UltraGroth: Interactive Groth16

August 29, 2025

Dmytro Zakharov
Distributed Lab

distributedlab.com/
github.com/rarimo/ultragroth

Why we should care?

Range Checks

Problem

Write a circuit that checks whether x is a 128-bit integer.

Range Checks

Problem

Write a circuit that checks whether x is a 128-bit integer.

Current R1CS (and, consequently, Circom's) approach is to conduct the following steps:

- Find bit decomposition of x off-circuit: say, $x = \sum_{i=0}^{127} x_i 2^i$.

Range Checks

Problem

Write a circuit that checks whether x is a 128-bit integer.

Current R1CS (and, consequently, Circom's) approach is to conduct the following steps:

- Find bit decomposition of x off-circuit: say, $x = \sum_{i=0}^{127} x_i 2^i$.
- Check that $x_i \in \{0, 1\}$: impose 128 constraints $x_i(1 - x_i) = 0$.

Range Checks

Problem

Write a circuit that checks whether x is a 128-bit integer.

Current R1CS (and, consequently, Circom's) approach is to conduct the following steps:

- Find bit decomposition of x off-circuit: say, $x = \sum_{i=0}^{127} x_i 2^i$.
- Check that $x_i \in \{0, 1\}$: impose 128 constraints $x_i(1 - x_i) = 0$.

Result: 128 constraints per *128-bit range check*.

Range Checks

Problem

Write a circuit that checks whether x is a 128-bit integer.

Current R1CS (and, consequently, Circom's) approach is to conduct the following steps:

- Find bit decomposition of x off-circuit: say, $x = \sum_{i=0}^{127} x_i 2^i$.
- Check that $x_i \in \{0, 1\}$: impose 128 constraints $x_i(1 - x_i) = 0$.

Result: 128 constraints per *128-bit range check*.

Question

Suppose one needs to conduct 10000 such range checks. How many constraints does one need to implement this?

Range Checks

Problem

Write a circuit that checks whether x is a 128-bit integer.

Current R1CS (and, consequently, Circom's) approach is to conduct the following steps:

- Find bit decomposition of x off-circuit: say, $x = \sum_{i=0}^{127} x_i 2^i$.
- Check that $x_i \in \{0, 1\}$: impose 128 constraints $x_i(1 - x_i) = 0$.

Result: 128 constraints per *128-bit range check*.

Question

Suppose one needs to conduct 10000 such range checks. How many constraints does one need to implement this?

Using quite unsophisticated math, $128 \times 10000 =$ **1.28 mln**.

Better range checks

Using lookup checks, we can implement the same logic in just $\approx$ 100k constraints! Here is how.

Better range checks

Using lookup checks, we can implement the same logic in just $\approx$ 100k constraints! Here is how.

Assumption. Assume we can check whether the given signal s is the w-bit integer in *a single constraint*. But this requires a *one-time* cost of 2^w constraints. How does it help us?

Better range checks

Using lookup checks, we can implement the same logic in just $\approx$ 100k constraints! Here is how.

Assumption. Assume we can check whether the given signal s is the w-bit integer in *a single constraint*. But this requires a *one-time* cost of 2^w constraints. How does it help us?

Suppose we use $w := 16$. Then, our algorithm proceeds as follows:

Better range checks

Using lookup checks, we can implement the same logic in just $\approx$ 100k constraints! Here is how.

Assumption. Assume we can check whether the given signal s is the w-bit integer in *a single constraint*. But this requires a *one-time* cost of 2^w constraints. How does it help us?

Suppose we use $w := 16$. Then, our algorithm proceeds as follows:

- We pay $2^{16} \approx 65.5\text{k}$ for a one-time commitment.

Better range checks

Using lookup checks, we can implement the same logic in just $\approx$ 100k constraints! Here is how.

Assumption. Assume we can check whether the given signal s is the w-bit integer in *a single constraint*. But this requires a *one-time* cost of 2^w constraints. How does it help us?

Suppose we use $w := 16$. Then, our algorithm proceeds as follows:

- We pay $2^{16} \approx 65.5\text{k}$ for a one-time commitment.
- We find w-width decomposition of x: say, $x = \sum_{i=0}^{7} x_i 2^{wi}$.

Better range checks

Using lookup checks, we can implement the same logic in just $\approx$ 100k constraints! Here is how.

