Revisiting accepted wisdom in SNARK design

Justin Thaler

Georgetown University and a16z crypto research

Talk Outline

1. Survey of SNARK design.
2. What needs to change for scalability.
3. Whirlwind overview of recent developments.
   - Lasso [STW 2023]
   - Jolt [AST 2023]
   - BabySpartan [ST 2023]
   - Binius [DP 2023]

SNARKs: What are they

- Prover P claims to know witness w satisfying some property.
  - For example: A preimage w for a designated output y of a cryptographic hash function h.
    - A w such that $h(w) = y$.

Trivial proof: P sends w to V, who directly checks it satisfies the claimed property.

A SNARK (Succinct Non-interactive ARgument of Knowledge) achieves the same, but with better costs to V.

- SNARK proof π must be shorter than the witness w.
  - This is what "succinct" means.
- Ideally, checking π is faster than checking w directly.
  - Called "work-saving" for V.

SNARKs: What are they

- Prover P claims to know witness w satisfying some property.
  - For example: A preimage w for a designated output y of a cryptographic hash function h.
    - A w such that $h(w) = y$.

- Trivial proof: P sends w to V, who directly checks it satisfies the claimed property.

A SNARK (Succinct Non-interactive ARgument of Knowledge) achieves the same, but with better costs to V.

SNARK proof π must be shorter than the witness w.

This is what "succinct" means.

Ideally, checking π is faster than checking w directly.

Called "work-saving" for V.

SNARKs: What are they

- Prover P claims to know witness w satisfying some property.
  - For example: A preimage w for a designated output y of a cryptographic hash function h.
    - A w such that $h(w) = y$.
- Trivial proof: P sends w to V, who directly checks it satisfies the claimed property.
- A SNARK (**S**uccinct **N**on-interactive **AR**gument of **K**nowledge) achieves the same, but with better costs to V.
  - SNARK proof π **must** be shorter than the witness w.
    - This is what "succinct" means.
  - Ideally, checking π is faster than checking w directly.
    - Called "work-saving" for V.

SNARK design details

General-Purpose SNARKs: Standard Paradigm

- Start with a computer program ψ written in high-level programming language (C, Java, etc.).
  - Hash pre-image example:
    - Program takes w as input, applies h to it, checks that output equals y.

Step 1: Turn ψ into an equivalent model amenable to probabilistic checking.

Typically some variant of circuit-satisfiability.

The "circuit" is called an intermediate representation (IR).

Examples: arithmetic circuits, R1CS, AIR, "Plonkish IR".

Called the Front End of the system.

Step 2: Run a SNARK for circuit-satisfiability/R1CS/AIR/Plonkish/etc.

General-Purpose SNARKs: Standard Paradigm

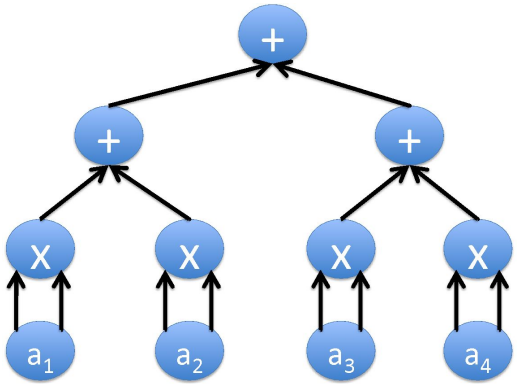
- Start with a computer program ψ written in high-level programming language (C, Java, etc.).
  - Hash pre-image example:
    - Program takes w as input, applies h to it, checks that output equals y.
- Step 1: Turn ψ into an equivalent model amenable to probabilistic checking.
  - Typically some variant of circuit-satisfiability.
    - The "circuit" is called an **intermediate representation (IR).**
    - Examples: arithmetic circuits, R1CS, AIR, "Plonkish IR", CCS.
  - Called the **Front End** of the system.
- Step 2: Run a SNARK for circuit-satisfiability/R1CS/AIR/Plonkish/etc.

General-Purpose SNARKs: Standard Paradigm

- Start with a computer program ψ written in high-level programming language (C, Java, etc.).
  - Hash pre-image example:
    - Program takes w as input, applies h to it, checks that output equals y.
- Step 1: Turn ψ into an equivalent model amenable to probabilistic checking.
  - Typically some variant of circuit-satisfiability.
    - The "circuit" is called an **intermediate representation (IR).**
    - Examples: arithmetic circuits, R1CS, AIR, "Plonkish IR", CCS.
  - Called the **Front End** of the system.
- Step 2: Run a SNARK for circuit-satisfiability/R1CS/AIR/Plonkish/etc.

