Challenge Proposal: Accelerating Witness Generation through Circuit Optimization

CEL Team

January 16, 2026

1 Context: The Bottleneck of Verifiable Computation

Zero-knowledge (zk) proofs are rapidly becoming foundational tools in cryptography, enabling applications such as scalable rollups, anonymous identity protocols, verifiable machine learning, and decentralized AI agents.
While zk proving systems have seen major performance improvements in proof generation and verification, a lesser-known bottleneck remains:

> *Witness generation, i.e. the computation of all intermediate values required to satisfy a zk circuit.*

This step can dominate runtime, and improving the performance of witness generation has the potential to accelerate proving systems across many domains.

1.1 Background: Rank-1 Constraint System (R1CS)

A Rank-1 Constraint System (R1CS) is a set of equations of the form:

$$\langle \mathbf{a}_i, \mathbf{w} \rangle \cdot \langle \mathbf{b}_i, \mathbf{w} \rangle = \langle \mathbf{c}_i, \mathbf{w} \rangle \tag{1}$$

where:

- $\mathbf{w}$ is the **witness vector**, containing all values involved in the computation:
  - A leading 1 (used for constants)
  - All **public inputs** (known to the verifier)
  - All **public outputs** (claimed results, known to the verifier)
  - All **private inputs** (secret data, hidden from the verifier)
  - All **intermediate variables** (internal computation values, hidden from the verifier)
- $\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i$ are coefficient vectors selecting and weighting elements of $\mathbf{w}$ for constraint i.

In matrix notation, an R1CS can be written as:

$$(\mathbf{A}\mathbf{w}) \circ (\mathbf{B}\mathbf{w}) = \mathbf{C}\mathbf{w} \tag{2}$$

where $\circ$ denotes element-wise multiplication, $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are $n \times m$ matrices, n is the number of constraints, and m is the witness size.

Example: $z = x^2 \cdot y + y$

We decompose $x^2 \cdot y + y$ into three constraints:

$$\begin{aligned} v_1 &= x \cdot x \quad \text{(Constraint 1)} \\ v_2 &= v_1 \cdot y \quad \text{(Constraint 2)} \\ z &= v_2 + y \quad \text{(Constraint 3)} \end{aligned}$$

The (unassigned) witness vector is: $\mathbf{w} = [1, z, x, y, v_1, v_2]$ where:

- 1: constant term
- x, y: **public inputs** (known to the verifier)
- z: **public output** (verifier checks it)
- v_1, v_2: intermediate variables (private)

The constraints in R1CS row form are:

$$\begin{aligned}
&\text{C1: } \langle [0,0,1,0,0,0],\, \mathbf{w} \rangle \cdot \langle [0,0,1,0,0,0],\, \mathbf{w} \rangle = \langle [0,0,0,0,1,0],\, \mathbf{w} \rangle \quad \Rightarrow \quad x \cdot x = v_1, \\
&\text{C2: } \langle [0,0,0,0,1,0],\, \mathbf{w} \rangle \cdot \langle [0,0,0,1,0,0],\, \mathbf{w} \rangle = \langle [0,0,0,0,0,1],\, \mathbf{w} \rangle \quad \Rightarrow \quad v_1 \cdot y = v_2, \\
&\text{C3: } \langle [0,0,0,1,0,1],\, \mathbf{w} \rangle \cdot \langle [1,0,0,0,0,0],\, \mathbf{w} \rangle = \langle [0,1,0,0,0,0],\, \mathbf{w} \rangle \quad \Rightarrow \quad (v_2 + y) \cdot 1 = z.
\end{aligned}$$

Assigned example

If $x = 3$, $y = 5$ (public), then: $v_1 = 9$, $v_2 = 45$, $z = 50$
The full witness is: $\mathbf{w} = [1, \mathbf{50}, \mathbf{3}, \mathbf{5}, \mathbf{9}, \mathbf{45}]$ The verifier sees $x = 3$ and $z = 50$; the proof ensures there exist private intermediate values v_1, v_2 satisfying all constraints without revealing them.