Assumption. Assume we can check whether the given signal s is the w-bit integer in *a single constraint*. But this requires a *one-time* cost of 2^w constraints. How does it help us?

Suppose we use $w := 16$. Then, our algorithm proceeds as follows:

- We pay $2^{16} \approx 65.5\text{k}$ for a one-time commitment.
- We find w-width decomposition of x: say, $x = \sum_{i=0}^{7} x_i 2^{wi}$.
- We check whether x_i is a 16-bit integer. Since we have 8 chunks, this costs 8 constraints.

Better range checks

Using lookup checks, we can implement the same logic in just $\approx$ 100k constraints! Here is how.

Assumption. Assume we can check whether the given signal s is the w-bit integer in *a single constraint*. But this requires a *one-time* cost of 2^w constraints. How does it help us?

Suppose we use $w := 16$. Then, our algorithm proceeds as follows:

- We pay $2^{16} \approx 65.5$k for a one-time commitment.
- We find w-width decomposition of x: say, $x = \sum_{i=0}^{7} x_i 2^{wi}$.
- We check whether x_i is a 16-bit integer. Since we have 8 chunks, this costs 8 constraints.

Result: We pay 65.5k constraints once and then every 128-bit range checks costs only 8 constraints instead of 128!

Illustration

Let us illustrate this visually for a 16-bit range check over x!

Illustration

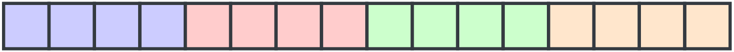
Let us illustrate this visually for a 16-bit range check over x!

$x =$

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}

16 constraints

$x =$



x_0 x_1 x_2 x_3

4 constraints + one-time 2^4 commitment

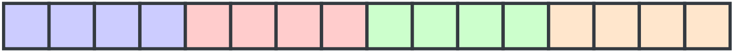
Illustration

Let us illustrate this visually for a 16-bit range check over x!

$$x = \boxed{x_0}\boxed{x_1}\boxed{x_2}\boxed{x_3}\boxed{x_4}\boxed{x_5}\boxed{x_6}\boxed{x_7}\boxed{x_8}\boxed{x_9}\boxed{x_{10}}\boxed{x_{11}}\boxed{x_{12}}\boxed{x_{13}}\boxed{x_{14}}\boxed{x_{15}}$$

16 constraints

$x =$



x_0 x_1 x_2 x_3

4 constraints + one-time 2^4 commitment

Example: 10000 such range checks would cost $16 \times 10000 = 160\text{k}$ constraints for a regular R1CS while $2^4 + 4 \times 10000 \approx 40\text{k}$ constraints over ZK system with lookups.

Applications

- Wrappings of non-native ZKP verifications: e.g., zk-STARKs, sumcheck-based approaches.

Applications

- Wrappings of non-native ZKP verifications: e.g., zk-STARKs, sumcheck-based approaches.
- Non-native field arithmetic: e.g., optimized ECDSA verification for Rarimo passport verification.

Applications

- Wrappings of non-native ZKP verifications: e.g., zk-STARKs, sumcheck-based approaches.
- Non-native field arithmetic: e.g., optimized ECDSA verification for Rarimo passport verification.
- And surely, zero-knowledge Machine Learning — Bionetta.

Framework	Metric	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6	ResNet	MobileNet
Bionetta (UltraGroth)	Constraints #	68.4K	66.7K	106.8K	126.8K	108.4K	187.7K	1.03M	2.50M
	Proof Size (KB)	1.20	1.20	1.20	1.20	1.20	1.20	1.20	1.20
	PK (MB)	48.40	50.60	80.60	106.30	81.90	156.20	0.95GB	1.90GB
	VK (KB)	3.78	3.79	3.78	3.78	3.78	3.78	4.05	4.20
	Prove (s)	0.57	0.73	0.74	1.08	0.89	1.79	6.27	15.22
	Verify (s)	0.006	0.005	0.005	0.006	0.006	0.005	0.006	0.006
Bionetta (Groth16)	Constraints #	29.0K	5.9K	522.4K	779.4K	543.0K	1.56M	12.01M	31.78M
	Proof Size (KB)	0.81	0.81	0.81	0.81	0.81	0.81	0.81	0.81
	PK (MB)	21.30	10.20	396.20	560.20	409.30	1.2GB	≈9.0GB	≈23.8GB
	VK (KB)	3.65	3.65	3.65	3.65	3.65	3.65	≈4.0	≈4.0
	Prove (s)	0.12	0.27	2.19	2.20	2.22	4.72	≈180	≈480
	Verify (s)	0.006	0.006	0.006	0.006	0.006	0.005	≈0.005	≈0.006

Up to x12.7 boost in # of constraints!