  Called the **backend** of the system.

Front End



P and V run SNARK (back end) for circuit-satisfiability.

Backends

Key Paradigm for Backend Design

1. Give a "**polynomial IOP**" for R1CS or circuit-satisfiability.
2. Combine with **polynomial commitment scheme** to get succinct interactive argument.
3. Apply Fiat-Shamir transformation to render non-interactive.

Key Paradigm for Backend Design

1. Give a "**polynomial IOP**" for R1CS or circuit-satisfiability.
2. Combine with **polynomial commitment scheme** to get succinct interactive argument.
3. Apply Fiat-Shamir transformation to render non-interactive.

- This is how all SNARKs are designed other than "linear-PCP based" ones (GGPR13, Pinocchio, Groth16, etc.) and "folding-based" ones (Nova, etc.)
  - Even linear-PCP based SNARKs use techniques **very** similar to certain polynomial commitments (KZG).

Polynomial IOPs: Two classes

1. Constant-round, uses univariate polynomials
   - Plonk, Marlin: targeted at *non-uniform* computation (arbitrary circuits)
   - STARKs: targeted at *uniform* computation (e.g., CPU execution)

- Many-round, uses multivariate polynomials
  - Based on the **sum-check protocol**.
  - Spartan, Hyperplonk, Lasso, Jolt, BabySpartan, etc.

Polynomial Commitment Schemes: Three Classes

1. Based on pairings (**not** transparent **nor** post-quantum).
   - e.g., KZG10, PST13, Zeromorph
   - Proof size can be **constant**.

2. Based on discrete logarithm (transparent, **not** post-quantum).
   - Examples: IPA/Bulletproofs, Hyrax, Dory.
     - Proof size can be **logarithmic**.

3. Based on IOPs + hashing (transparent **and** post-quantum)
   - e.g., FRI, Ligero, Brakedown.
     - Proof size can be **polylogarithmic.**
     - $O(\lambda \log^2(n)) \gg O(\log^3(n))$ hash evaluations.

Polynomial Commitment Schemes: Three Classes

1. Based on pairings (**not** transparent **nor** post-quantum).
   - e.g., KZG10, PST13, Zeromorph
   - Proof size can be **constant**.

2. Based on discrete logarithm (transparent, **not** post-quantum).
   - Examples: IPA/Bulletproofs, Hyrax, Dory.
     - Proof size can be **logarithmic**.

3. Based on IOPs + hashing (transparent **and** post-quantum)
   - e.g., FRI, Ligero, Brakedown.
     - Proof size can be **polylogarithmic.**
       - $O\left(\lambda \log^2(n)\right) \gg O\left(\log^3(n)\right)$ hash evaluations.

SNARK performance

Performance bottlenecks

- Today's most popular SNARKs are optimized for **verifier** costs at the expense of **prover** costs.

- But the key scalability bottleneck is P time.

Performance bottlenecks

- Today's most popular SNARKs are optimized for **verifier** costs at the expense of **prover** costs.

- But the key scalability bottleneck is P time.
  - 18 months ago, I estimated P overhead as 1 million-100 million-fold.
    - To prove it correctly ran a computer program:
      - If the program runs in 1 second, P runs in 1 million seconds.
      - Though the overhead is highly parallelizable.
      - See https://a16zcrypto.com/posts/article/measuring-snark-performance-frontends-backends-and-the-future/

Performance bottlenecks

- Today's most popular SNARKs are optimized for **verifier** costs at the expense of **prover** costs.

- But the key scalability bottleneck is P time.
  - A key contributor to P time is **commitment costs**.
    - i.e., how much time it takes P to cryptographically commit to large vectors.
    - This involves expensive operations like hashing and FFTs, or group operations (depending on the polynomial commitment scheme used).

What should change?

- For a fast P, a complete reconfiguration of today's deployments is needed.

What should change?

- For a fast P, a complete reconfiguration of today's deployments is needed.

- Today's deployed SNARKs are based on a combination of:
  - **Constant-round** polynomial IOPs
    - Plonk, Marlin, STARKs, Groth16*.
  - **Many-round** (or non-transparent) polynomial commitments.
    - FRI, Bulletproofs, KZG.

What should change?

- For a fast P, a complete reconfiguration of today's deployments is needed.

