldots,x_m)\in\{0,1\}^{m-i}} G(r_1, \ldots, r_{i-1}, X_i, x_{i+1}, \ldots, x_m)$
        $G_i(\cdot) \leftarrow$ ReceiveFromProver()

        if $G_i(0) + G_i(1) \neq e$ then
            return reject

        SendToProver($r_i$)
        $e \leftarrow G_i(r_i)$ // evaluate $G_i(r_i)$ using its point-value form received from the prover
```

Subsequent iterations fix each variable one by one, with the prover taking the sum over later variables

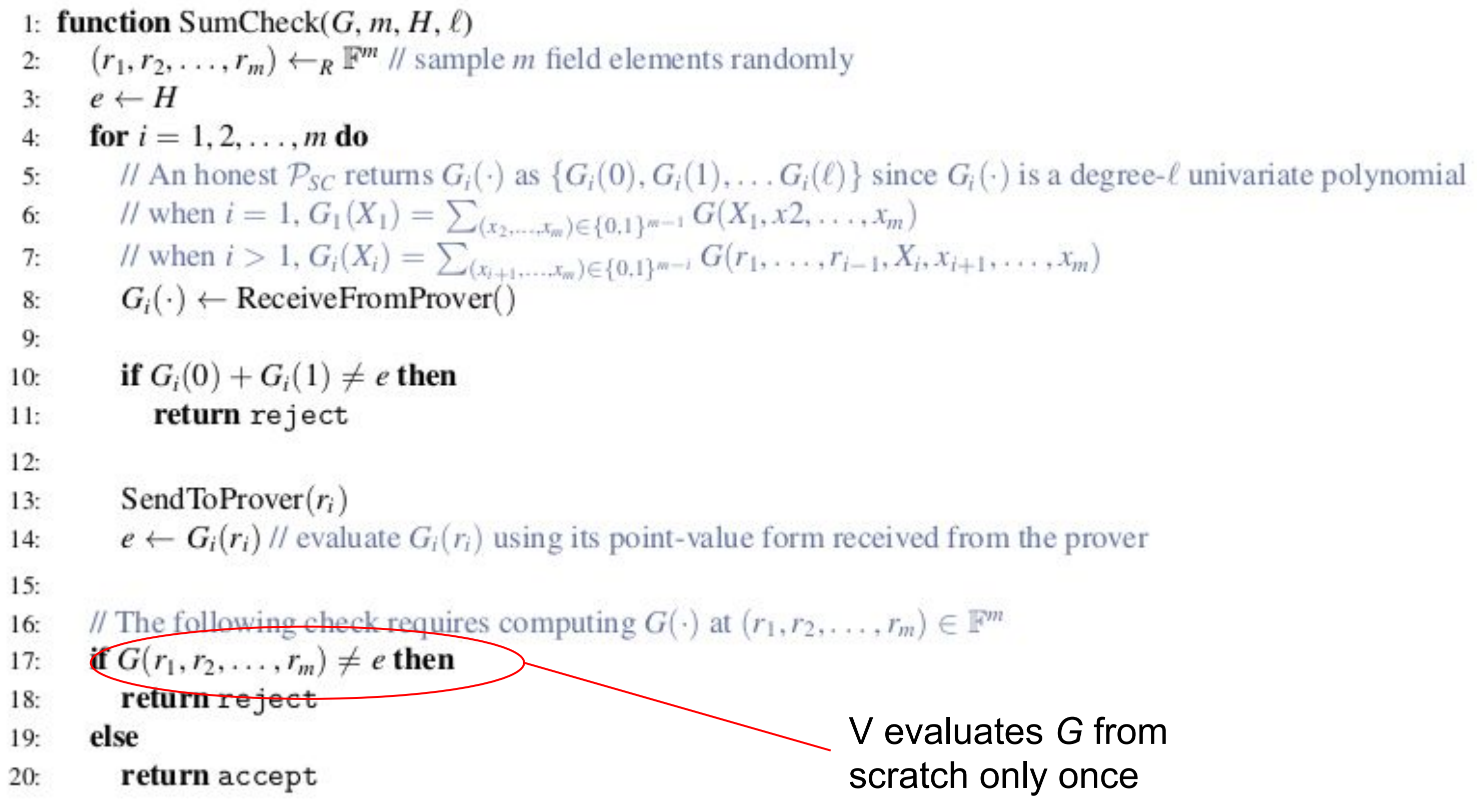
Tool: Sumcheck Protocol



```
function SumCheck($G, m, H, \ell$)
    $(r_1, r_2, \ldots, r_m) \leftarrow_R \mathbb{F}^m$ // sample $m$ field elements randomly
    $e \leftarrow H$
    for $i = 1, 2, \ldots, m$ do
        // An honest $\mathcal{P}_{SC}$ returns $G_i(\cdot)$ as $\{G_i(0), G_i(1), \ldots G_i(\ell)\}$ since $G_i(\cdot)$ is a degree-$\ell$ univariate polynomial
        // when $i = 1$, $G_1(X_1) = \sum_{(x_2,\ldots,x_m)\in\{0,1\}^{m-1}} G(X_1, x2, \ldots, x_m)$
        // when $i > 1$, $G_i(X_i) = \sum_{(x_{i+1},\ldots,x_m)\in\{0,1\}^{m-i}} G(r_1, \ldots, r_{i-1}, X_i, x_{i+1}, \ldots, x_m)$
        $G_i(\cdot) \leftarrow$ ReceiveFromProver()

        if $G_i(0) + G_i(1) \neq e$ then
            return reject

        SendToProver($r_i$)
        $e \leftarrow G_i(r_i)$ // evaluate $G_i(r_i)$ using its point-value form received from the prover

    // The following check requires computing $G(\cdot)$ at $(r_1, r_2, \ldots, r_m) \in \mathbb{F}^m$
    if $G(r_1, r_2, \ldots, r_m) \neq e$ then
        return reject
    else
        return accept
```

Eventually, Verifier only evaluates polynomial at the one point it has built up with its random choices

Tool: Sumcheck Protocol

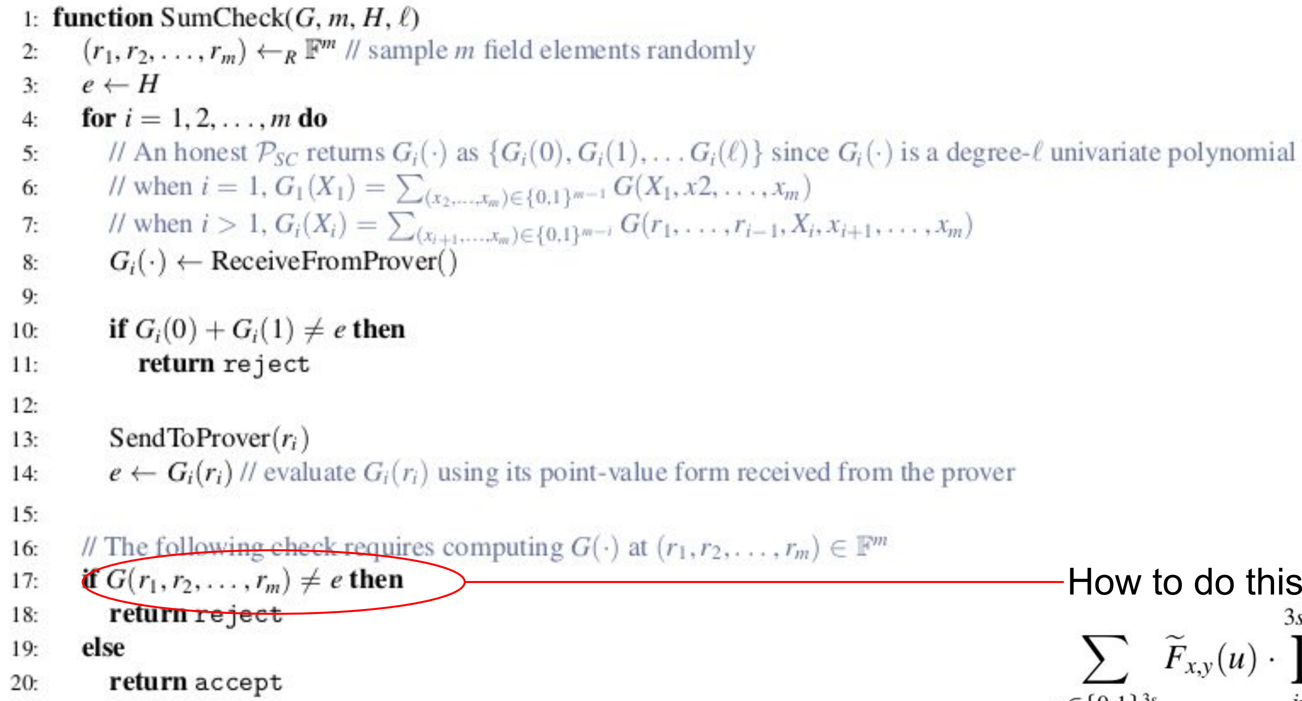


```
function SumCheck($G, m, H, \ell$)
    $(r_1, r_2, \ldots, r_m) \leftarrow_R \mathbb{F}^m$ // sample $m$ field elements randomly
    $e \leftarrow H$
    for $i = 1, 2, \ldots, m$ do
        // An honest $\mathcal{P}_{SC}$ returns $G_i(\cdot)$ as $\{G_i(0), G_i(1), \ldots G_i(\ell)\}$ since $G_i(\cdot)$ is a degree-$\ell$ univariate polynomial
        // when $i = 1$, $G_1(X_1) = \sum_{(x_2,\ldots,x_m)\in\{0,1\}^{m-1}} G(X_1, x2, \ldots, x_m)$
        // when $i > 1$, $G_i(X_i) = \sum_{(x_{i+1},\ldots,x_m)\in\{0,1\}^{m-i}} G(r_1, \ldots, r_{i-1}, X_i, x_{i+1}, \ldots, x_m)$
        $G_i(\cdot) \leftarrow$ ReceiveFromProver()