2 Challenge Statement: Accelerating Witness Generation through Circuit Optimization

Witness generation becomes a bottleneck because it requires evaluating all intermediate constraints in the circuit. These operations dominate both runtime and memory use. While proof systems have been aggressively optimized, this layer remains under-explored and has become a new performance bottleneck for proof generation.[1]

This challenge addresses a critical systems-level bottleneck in zero-knowledge workflows:

> *Given a random R1CS circuit C^0, participants must produce an alternative R1CS circuit C^* that computes the same function as C^0 and uses fewer constraints than C^0.*

- Submissions are evaluated by verifying correctness (equivalence to C^0), comparing constraint counts with C^0 and verifying that C^* is not underconstrained.
- Input C^0 R1CS circuits are randomly generated to prevent hand-optimizing against known benchmarks.
- Participants must support various difficulty tiers. Increasing difficulty is obtained by scaling the circuit complexity parameters such as size, depth, and redundancy (see Sec. 2.4).

The primary goal for challengers is to write or compile more optimized circuits (C^*) representing the same function. Successful optimization results in smaller constraint systems and, by consequence, faster witness generation and reduced memory overhead. This challenge can add value in:

1. Optimizing unoptimized circuits:
    - Current compilers (e.g., Circom) perform only basic optimizations, so unoptimized code lead to inefficient circuits;
    - Writing hand-crafted optimized code is a high barrier to entry. Instead, to lower the barrier for developers, Circom can be used to compile unoptimized code, and then TIG can take the unoptimized circuits and get them much closer (if not better) than what can be achieved with optimized code.
2. Making Advancements in circuit simplification, e.g., extracting common subexpressions or additive terms, is non-trivial but offers large gains.

2.1 Random Instance Generation

To ensure fair, reproducible, and scalable benchmarks, we utilize a deterministic procedural generation pipeline. This approach mitigates the risk of overfitting to static benchmarks and allows for arbitrary scaling of difficulty. The generation process transforms a cryptographic seed into a valid R1CS circuit through three formal stages:

1. **Procedural Signal Graph Generation:** We define a custom generator $\mathcal{G}$ that accepts a random seed s and a configuration vector θ. While s provides the entropy for uniqueness, $\theta = \langle K_{target}, \rho, D, P_{map} \rangle$ acts as the blueprint for complexity. Specifically:

   - K_{target}: The target constraint count.
   - ρ: The density of intentionally redundant logic (for optimization potential).
   - D: The maximum computational depth.
   - P_{map}: The probability of injecting **Algebraic Power Maps** (e.g., $x \mapsto x^5$) instead of standard linear operations.

   This inclusion of Power Maps mimics the non-linear S-Boxes found in ZK-friendly hash functions (like Poseidon over BN254), ensuring the challenge targets realistic algebraic bottlenecks rather than unstructured noise.

2. **Transpilation to High-Level IR:** The raw DAG is deterministically lowered into Circom code. Crucially, high-degree operations (like x^5) are explicitly **unrolled** into chains of quadratic constraints (e.g., $x^2 = x \cdot x$, $x^4 = x^2 \cdot x^2$, $x^5 = x^4 \cdot x$). This unrolling introduces intermediate signals that serve as prime targets for optimization, testing the solver's ability to identify and collapse algebraic redundancy.

3. **R1CS Compilation:** The high-level IR is compiled to the final R1CS baseline C^0 with optimization flags disabled (e.g., `--O0`). This ensures that the structural inefficiencies injected by the generator are preserved, providing the necessary margin for challengers to achieve the required Better-Than-Baseline (BTB) improvement.

Since the pipeline is deterministic, the tuple (s, θ) acts as a compressed representation of the challenge. Both Prover and Verifier can independently reconstruct C^0 from the seed, ensuring transparency.