- Today's deployed SNARKs are based on a combination of:
  - **Constant-round** polynomial IOPs
    - Plonk, Marlin, STARKs, Groth16*.
  - **Many-round** (or non-transparent) polynomial commitments.
    - FRI, Bulletproofs, KZG.

- For a fast P, we should use:
  - **Many-round** polynomial IOPs
    - Use the sum-check protocol.
  - **Constant-round** polynomial commitments
    - E.g., Ligero/Brakedown/Binius

What should change?

- For a fast P, a complete reconfiguration of today's deployments is needed.

- Today's deployed SNARKs are based on a combination of:
  - **Constant-round** polynomial IOPs
    - Plonk, Marlin, STARKs, Groth16*.
  - **Many-round** (or non-transparent) polynomial commitments.
    - FRI, Bulletproofs, KZG.

- For a fast P, we should use:
  - **Many-round** polynomial IOPs
    - Use the sum-check protocol.
  - **Constant-round** polynomial commitments
    - E.g., Ligero/Brakedown/Binius
- If prover **space** is a key bottleneck, should use a folding scheme like Nova.

What performance can now be achieved?

- Developments since August (Lasso/Jolt/BabySpartan/Binius)
  - P overhead may now be ~50,000-fold.
  - ~20x improvement relative to 18 months ago.

- Compared to today's SNARK deployments, this involves:
  - Different polynomial IOPs.
    - Under the hood: multivariate rather univariate polynomials.
  - Different polynomial commitment schemes.
  - Different hash functions.
  - Different finite fields.
  - Different front-end techniques.

Fast-prover SNARKs: overview

Key tool: the sum-check protocol

Sum-Check Protocol [LFKN90]

- Input: An ℓ-variate polynomial g over field $\boldsymbol{F}$.
- Goal: compute the quantity:

$$\sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \dots \sum_{b_\ell \in \{0,1\}} g(b_1, \dots, b_\ell).$$

Sum-Check Protocol [LFKN90]

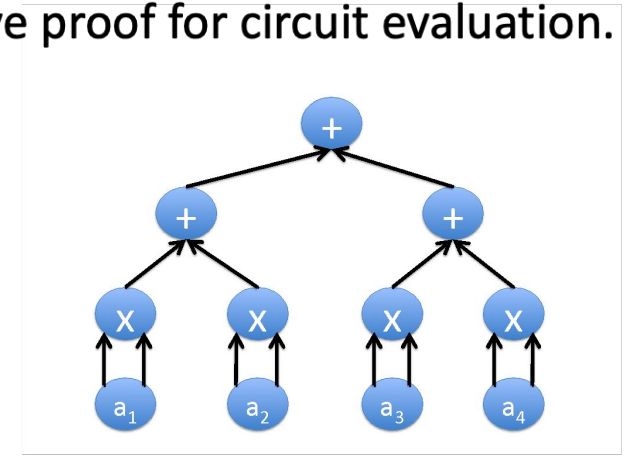
- Input: An ℓ-variate polynomial g over field $\boldsymbol{F}$.
- Goal: compute the quantity:

$$\sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_\ell \in \{0,1\}} g(b_1, \ldots, b_\ell).$$

- V only evaluates g at **one** point.
  - Much better than the 2^ℓ evaluations required to compute the sum herself.
- P does just a constant factor extra work compared to "native computation".
  - i.e., computing the sum with no proof of correctness.

The GKR protocol (2008)

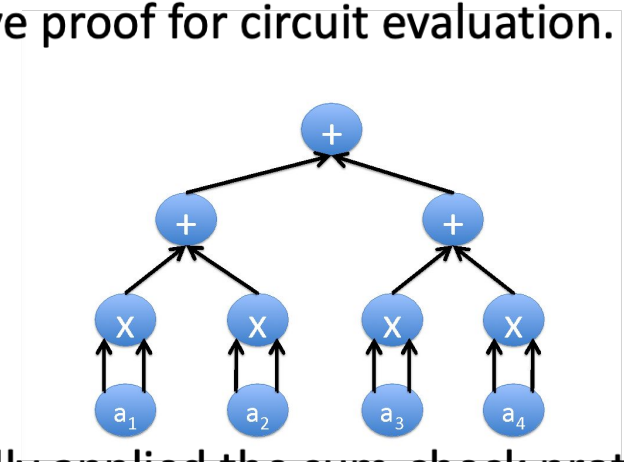
- Interactive proof for circuit evaluation.