        if $G_i(0) + G_i(1) \neq e$ then
            return reject

        SendToProver($r_i$)
        $e \leftarrow G_i(r_i)$ // evaluate $G_i(r_i)$ using its point-value form received from the prover

    // The following check requires computing $G(\cdot)$ at $(r_1, r_2, \ldots, r_m) \in \mathbb{F}^m$
    if $G(r_1, r_2, \ldots, r_m) \neq e$ then
        return reject
    else
        return accept
```

V evaluates *G* from scratch only once

Spartan: Argument System (“back-end”)

Now verifier just needs to run sumcheck with *m=3s, l=3*



```
function SumCheck($G$, $m$, $H$, $\ell$)
    $(r_1, r_2, \dots, r_m) \leftarrow_R \mathbb{F}^m$ // sample $m$ field elements randomly
    $e \leftarrow H$
    for $i = 1, 2, \dots, m$ do
        // An honest $\mathcal{P}_{SC}$ returns $G_i(\cdot)$ as $\{G_i(0), G_i(1), \dots G_i(\ell)\}$ since $G_i(\cdot)$ is a degree-$\ell$ univariate polynomial
        // when $i = 1$, $G_1(X_1) = \sum_{(x_2,\dots,x_m)\in\{0,1\}^{m-1}} G(X_1, x2, \dots, x_m)$
        // when $i > 1$, $G_i(X_i) = \sum_{(x_{i+1},\dots,x_m)\in\{0,1\}^{m-i}} G(r_1, \dots, r_{i-1}, X_i, x_{i+1}, \dots, x_m)$
        $G_i(\cdot) \leftarrow$ ReceiveFromProver()

        if $G_i(0) + G_i(1) \neq e$ then
            return reject

        SendToProver($r_i$)
        $e \leftarrow G_i(r_i)$ // evaluate $G_i(r_i)$ using its point-value form received from the prover

    // The following check requires computing $G(\cdot)$ at $(r_1, r_2, \dots, r_m) \in \mathbb{F}^m$
    if $G(r_1, r_2, \dots, r_m) \neq e$ then
        return reject
    else
        return accept
```

How to do this?

$$\sum_{u\in\{0,1\}^{3s}} \widetilde{F}_{x,y}(u) \cdot \prod_{i=0}^{3s-1} S_\tau(u, i)$$

Spartan: Argument System (“back-end”)

Problematic part of *G*:

$$\begin{aligned}\widetilde{F}_{x,y}(u_1,u_2,u_3) = {} & \widetilde{\text{io}}(u_1,u_2,u_3)\cdot(\widetilde{I}_{x,y}(u_1)-\widetilde{Z}(u_1))+\\ & \widetilde{\text{add}}(u_1,u_2,u_3)\cdot(\widetilde{Z}(u_1)-(\widetilde{Z}(u_2)+\widetilde{Z}(u_3)))+\\ & \widetilde{\text{mul}}(u_1,u_2,u_3)\cdot(\widetilde{Z}(u_2)-\widetilde{Z}(u_2)\cdot\widetilde{Z}(u_3))\end{aligned}$$

Naive solution: Prover sends *Z* to verifier, verifier computes F on its own.

Spartan: Argument System (“back-end”)