How to actually implement?

Surprising result: if the circuit consists of L range-checks, each costing b constraints, using lookup protocol, you can reduce $\mathcal{O}(n)$ constraints ($n = Lb$) down to $\mathcal{O}(n/\log n)$.

How to actually implement?

Surprising result: if the circuit consists of L range-checks, each costing b constraints, using lookup protocol, you can reduce $\mathcal{O}(n)$ constraints ($n = Lb$) down to $\mathcal{O}(n/\log n)$.

Key question: *how do we implement it **in Groth16?*** Since PlonK and SumCheck already have them! (see plookup+logup).

How to actually implement?

Surprising result: if the circuit consists of L range-checks, each costing b constraints, using lookup protocol, you can reduce $\mathcal{O}(n)$ constraints ($n = Lb$) down to $\mathcal{O}(n/\log n)$.

Key question: *how do we implement it **in Groth16?*** Since PlonK and SumCheck already have them! (see plookup+logup).

Theorem (Some stuff from ZKDL Camp)

The inclusion check $\{z_i\}_{i\in[n]} \subseteq \{t_i\}_{i\in[v]}$ is satisfied if and only if there exists the set of multiplicities $\{\mu_i\}_{i\in[v]}$ where $\mu_i = \#\{j \in [n] : z_j = t_i\}$ such that for $\gamma \xleftarrow{\$} \mathbb{F}$:

$$\sum_{i\in[n]} \frac{1}{\gamma + z_i} = \sum_{i\in[v]} \frac{\mu_i}{\gamma + t_i}$$

How to actually implement?

Surprising result: if the circuit consists of L range-checks, each costing b constraints, using lookup protocol, you can reduce $\mathcal{O}(n)$ constraints ($n = Lb$) down to $\mathcal{O}(n/\log n)$.

Key question: *how do we implement it **in Groth16?*** Since PlonK and SumCheck already have them! (see plookup+logup).

Theorem (Some stuff from ZKDL Camp)

The inclusion check $\{z_i\}_{i\in[n]} \subseteq \{t_i\}_{i\in[v]}$ is satisfied if and only if there exists the set of multiplicities $\{\mu_i\}_{i\in[v]}$ where $\mu_i = \#\{j \in [n] : z_j = t_i\}$ such that for $\gamma \xleftarrow{\$} \mathbb{F}$:

$$\sum_{i\in[n]} \frac{1}{\gamma + z_i} = \sum_{i\in[v]} \frac{\mu_i}{\gamma + t_i}$$

High-level idea: We can: (1) compute $\{\mu_i\}_{i\in[v]}$ off-circuit, (2) write circuit in $n + 2v$ constraints, given γ signal is passed randomly.

Circom-like Implementation

```
signal input t[M];          // The lookup table
signal random input gamma; // Random challenge value
signal input z[N];          // The array of values to check

var sum_z, sum_t = 0;
for (var i = 0; i < N; i++) {
    inv_z[i] <== 1 / (z[i] + gamma);
    sum_z += inv_z[i]; // Compute the left-hand side
}

for (var j = 0; j < M; j++) {
    mu[j] <-- 0; // Compute the multiplicities off-circuit
    for (var k = 0; k < N; k++) {
        mu[j] += (t[j] == z[k]);
    }
    inv_t[i] <== mu[j] / (t[j] + gamma);
    sum_t += int_v[i]; // Compute the right-hand side
}

sum_z === sum_t; // Check both sides are equal
```

Problem

```
signal input t[M];           // The lookup table
signal random input gamma; // Random challenge value
signal input z[N];           // The array of values to check

var sum_z, sum_t = 0;
for (var i = 0; i < N; i++) {
    inv_z[i] <== 1 / (z[i] + gamma);
    sum_z += inv_z[i]; // Compute the left-hand side
}

for (var j = 0; j < M; j++) {
    mu[j] <-- 0; // Compute the multiplicities off-circuit
    for (var k = 0; k < N; k++) {
        mu[j] += (t[j] == z[k]);
    }
    inv_t[i] <== mu[j] / (t[j] + gamma);
    sum_t += int_t[i]; // Compute the right-hand side
}

sum_z === sum_t; // Check both sides are equal
```

UltraGroth Explained

Some Historical Notes

- First paper on this problem is "MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs", published in 2020.