- Repeatedly applied the sum-check protocol.
- For circuits of size S, P does $O(S)$ field operations [CMT12, T13, XZZPS19].
- In applications to SNARKs, the input to the circuit is committed, and the proof size is $O(\log(S)^2)$.

The GKR protocol (2008)

- Interactive proof for circuit evaluation.



- Repeatedly applied the sum-check protocol.
- For circuits of size S, P does $O(S)$ field operations [CMT12, T13, XZZPS19].
  - No cryptography involved.
- In applications to SNARKs, the input to the circuit is committed, and the proof size is $O(\log(S)^2)$.

Takeaways for SNARK design

- The sum-check protocol is a tool for minimizing the prover's **commitment costs.**
  - Allows P to commit to **less data** than in SNARKs based on constant-round polynomial IOPs.
    - There are **fewer** committed values.
    - Each committed value is **smaller.**
      - Smaller values can be **much** faster to commit to than big values.
      - E.g., for KZG or Bulletproofs, a small value is over 10x faster to commit to than a big one.

Lasso: A lookup argument with low commitment costs

Indexed lookup arguments

- Let $t \in \boldsymbol{F}^N$ be the vector of N table entries.
- P has committed to a vector $((a_1, b_1), \ldots, (a_m, b_m))$.
- P claims that $a_i = t[b_i]$ for $i = 1 \ldots m$.

- Lasso is a new lookup argument whereby P commits to fewer and smaller field elements than in prior work.
  - Under the hood, it uses the sum-check protocol and the GKR protocol to achieve this.

Lasso Performance

- For m lookups into a table of size N:
  - Lasso P commits to $m + N$ field elements.
    - All of them “small” (in {0, 1, …, m}).
    - No commitment to the table t is needed if the table is “structured”.
    - Can reduce the dependence on N for ”decomposable” tables
  - Comparison points:
    - Plookup [GW20] P commits to $5 * \max\{m, N\}$ field elements.
      - Plus the table commitment.
      - $3 * \max\{m, N\}$ of the committed field elements are random.
    - CQ [EFG22]: P commits to $8m$ field elements (after pre-processing)
      - $7m$ of the committed field elements are random.
    - Lasso is implemented and shows more than an order of magnitude improvement over the lookup argument in Halo2 (variant of Plookup).

Lasso Performance

- For m lookups into a table of size N:
  - Lasso P commits to $m + N$ field elements.
    - All of them “small” (in {0, 1, …, m}).
    - No commitment to the table t is needed if the table is “structured”.
    - Can reduce the dependence on N for ”decomposable” tables
  - Comparison points:
    - Plookup [GW20] P commits to $5 * \max\{m, N\}$ field elements.
      - Plus the table commitment.
      - $3 * \max\{m, N\}$ of the committed field elements are random.
    - Lasso is implemented and shows more than an order of magnitude improvement over the lookup argument in Halo2 (variant of Plookup).
    - CQ [EFG22]: P commits to $8m$ field elements (after pre-processing)
      - $7m$ of the committed field elements are random.

Jolt: a zkVM based on Lasso

Front-ends today for VM execution

- Say P claims to have run a certain computer program for m steps.
  - Say the program is written in the assembly language for a VM.
  - Popular VM's targeted: RISC-V, Ethereum Virtual Machine (EVM)
- Today, front-ends produce a circuit that, for each step of the computation:
  1. Figures out what instruction to execute at that step.
  2. Executes that instruction.

Lasso lets one replace Step 2 with a single lookup.

For each instruction, the table stores the entire evaluation table of the function.

If instruction f operations on two 64-bit inputs, the table stores $f(x, y)$ for every pair of 64-bit inputs (x, y).

This table has size 2^{128}.

All RISC-V instructions are decomposable.

Jolt: A new front-end paradigm

- Say P claims to have run a certain computer program for m steps.
  - Say the program is written in the assembly language for a VM.
  - Popular VM's targeted: RISC-V, Ethereum Virtual Machine (EVM)
- Today, front-ends produce a circuit that, for each step of the computation:
  1. Figures out what instruction to execute at that step.
  2. Executes that instruction.
- Lasso lets one replace Step 2 with a single lookup.
  - For each instruction, the table stores the **entire evaluation table of the instruction.**

If instruction f operations on two 64-bit inputs, the table stores $f(x, y)$ for every pair of 64-bit inputs (x, y).

This table has size 2^{128}.

All RISC-V instructions are decomposable.