Problematic part of *G*: $$\begin{aligned}\widetilde{F}_{x,y}(u_1,u_2,u_3) = {} & \widetilde{\mathrm{io}}(u_1,u_2,u_3)\cdot(\widetilde{I}_{x,y}(u_1)-\widetilde{Z}(u_1))+\\ & \widetilde{\mathrm{add}}(u_1,u_2,u_3)\cdot(\widetilde{Z}(u_1)-(\widetilde{Z}(u_2)+\widetilde{Z}(u_3)))+\\ & \widetilde{\mathrm{mul}}(u_1,u_2,u_3)\cdot(\widetilde{Z}(u_2)-\widetilde{Z}(u_2)\cdot\widetilde{Z}(u_3))\end{aligned}$$

Naive solution: Prover sends *Z* to verifier, verifier computes F on its own.

Problem: this is not succinct!

1. Communication cost linear in |*C*|
2. Verifier computation linear in |*C*|

Spartan: Argument System (“back-end”)

Problem 1: reduce communication cost to sublinear in |*C*|

Solution: polynomial commitment to multilinear polynomial *Z*

Prover commits to *Z* and sends openings of *Z* at the desired points u_1, u_2, u_3

Verifier can compute other parts of *G* on its own (they only depend on public *C*)

$$
\begin{aligned}
&\widetilde{\mathrm{io}}(u_1,u_2,u_3)\cdot(\widetilde{I}_{x,y}(u_1)-\widetilde{Z}(u_1))+\\
&\widetilde{\mathrm{add}}(u_1,u_2,u_3)\cdot(\widetilde{Z}(u_1)-(\widetilde{Z}(u_2)+\widetilde{Z}(u_3)))+\\
&\widetilde{\mathrm{mul}}(u_1,u_2,u_3)\cdot(\widetilde{Z}(u_2)-\widetilde{Z}(u_2)\cdot\widetilde{Z}(u_3))
\end{aligned}
$$

Spartan: Argument System (“back-end”)

Problem 2: reduce Verifier computation cost to sublinear in |*C*|

All these functions require work linear in |*C*| to compute

$$\begin{aligned}
&\boxed{\widetilde{\mathrm{io}}(u_1,u_2,u_3)}\cdot\boxed{(\widetilde{I}_{x,y}(u_1)}-\widetilde{Z}(u_1))+\\
&\boxed{\widetilde{\mathrm{add}}(u_1,u_2,u_3)}\cdot(\widetilde{Z}(u_1)-(\widetilde{Z}(u_2)+\widetilde{Z}(u_3)))+\\
&\boxed{\widetilde{\mathrm{mul}}(u_1,u_2,u_3)}\cdot(\widetilde{Z}(u_2)-\widetilde{Z}(u_2)\cdot\widetilde{Z}(u_3))
\end{aligned}$$

Spartan: Argument System (“back-end”)

Problem 2: reduce Verifier computation cost to sublinear in $|C|$

All these functions require work linear in $|C|$ to compute

$$\begin{aligned}&\boxed{\widetilde{\mathrm{io}}(u_1, u_2, u_3)} \cdot (\boxed{\widetilde{I}_{x,y}(u_1)} - \widetilde{Z}(u_1)) + \\ &\boxed{\widetilde{\mathrm{add}}(u_1, u_2, u_3)} \cdot (\widetilde{Z}(u_1) - (\widetilde{Z}(u_2) + \widetilde{Z}(u_3))) + \\ &\boxed{\widetilde{\mathrm{mul}}(u_1, u_2, u_3)} \cdot (\widetilde{Z}(u_2) - \widetilde{Z}(u_2) \cdot \widetilde{Z}(u_3))\end{aligned}$$

Solution: *Verifier* produces polynomial commitments to all circled functions itself!