2.2 Solution verification

To verify that the circuit C^* provided by the Challenger is indeed a valid solution, we need to check that the C^* performs the same computation as C^0, i.e.

$$C^*(x) = C^0(x) \ \ , \forall x \in \mathbb{F} \tag{3}$$

Where $\mathbb{F}$ is a finite field. In the following, we reduce the condition in Eq. 3 to checking circuit equivalence at a random point, meaning:

$$C^*(x_{eval}) = C^0(x_{eval}) \ \ , x_{eval} \in \mathbb{F} \tag{4}$$

This reduction is rooted in the fact that each R1CS set of equations can be represented as a set of polynomial functions mapping the public input variables to the public output variables. We thus leverage the Schwartz-Zippel lemma to prove probabilistic equality of two polynomials (or sets of polynomials) by evaluating them at a random point over a sufficiently large field.

To this end, we adopt the following scheme. We refer to **Prover** as the entity submitting the solution and **Verifier** as the entity verifying the correctness of the solution.

Given this notation, the challenge and verification protocol proceeds as follows:

1. **Setup:** Verifier sends an instance of the problem, i.e., an R1CS circuit C^0 with K^0 constraints.
2. **Optimization:** Prover computes a new R1CS circuit C^* with K^* constraints.
3. **Random Point Derivation:** Given a hashing function $\mathcal{H}$ (known to both parties), the Prover computes the canonical hashes[1] of:
   - The input circuit C^0, i.e., $h_0 = \mathcal{H}(C^0)$;
   - The computed circuit C^*, i.e., $h_* = \mathcal{H}(C^*)$;
   - The random evaluation point $x_{eval} = \mathcal{H}(h_0 || h_*)$.
4. **Execution:** Prover executes *both* circuits locally using x_{eval} as the input:
   - Computes the witness w^0 and public output y^0_{pub} for C^0;
   - Computes the witness w^* and public output y^*_{pub} for C^*.
5. **Proof Generation:** To relieve the Verifier of re-executing C^0, the Prover generates two Zero-Knowledge proofs using Spartan (or an equivalent proving system):
   - π^0: A proof that $C^0(x_{eval}, w^0) \rightarrow y^0_{pub}$;
   - π^*: A proof that $C^*(x_{eval}, w^*) \rightarrow y^*_{pub}$.
6. **Submission:** Prover sends the tuple $\mathcal{S} = \left[C^*, x_{eval}, y^0_{pub}, y^*_{pub}, \pi^0, \pi^*\right]$ to the Verifier.

[1]To compute the hash of a circuit, we define a canonical representation. The canonical hash is computed by lexicographically sorting all constraints based on the string representation of their coefficients and variable indices. This ensures that topologically identical circuits yield the same hash regardless of variable naming permutations.

The **Verifier** then validates the submission via the following steps:

1. **Hash Validation:** Verifier independently computes $h'_0 = \mathcal{H}(C^0)$ and $h'_* = \mathcal{H}(C^*)$ to verify that the evaluation point provided matches the canonical derivation:

$$x_{eval} \overset{?}{=} \mathcal{H}(h'_0 || h'_*)$$

2. **Constraint Check:** Verifier checks that the challenger has actually optimized the circuit:

$$K^* < K^0$$

3. **Output Equivalence:** Verifier checks that both circuits produced identical public outputs:

$$y^0_{pub} \overset{?}{=} y^*_{pub}$$

4. **Proof Verification:** Verifier leverages the Spartan verifier to check the validity of both proofs:

   - Verify π^0: Confirms y^0_{pub} is the true output of the baseline circuit C^0 on x_{eval};
   - Verify π^*: Confirms y^*_{pub} is the true output of the optimized circuit C^* on x_{eval}.

Note on Performance: By requiring the Prover to submit a proof (π^0) for the baseline execution, we transform the verification complexity from linear $O(|C^0|)$ to constant time $O(1)$ (dominated only by SNARK verification). This ensures the Verifier remains lightweight regardless of the difficulty or size of the challenge circuit.[2]

[2]Further optimization is possible by constructing a single "Solution Circuit" that composes the hashing, execution, and comparison into one proving system. However, the "Double Proof" scheme described here offers the best balance between implementation simplicity and verification speed.