Some Historical Notes

- First paper on this problem is "MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs", published in 2020.
- Unaware of this protocol, in 2023 Lev Soukhanov published the post on **UltraGroth**, where he invented *multi-round* MIRAGE.

Some Historical Notes

- First paper on this problem is "MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs", published in 2020.
- Unaware of this protocol, in 2023 Lev Soukhanov published the post on **UltraGroth**, where he invented *multi-round* MIRAGE.
- Likely, unaware of Lev Soukhanov's blog, Alex Ozdemir, Evan Laufer, Dan Boneh published "Volatile and persistent memory for zkSNARKs via algebraic interactive proofs" paper in 2025.

Some Historical Notes

- First paper on this problem is "MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs", published in 2020.
- Unaware of this protocol, in 2023 Lev Soukhanov published the post on **UltraGroth**, where he invented *multi-round* MIRAGE.
- Likely, unaware of Lev Soukhanov's blog, Alex Ozdemir, Evan Laufer, Dan Boneh published "Volatile and persistent memory for zkSNARKs via algebraic interactive proofs" paper in 2025.
- Well... Their construction, called MIRAGE+, is exactly an **UltraGroth**, published back in 2023.

Some Historical Notes

- First paper on this problem is "MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs", published in 2020.
- Unaware of this protocol, in 2023 Lev Soukhanov published the post on **UltraGroth**, where he invented *multi-round* MIRAGE.
- Likely, unaware of Lev Soukhanov's blog, Alex Ozdemir, Evan Laufer, Dan Boneh published "Volatile and persistent memory for zkSNARKs via algebraic interactive proofs" paper in 2025.
- Well... Their construction, called MIRAGE+, is exactly an **UltraGroth**, published back in 2023.

One important consequence

The protocol is **safe**. It is sound and zero-knowledge! And it is now proven in **three** different independent papers.

UltraGroth Performance

Now, let us recap the **Groth16** performance over the circuit of size n and statement size ℓ.

- **Prover work**: MSM of size $\mathcal{O}(n)$ over $\mathbb{G}_1$ and $\mathbb{G}_2$.

UltraGroth Performance

Now, let us recap the **Groth16** performance over the circuit of size n and statement size ℓ.

- **Prover work**: MSM of size $\mathcal{O}(n)$ over $\mathbb{G}_1$ and $\mathbb{G}_2$.
- **Proof size:** $2\mathbb{G}_1 + \mathbb{G}_2$.

UltraGroth Performance

Now, let us recap the **Groth16** performance over the circuit of size n and statement size ℓ.

- **Prover work**: MSM of size $\mathcal{O}(n)$ over $\mathbb{G}_1$ and $\mathbb{G}_2$.
- **Proof size:** $2\mathbb{G}_1 + \mathbb{G}_2$.
- **Verifier work:** 3 pairings + $\mathcal{O}(\ell)$ $\mathbb{G}_1$ exps.

UltraGroth Performance

Now, let us recap the **Groth16** performance over the circuit of size n and statement size ℓ.

- **Prover work**: MSM of size $\mathcal{O}(n)$ over $\mathbb{G}_1$ and $\mathbb{G}_2$.
- **Proof size:** $2\mathbb{G}_1 + \mathbb{G}_2$.
- **Verifier work:** 3 pairings + $\mathcal{O}(\ell)$ $\mathbb{G}_1$ exps.

UltraGroth performance in turn:

- **Prover work**: MSM of size $\mathcal{O}(n/\log n)$ over $\mathbb{G}_1$ and $\mathbb{G}_2$.

UltraGroth Performance

Now, let us recap the **Groth16** performance over the circuit of size n and statement size ℓ.

- **Prover work**: MSM of size $\mathcal{O}(n)$ over $\mathbb{G}_1$ and $\mathbb{G}_2$.
- **Proof size:** $2\mathbb{G}_1 + \mathbb{G}_2$.
- **Verifier work:** 3 pairings + $\mathcal{O}(\ell)$ $\mathbb{G}_1$ exps.