Prover only needs to supply decommitments at evaluation points

Verifier can verify that they are valid evaluations for the polynomial it committed to

Spartan: Summary

Front-end:

Convert arithmetic circuit into multilinear polynomial such that sum over m-dimensional boolean hypercube is 0.

Back-end:

Run sumcheck protocol using polynomial commitments to save on communication and computation costs.

Sonic & AuroraLight

Sonic & AuroraLight

Sonic

Pros: Updatable trusted setup, constant size proofs

Cons: Still has setup

AuroraLight

Pros: Prover 2x faster, SRS 6x smaller than Sonic

Cons: Still has setup, only works for batched or parallelized proofs

Both protocols convert arithmetic circuit to polynomial such that the constant coefficient is zero if and only if the Prover has a satisfying assignment.

AuroraLight

Start with standard r1cs format: $(a_i \cdot x)(b_i \cdot x) - (c_i \cdot x) = 0$

AuroraLight

Start with standard r1cs format: $(\boxed{a_i} \cdot x)(\boxed{b_i} \cdot x) - (\boxed{c_i} \cdot x) = 0$

Vectors a_i, b_i, and c_i represent structure of circuit

AuroraLight

Start with standard r1cs format: $(a_i \cdot x)(b_i \cdot x) - (c_i \cdot x) = 0$

Modify to “flattened” format:

1. $y_i \cdot z_i - (c_i \cdot x) = 0.$
2. $y_i - (a_i \cdot x) = 0.$
3. $z_i - (b_i \cdot x) = 0.$

AuroraLight

Start with standard r1cs format: $(a_i \cdot x)(b_i \cdot x) - (c_i \cdot x) = 0$

Modify to “flattened” format:

1. $y_i \cdot z_i - (c_i \cdot x) = 0.$
2. $y_i - (a_i \cdot x) = 0.$
3. $z_i - (b_i \cdot x) = 0.$

Goal: build a polynomial that has a zero constant term if and only if there is a satisfying assignment to the constraints, so we can use the following math fact.

Lemma 4.1. *Fix any $f \in \mathbb{F}_{<n}[X]$. Then for any multiplicative subgroup $H \subset \mathbb{F}$ with $|H| = n$,*

$$\sum_{a \in H} f(a) = 0$$

if and only if f has a zero constant term.

AuroraLight

Define functions W,Y,Z$\in$***F**[X]* of degree <n (circuit size) s.t. for each $h_i \in H$, $W(i)=x_i$, $Y(i)=y_i$, *and* $Z(i)=z_i$. Prover starts by committing to these three polynomials.

The following sum is zero iff the values of x satisfy the constraints, where *r,r',r'' are* random values chosen by the Verifier.

$$\sum_{i\in H} r^i (y_i z_i - c_i \cdot x) + \sum_{i\in H} r'^i (y_i - a_i \cdot x) + \sum_{i\in H} r''^i (z_i - b_i \cdot x)$$

AuroraLight

Define functions W,Y,Z ∈ ***F**[X]* of degree <n (circuit size) s.t. for each $h_i \in H$, $W(i)=x_i$, $Y(i)=y_i$, *and* $Z(i)=z_i$. Prover starts by committing to these three polynomials.

The following sum is zero iff the values of x satisfy the constraints, where *r,r',r'' are* random values chosen by the Verifier.

$$\sum_{i \in H} r^i (y_i z_i - c_i \cdot x) + \sum_{i \in H} r'^i (y_i - a_i \cdot x) + \sum_{i \in H} r''^i (z_i - b_i \cdot x)$$

AuroraLight

Define functions W,Y,Z∈***F**[X]* of degree <n (circuit size) s.t. for each $h_i \in H$, $W(i)=x_i$, $Y(i)=y_i$, *and* $Z(i)=z_i$. Prover starts by committing to these three polynomials.

The following sum is zero iff the values of x satisfy the constraints, where *r,r',r" are* random values chosen by the Verifier.