2.3 Security Analysis

The proposed verification protocol relies on a combination of probabilistic checking and cryptographic commitments to guarantee soundness. The security of the scheme holds under the following standard assumptions:

1. **Collision Resistance of the Hash Function:** We assume $\mathcal{H}$ (e.g., Poseidon or SHA-256) is a collision-resistant hash function. This prevents a malicious Prover from finding two distinct circuits $C_1 \neq C_2$ such that $\mathcal{H}(C_1) = \mathcal{H}(C_2)$.

2. **Schwartz-Zippel Lemma (Probabilistic Equivalence):** The core security guarantee relies on the fact that any directed acyclic arithmetic circuit (such as our R1CS) inherently computes a multivariate polynomial map from inputs to outputs [2]. The *Schwartz-Zippel Lemma* [3] states that for two distinct polynomials $P(x)$ and $Q(x)$ over a large finite field $\mathbb{F}$, the probability that they agree on a randomly selected point $r \in \mathbb{F}$ is bounded by:

$$\Pr[P(r) = Q(r) \mid P \neq Q] \leq \frac{d}{|\mathbb{F}|}$$

   where d is the total multiplicative degree (depth) of the circuit. Since the field size $|\mathbb{F}|$ is cryptographically large (e.g., $\approx 2^{254}$) and vastly exceeds any realizable circuit depth d, this probability is negligible.

3. **Input Commitment (Fiat-Shamir Transformation):** To prevent "Input Grinding" (where an attacker crafts a circuit C^* that is valid only for a specific input), the protocol enforces a strict dependency chain via the *Fiat-Shamir heuristic* [4]:

$$C^* \xrightarrow{\text{hash}} \mathcal{H}(C^*) \xrightarrow{\text{derive}} x_{eval}$$

   The evaluation point x_{eval} is derived *after* the circuit is committed to. Any modification to C^* to accommodate a specific input would alter $\mathcal{H}(C^*)$, thereby changing x_{eval} itself. This circular dependency renders input-specific attacks computationally infeasible.

4. **Soundness of the ZK-SNARK:** The scheme assumes the underlying proving system, Spartan [5], is computationally sound. A malicious Prover cannot generate a valid proof π for a false statement (e.g., claiming y^*_{pub} is the output of C^* when it is not) except with negligible probability.

Under these conditions, the "Double Proof" verification scheme provides security equivalent to direct execution by the Verifier, while reducing the Verifier's computational complexity from linear $O(N)$ to constant $O(1)$.

2.4 Difficulty Tracks

We parameterize the challenge difficulty tracks by a $\boldsymbol{\delta}$, the **Circuit complexity** scalar.

2.4.1 Circuit Complexity (δ)

Our procedural generation allows us to scale challenges dynamically. We define the circuit complexity δ as a target scalar value. To generate an instance matching a specific difficulty δ, we map it to a concrete configuration vector θ via a scaling function $\mathcal{F}$:

$$\theta = \mathcal{F}(\delta) = \langle K_{target}(\delta), \rho, P_{alias}, P_{lin}, P_{map} \rangle \tag{5}$$

Where:

- $K_{target}(\delta) = \delta \times 1000$ – scales the constraint count linearly with difficulty.
- $\rho = 0.25$ – redundancy ratio held constant at 25%, ensuring consistent optimization potential from shared subexpressions across all difficulty levels.
- $P_{alias} = 0.15$ – alias operation frequency held constant at 15%.
- $P_{lin} = 0.20$ – linear scaling operation frequency held constant at 20%.
- $P_{map} = 0.15$ – algebraic power map (S-boxed) frequency held constant at 15%.