Jolt: A new front-end paradigm

- Say P claims to have run a certain computer program for m steps.
  - Say the program is written in the assembly language for a VM.
  - Popular VM's targeted: RISC-V, Ethereum Virtual Machine (EVM)
- Today, front-ends produce a circuit that, for each step of the computation:
  1. Figures out what instruction to execute at that step.
  2. Executes that instruction.
- Lasso lets one replace Step 2 with a single lookup.
  - For each instruction, the table stores the **entire evaluation table of the instruction.**
  - If instruction f operations on two 64-bit inputs, the table stores $f(x, y)$ for every pair of 64-bit inputs (x, y).
    - This table has size 2^{128}.
    - Jolt shows that all RISC-V instructions are structured/decomposable.

Costs of Jolt

- For RISC-V instructions on 32-bit data types (with "multiply extension"):
  - Jolt's P commits to about **60** values per step, requiring **~800 bits** in total to specify.
    - Where a b-bit value is between 0 and $2^b - 1$.
  - **Rough** comparison points:
    - RISC-Zero currently commits to at least 275 values per step, each in a 31-bit field.
      - They use FRI as the polynomial commitment scheme, which is not significantly faster for small values than big ones.
      - So 275 31-bit field elements equations to 8500+ committed bits per step.

Jolt implementation status

- Initial implementation of Jolt (with elliptic-curve-based commitment schemes) is nearing completion.

BabySpartan: Applying Lasso to arbitrary circuits

BabySpartan

- Jolt can be seen as an application of Lasso to VM execution, but Lasso is more generally applicable.
- BabySpartan is a new SNARK for circuit satisfiability (supports a generalization of “Plonkish circuits”).
  - P commits to 4 values per gate of the circuit. All values are small.
  - Comparison point:
    - Plonk P commits to 7 random values per gate (and additional non-random ones).

Binius
[Diamond and Posen 2023]

Context: Ligero/Brakedown commitments

- Lasso/Jolt and other sum-check-based SNARKs can use **any** commitment scheme for multilinear polynomials.
- The best one for P is Ligero/Brakedown.
  - A hashing-based commitment scheme.
    - Same family as FRI.
    - No group operations, works over small fields.
  - But its proofs are somewhat big.
    - Square root in the size of the polynomial.

Context: Ligero/Brakedown commitments

- Lasso/Jolt and other sum-check-based SNARKs can use **any** commitment scheme for multilinear polynomials.
- The best one for P is Ligero/Brakedown.
  - A hashing-based commitment scheme.
    - Same family as FRI.
    - No group operations, works over small fields.
  - But its proofs are somewhat big.
    - Square root in the size of the polynomial.
  - One can reduce proof size and V work via SNARK recursion.
    - Pushes V's work onto P.
    - But since the Ligero/Brakedown verifier is expensive (lots of hash evaluations), applying recursion will bottleneck P.

Binius (Diamond and Posen, last week)

1. Make Ligero/Brakedown P **even faster** when committing to small values.
    - E.g., committing to a 1-bit value (0 or 1) is now ~32x faster than committing to a 32-bit value.
2. Combine this with a Lasso-based SNARK for circuit-SAT (similar to BabySpartan) to get **much** faster SNARKs for standard hash functions like Keccak.
3. This enables applying recursion to Ligero/Brakedown, addressing proof size issue.
    - Because the Ligero/Brakedown verifier mainly does hashing.
    - So applying recursion to Ligero/Brakedown amounts to P proving it correctly hashed a lot of data.

Side benefits

- Can now use standard hash functions like Keccak in recursive SNARKs, rather than “SNARK-friendly ones” like Poseidon.
    - Faster, better-understood security.
    - Addresses key bottleneck for “Type-1 zkEVMs” (full Ethereum equivalence).
- Takeaway: we’ve been designing hash functions to be friendly to the artificial limitations of today’s popular SNARKs.

Conclusion

- SNARK provers can be sped up well over an order of magnitude, compared to today's popular SNARKs.

- Side benefit of better performance: **better security.**
    - Simpler+ more auditable SNARKs and toolchains.
    - Less aggressive cryptographic assumptions.

Conclusion

- SNARK provers can be sped up well over an order of magnitude, compared to today's popular SNARKs.

- Side benefit of better performance: **better security.**
    - Simpler+ more auditable SNARKs and toolchains.
    - Less aggressive cryptographic assumptions.

- Compared to today's deployments, this involves:
    - Different polynomial IOPs.
        - Under the hood: multivariate rather univariate polynomials, the sum-check protocol.
    - Different commitment schemes.
    - Different hash functions, different finite fields.
    - Different front-end techniques.

Thank you!