$$\sum_{i\in H} r^i(y_i z_i - c_i\cdot x) + \sum_{i\in H} r'^i(y_i - a_i\cdot x) + \sum_{i\in H} r''^i(z_i - b_i\cdot x)$$

$$= \sum_{i\in H} r^i y_i z_i + \sum_{i\in H} r'^i y_i + \sum_{i\in H} r''^i z_i + \sum_{i\in H} \alpha_i x_i + \alpha_0 = \sum_{i\in H} D(i)$$

α_i are some expression of the other variables

AuroraLight

Define functions W,Y,Z∈***F**[X]* of degree <n (circuit size) s.t. for each $h_i \in H$, $W(i)=x_i$, $Y(i)=y_i$, *and* $Z(i)=z_i$. Prover starts by committing to these three polynomials.

The following sum is zero iff the values of x satisfy the constraints, where *r,r',r" are* random values chosen by the Verifier.

$$\sum_{i\in H} r^i(y_i z_i - c_i\cdot x) + \sum_{i\in H} r'^i(y_i - a_i\cdot x) + \sum_{i\in H} r''^i(z_i - b_i\cdot x)$$

$$= \sum_{i\in H} r^i y_i z_i + \sum_{i\in H} r'^i y_i + \sum_{i\in H} r''^i z_i + \sum_{i\in H} \alpha_i x_i + \alpha_0 = \sum_{i\in H} D(i)$$

$$D := \underline{R}\cdot Y\cdot Z + \underline{R'}\cdot Y + \underline{R''}\cdot Z + \underline{Q}\cdot W + \alpha_0/n.$$

R, R', R'', and Q are polynomials that give the corresponding values of r, r', r'', and α

AuroraLight

Define functions W,Y,Z∈***F**[X]* of degree <n (circuit size) s.t. for each $h_i \in H$, $W(i)=x_i$, $Y(i)=y_i$, *and* $Z(i)=z_i$. Prover starts by committing to these three polynomials.

The following sum is zero iff the values of x satisfy the constraints, where *r,r',r" are* random values chosen by the Verifier.

$$\sum_{i\in H} r^i(y_i z_i - c_i \cdot x) + \sum_{i\in H} r'^i(y_i - a_i \cdot x) + \sum_{i\in H} r''^i(z_i - b_i \cdot x)$$

$$= \sum_{i\in H} r^i y_i z_i + \sum_{i\in H} r'^i y_i + \sum_{i\in H} r''^i z_i + \sum_{i\in H} \alpha_i x_i + \alpha_0 = \sum_{i\in H} D(i)$$

$$D := R \cdot Y \cdot Z + R' \cdot Y + R'' \cdot Z + Q \cdot W + \alpha_0/n.$$

Now we can check that the sum of this polynomial over *H* is 0.

AuroraLight

Now, instead of taking the actual sum, we will try to verify that *D* sums to zero over *H* another way.

For $a \in H$, define $Z_H(X) := \prod_{a \in H}(X - a)$

AuroraLight

Now, instead of taking the actual sum, we will try to verify that *D* sums to zero over *H* another way.

For *a*∈*H*, define $Z_H(X) := \prod_{a \in H}(X - a)$

Then, use polynomial long division to get $D(X) = g(X) \cdot Z_H(X) + X \cdot f(X)$

AuroraLight

Now, instead of taking the actual sum, we will try to verify that *D* sums to zero over *H* another way.

For $a \in H$, define $Z_H(X) := \prod_{a \in H}(X - a)$

Then, use polynomial long division to get $D(X) = g(X) \cdot Z_H(X) + X \cdot f(X)$

D can only be written in this form if there is no constant term

AuroraLight

Now, instead of taking the actual sum, we will try to verify that *D* sums to zero over *H* another way.

For *a*∈*H*, define $Z_H(X) := \prod_{a \in H}(X - a)$

Then, use polynomial long division to get $D(X) = g(X) \cdot Z_H(X) + \bigcirc\!\!\!\!\!X \cdot f(X)$

D can only be written in this form if there is no constant term

$g(X) \cdot Z_H(X)$ is 0 at every point in *H* because every *a*∈*H* is a root.

$X \cdot f(X)$ is 0 at every point in *H* because it has no constant term.