This scaling approach maintains isomorphic difficulty characteristics across all tiers. Each difficulty level produces circuits with the same statistical distribution of optimization traps and redundancy patterns, scaled proportionally in size. This ensures that:

1. The optimization challenge remains consistent across difficulties – a solver that performs well on difficulty 1 should scale predictably to higher difficulties.
2. Performance benchmarks are comparable – doubling the difficulty approximately doubles the circuit size while maintaining the same 65% theoretical reduction potential.

If future versions require non-uniform difficulty scaling (where higher difficulties become progressively harder to optimize beyond just size), the parameters ρ, P_{alias}, P_{lin}, and P_{map} can be made functions of δ. For example, decreasing $\rho(\delta)$ at higher difficulties would reduce redundancy and make common subexpression elimination less effective.

2.4.2 Empirical Calibration and Variance Control

A critical requirement for a fair challenge is **Isotropic Difficulty**: two distinct random instances generated with the same complexity parameters θ must present a comparable optimization challenge. If the variance between instances is high, the competition risks rewarding "lucky" seeds rather than superior solvers.
To ensure consistency, we employ a pre-computation calibration phase using a Reference Solver S_{ref} (e.g., standard Circom compiler with `-O1`). We validate the difficulty parameters as follows:

1. **Monte Carlo Sampling:** For a candidate configuration θ, we generate a large batch of instances $\{C_1^0, \ldots, C_N^0\}$.

2. **Reference Reduction:** We measure the *Natural Reducibility* η_i of each instance using the Reference Solver:
$$\eta_i = 1 - \frac{|S_{ref}(C_i^0)|}{|C_i^0|} \tag{6}$$

3. **Variance Constraint:** We calculate the mean reducibility μ_η and standard deviation σ_η of the batch. A configuration θ is valid *if and only if* the standard deviation is below a strict tolerance threshold τ (e.g., $\sigma_\eta < 0.05$).

This protocol ensures that the difficulty track parameter δ is a reliable predictor of effort. By filtering out configuration vectors that produce unstable or ”noisy” difficulty distributions, we guarantee that all participants in a given tier face a statistically equivalent optimization task, regardless of the random seed assigned to them.

3 Quality

Given a challenger's solution, we define its quality $\epsilon \in [0, 1]$ as :

$$\boxed{1 - \frac{K^*}{K^0} = \epsilon} \tag{7}$$

where K^* and K^0 are the number of constraints of the challenger's solution circuit and the original circuit, respectively. Thus, larger ϵ means higher circuit reduction. For intuition: $\epsilon = 0.6$ means to achieve a 60% reduction in the circuit's size.

Acknowledgments We are deeply grateful to Carlo Modica and Wisdom Ogwu for their invaluable contributions to this work. Carlo Modica provided essential insights into the challenge design methodology. Wisdom Ogwu contributed significantly to the development and analysis of the solution verification algorithm.

References

[1] Zheming Ye, Xiaodong Qi, Zhao Zhang, and Cheqing Jin. *Yoimiya: A Scalable Framework for Optimal Resource Utilization in ZK-SNARK Systems*. arXiv preprint arXiv:2502.18288, 2024. `https://arxiv.org/pdf/2502.18288`

[2] Amir Shpilka and Amir Yehudayoff. *Arithmetic Circuits: A survey of recent results and open questions*. Foundations and Trends® in Theoretical Computer Science, 5(3–4):207–388, 2010.

[3] Jacob T. Schwartz. *Fast probabilistic algorithms for verification of polynomial identities*. Journal of the ACM (JACM), 27(4):701–717, 1980.

[4] Amos Fiat and Adi Shamir. *How to Prove Yourself: Practical Solutions to Identification and Signature Problems*. Advances in Cryptology — CRYPTO' 86, Lecture Notes in Computer Science, vol 263. Springer, 1986.

[5] Srinath Setty. *Spartan: Efficient and general-purpose zkSNARKs without trusted setup*. Advances in Cryptology – CRYPTO 2020, pages 704–737. Springer, 2020.