Thus if *D* can be represented this way, we know that it has no constant term

AuroraLight

Final Protocol

V chooses randomness *r,r’,r”* to determine polynomials *R,R’,R”*

$$D := R \cdot Y \cdot Z + R' \cdot Y + R'' \cdot Z + Q \cdot W + \alpha_0/n.$$

AuroraLight

Final Protocol

V chooses randomness *r,r',r"* to determine polynomials *R,R',R"*

P commits to polynomials *W,Y,Z,g,f* used to compute *D*

$$D := R \cdot Y \cdot Z + R' \cdot Y + R'' \cdot Z + Q \cdot W + \alpha_0/n.$$

AuroraLight

Final Protocol

V chooses randomness *r,r',r"* to determine polynomials *R,R',R"*

P commits to polynomials *W,Y,Z,g,f* used to compute *D*

V chooses random evaluation point *z*

$$D := R \cdot Y \cdot Z + R' \cdot Y + R'' \cdot Z + Q \cdot W + \alpha_0/n.$$

AuroraLight

Final Protocol

V chooses randomness *r,r’,r”* to determine polynomials *R,R’,R”*

P commits to polynomials *W,Y,Z,g,f* used to compute *D*

V chooses random evaluation point *z*

P opens commitments to *W(z), Y(z), Z(z), g(z), f(z)*

$$D := R \cdot Y \cdot Z + R' \cdot Y + R'' \cdot Z + Q \cdot W + \alpha_0/n.$$

AuroraLight

Final Protocol

V chooses randomness *r,r',r''* to determine polynomials *R,R',R''*

P commits to polynomials *W,Y,Z,g,f* used to compute *D*

V chooses random evaluation point *z*

P opens commitments to *W(z), Y(z), Z(z), g(z), f(z)*

V computes *D* and accepts iff $D(z) = g(z) \cdot Z_H(z) + z \cdot f(z)$

$$D := R \cdot Y \cdot Z + R' \cdot Y + R'' \cdot Z + Q \cdot W + \alpha_0/n.$$

AuroraLight

Final Protocol

V chooses randomness *r,r’,r”* to determine polynomials *R,R’,R”*

P commits to polynomials *W,Y,Z,g,f* used to compute *D*

V chooses random evaluation point *z*

P opens commitments to *W(z), Y(z), Z(z), g(z), f(z)*

V computes *D* and accepts iff $D(z) = g(z) \cdot Z_H(z) + z \cdot f(z)$

$$D := R \cdot Y \cdot Z + R' \cdot Y + R'' \cdot Z + Q \cdot W + \alpha_0/n.$$

Verifier computation is only succinct if many proofs are generated in parallel because computing *R, R’, R”*, and *Q* takes time linear in circuit size

AuroraLight with Sonic Helpers

How to make Verifier computation succinct for batches of proofs not generated together in parallel?

AuroraLight with Sonic Helpers

How to make Verifier computation succinct for batches of proofs not generated together in parallel?

Idea (Sonic): Have untrusted “helpers” (e.g. cryptocurrency miners) compute expensive parts (*R*, *R’*, *R”*, and *Q*), and prove to the Verifier that they have done this honestly.

AuroraLight with Sonic Helpers

How to make Verifier computation succinct for batches of proofs not generated together in parallel?

Idea (Sonic): Have untrusted “helpers” (e.g. cryptocurrency miners) compute expensive parts (R, R', R'', and Q), and prove to the Verifier that they have done this honestly.

Sonic helper applies to 2-variable polynomials, so we rewrite our polynomials to fit this form, e.g. $R=R(X,Y)$ where X represents the choice of r and Y represents the choice of z.

The Sonic Helper

Helper assists verifying that $R(\bullet,\bullet)$ is evaluated correctly at points $(y_1,z_1),\ldots,(y_m,z_m)$

Helper Verifier

The Sonic Helper

Helper assists verifying that R(•,•) is evaluated correctly at points $(y_1,z_1),...,(y_m,z_m)$

Helper Verifier

For $i \in [m]$, Commit to $c_{1i} = R(\bullet, y_i)$
Open $c_{1i}(z_i)$

$\longrightarrow$

The Sonic Helper

Helper assists verifying that $R(\bullet,\bullet)$ is evaluated correctly at points $(y_1,z_1),\ldots,(y_m,z_m)$

Helper | Verifier

For $i \in [m]$, Commit to $c_{1i} = R(\bullet, y_i)$
Open $c_{1i}(z_i)$

$$\xrightarrow{\hspace{6cm}}$$

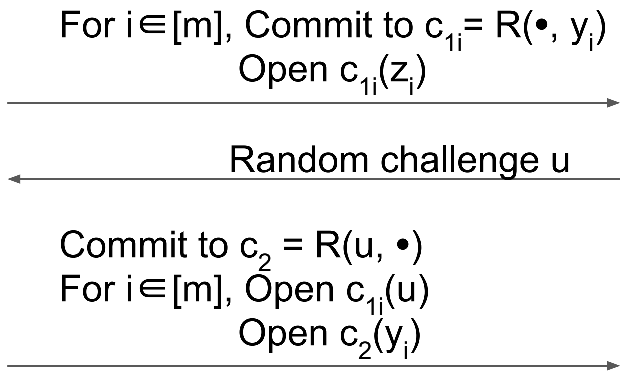
Random challenge u

$$\xleftarrow{\hspace{6cm}}$$

The Sonic Helper

Helper assists verifying that $R(\bullet,\bullet)$ is evaluated correctly at points $(y_1,z_1),\ldots,(y_m,z_m)$

<u>Helper</u> <u>Verifier</u>



For $i \in [m]$, Commit to $c_{1i} = R(\bullet, y_i)$

Open $c_{1i}(z_i)$

Random challenge u

Commit to $c_2 = R(u, \bullet)$

For $i \in [m]$, Open $c_{1i}(u)$

Open $c_2(y_i)$

The Sonic Helper

Helper assists verifying that $R(\bullet,\bullet)$ is evaluated correctly at points $(y_1,z_1),\ldots,(y_m,z_m)$

<u>Helper</u> <u>Verifier</u>

For $i \in [m]$, Commit to $c_{1i} = R(\bullet, y_i)$
Open $c_{1i}(z_i)$

$$\xrightarrow{\hspace{6cm}}$$

Random challenge u

$$\xleftarrow{\hspace{6cm}}$$

Commit to $c_2 = R(u, \bullet)$
For $i \in [m]$, Open $c_{1i}(u)$
Open $c_2(y_i)$

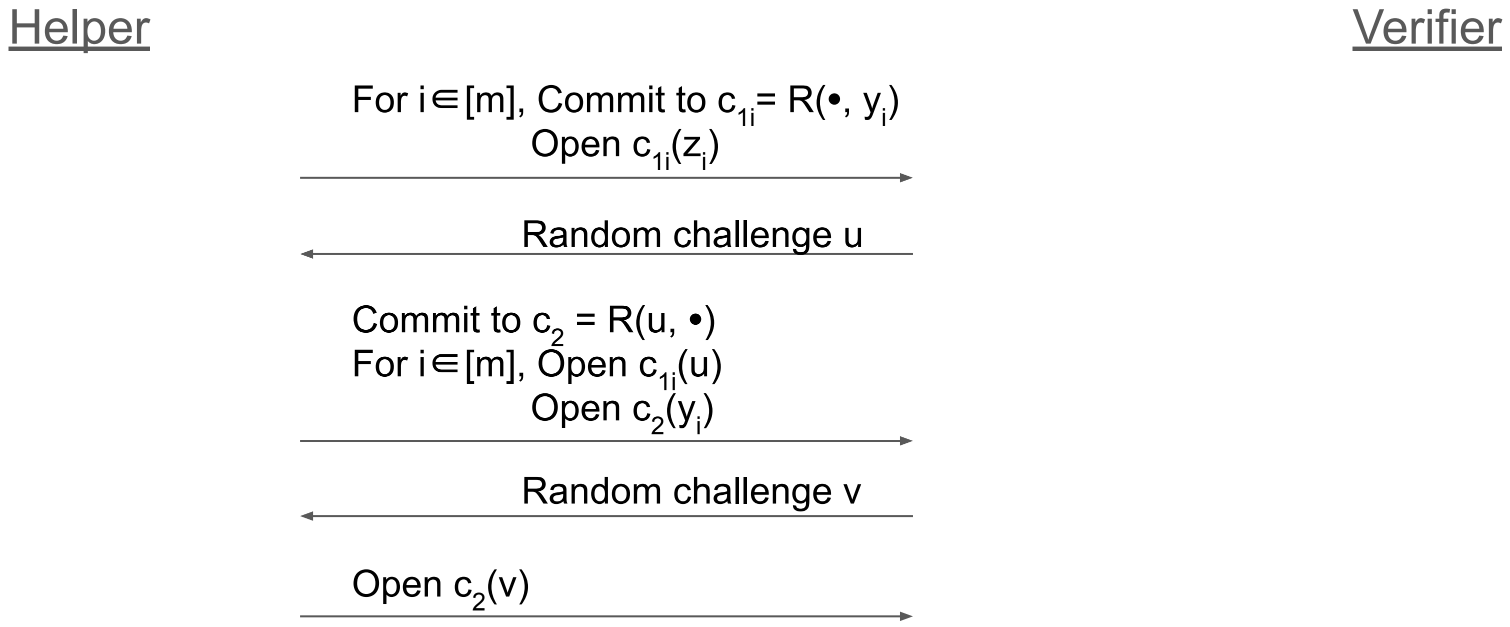
$$\xrightarrow{\hspace{6cm}}$$

Random challenge v

$$\xleftarrow{\hspace{6cm}}$$

The Sonic Helper

Helper assists verifying that R(•,•) is evaluated correctly at points $(y_1,z_1),\ldots,(y_m,z_m)$


Helper
Verifier
For $i \in [m]$, Commit to $c_{1i} = R(\bullet, y_i)$
Open $c_{1i}(z_i)$
Random challenge u
Commit to $c_2 = R(u, \bullet)$
For $i \in [m]$, Open $c_{1i}(u)$
Open $c_2(y_i)$
Random challenge v
Open $c_2(v)$

The Sonic Helper

Helper assists verifying that $R(\bullet,\bullet)$ is evaluated correctly at points $(y_1,z_1),\ldots,(y_m,z_m)$

<u>Helper</u>

For $i \in [m]$, Commit to $c_{1i} = R(\bullet, y_i)$
Open $c_{1i}(z_i)$

$\longrightarrow$

Random challenge u

$\longleftarrow$

Commit to $c_2 = R(u, \bullet)$
For $i \in [m]$, Open $c_{1i}(u)$
Open $c_2(y_i)$

$\longrightarrow$

Random challenge v

$\longleftarrow$

Open $c_2(v)$

$\longrightarrow$

<u>Verifier</u>

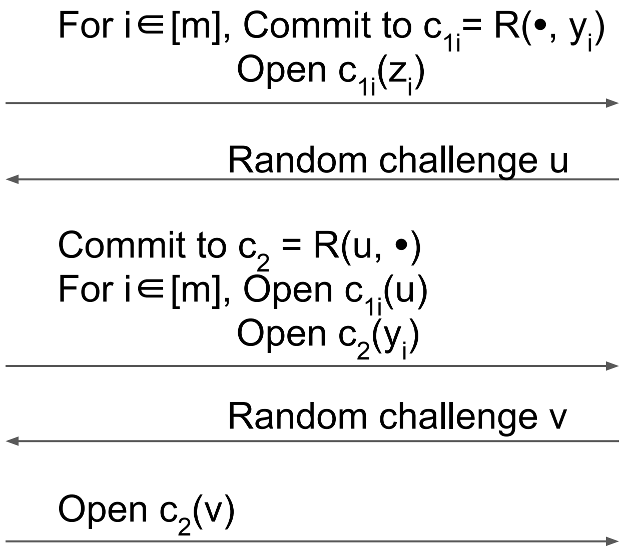
1. Check that all commitments verify
2. Check that $c_{1i}(u)=c_2(y_i)$
3. Compute $R(u,v)$ and check that $c_2(v)=R(u,v)$

The Sonic Helper

Helper assists verifying that $R(\bullet,\bullet)$ is evaluated correctly at points $(y_1,z_1),\ldots,(y_m,z_m)$

<u>Helper</u>

<u>Verifier</u>


For $i \in [m]$, Commit to $c_{1i} = R(\bullet, y_i)$
Open $c_{1i}(z_i)$
Random challenge u
Commit to $c_2 = R(u, \bullet)$
For $i \in [m]$, Open $c_{1i}(u)$
Open $c_2(y_i)$
Random challenge v
Open $c_2(v)$

1. Check that all commitments verify
2. Check that $c_{1i}(u)=c_2(y_i)$
3. Compute $R(u,v)$ and check that $c_2(v)=R(u,v)$

Verifier only evaluates R once, no matter how many proofs are batched!

Sonic

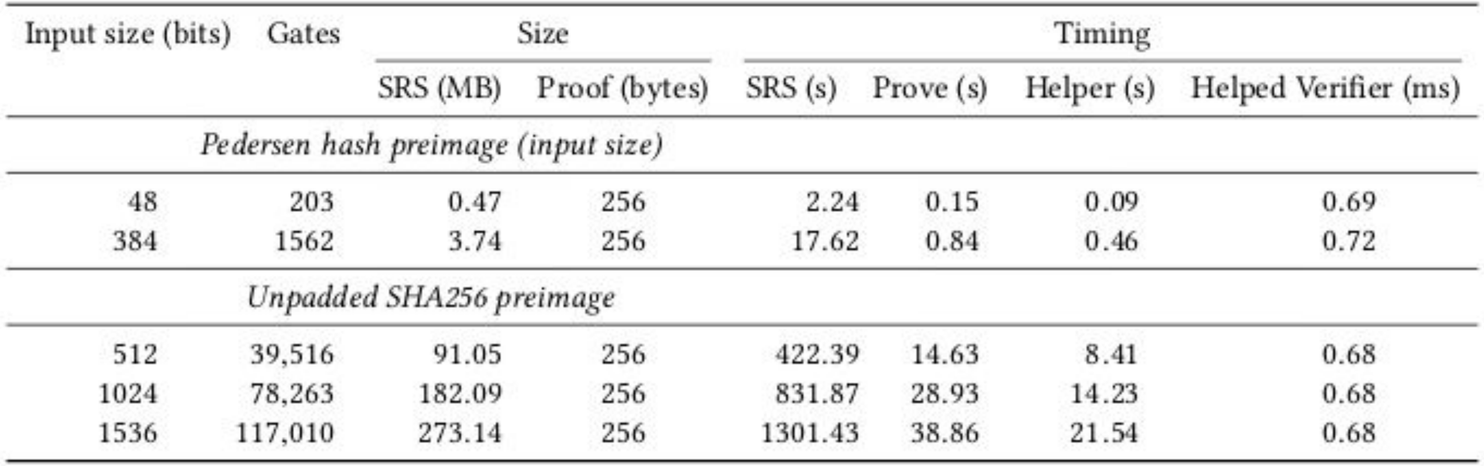
Also relies on polynomials with no constant coefficient

Guarantees this by using polynomial commitment scheme that does not allow committing to a constant term

Has 2 modes:

1. Helped verification: uses untrusted helpers to speed up verification (implemented)
2. Fully succinct mode (AuroraLight does not have such a mode)

Sonic Performance



Input size (bits)	Gates	Size		Timing			
		SRS (MB)	Proof (bytes)	SRS (s)	Prove (s)	Helper (s)	Helped Verifier (ms)
Pedersen hash preimage (input size)							
48	203	0.47	256	2.24	0.15	0.09	0.69
384	1562	3.74	256	17.62	0.84	0.46	0.72
Unpadded SHA256 preimage							
512	39,516	91.05	256	422.39	14.63	8.41	0.68
1024	78,263	182.09	256	831.87	28.93	14.23	0.68
1536	117,010	273.14	256	1301.43	38.86	21.54	0.68

Table 4: Sonic's efficiency in proving knowledge of x such that $H(x) = y$ for different sizes of x. Numbers are given to two significant digits. The first rows are for the Pedersen hash function and the final rows are for SHA256. "Helper" and "Helped Verifier" are the marginal cost of aggregating and verifying an additional proof assuming that the helper has been run. These are calculated by batch-verifying 100 proofs, subtracting the cost to verify one, and dividing by 99.

Takeaways

High level approach for this kind of SNARK construction:

1. Convert program to arithmetic circuit
2. **Convert arithmetic circuit to polynomial with special property**
3. **Build an argument to prove something about the polynomial**
4. Add on other features generically

Improved polynomial commitments → Improved zk-SNARKs