UltraGroth performance in turn:

- **Prover work**: MSM of size $\mathcal{O}(n/\log n)$ over $\mathbb{G}_1$ and $\mathbb{G}_2$.
- **Proof size:** $3\mathbb{G}_1 + \mathbb{G}_2$ (additional 64 bytes).

UltraGroth Performance

Now, let us recap the **Groth16** performance over the circuit of size n and statement size ℓ.

- **Prover work**: MSM of size $\mathcal{O}(n)$ over $\mathbb{G}_1$ and $\mathbb{G}_2$.
- **Proof size:** $2\mathbb{G}_1 + \mathbb{G}_2$.
- **Verifier work:** 3 pairings + $\mathcal{O}(\ell)$ $\mathbb{G}_1$ exps.

UltraGroth performance in turn:

- **Prover work**: MSM of size $\mathcal{O}(n/\log n)$ over $\mathbb{G}_1$ and $\mathbb{G}_2$.
- **Proof size:** $3\mathbb{G}_1 + \mathbb{G}_2$ (additional 64 bytes).
- **Verifier work:** 4 pairings + $\mathcal{O}(\ell)$ $\mathbb{G}_1$ exps + 1 hashing.

UltraGroth Overall Idea

Problem: Compared to PlonK or SumCheck, *Groth16* itself is not derived from the interactive protocol (via Fiat-Shamir).

UltraGroth Overall Idea

Problem: Compared to PlonK or SumCheck, *Groth16* itself is not derived from the interactive protocol (via Fiat-Shamir).

Recap: Proof in Groth16 consists of three points $g_1^{a(\tau)}$, $g_1^{c(\tau)}$, $g_2^{b(\tau)}$:

$$a(X) = \alpha + \sum_{i \in [n]} z_i \ell_i(X) + r\delta, \quad b(X) = \beta + \sum_{i \in [n]} z_i r_i(X) + s\delta,$$

$$c(X) = \delta^{-1} \left(\sum_{i \in \mathcal{I}_W} z_i \zeta_i(X) + h(X)t(X) \right) + a(X)s + b(X)r - rs\delta.$$

UltraGroth Overall Idea

Problem: Compared to PlonK or SumCheck, *Groth16* itself is not derived from the interactive protocol (via Fiat-Shamir).

Recap: Proof in Groth16 consists of three points $g_1^{a(\tau)}$, $g_1^{c(\tau)}$, $g_2^{b(\tau)}$:

$$a(X) = \alpha + \sum_{i \in [n]} z_i \ell_i(X) + r\delta, \quad b(X) = \beta + \sum_{i \in [n]} z_i r_i(X) + s\delta,$$

$$c(X) = \delta^{-1} \left(\sum_{i \in \mathcal{I}_W} z_i \zeta_i(X) + h(X)t(X) \right) + a(X)s + b(X)r - rs\delta.$$

The **verification equation** is:

$$e(\pi_A, \pi_B) = e(g_1^{\alpha}, g_2^{\beta}) \cdot e(g_1^{i(\tau)}, g_2^{\gamma}) \cdot e(\pi_C, g_2^{\delta}).$$

for $\pi_A = g_1^{a(\tau)}$, $\pi_C = g_1^{c(\tau)}$, $\pi_B = g_2^{b(\tau)}$, $i(X)$ is a polynomial derived from the public statement.

UltraGroth Overall Idea

- Do not touch $a(X)$ and $b(X)$.

UltraGroth Overall Idea

- Do not touch $a(X)$ and $b(X)$.
- Split R1CS into two rounds: *round 0* computes the circuit without lookup check, *round 1* imposes lookup check.

UltraGroth Overall Idea

- Do not touch $a(X)$ and $b(X)$.
- Split R1CS into two rounds: *round 0* computes the circuit without lookup check, *round 1* imposes lookup check.
- Split $c(X)$ into $c_0(X)$ and $c_1(X)$.

UltraGroth Overall Idea

- Do not touch $a(X)$ and $b(X)$.
- Split R1CS into two rounds: *round 0* computes the circuit without lookup check, *round 1* imposes lookup check.
- Split $c(X)$ into $c_0(X)$ and $c_1(X)$.
- $c_0(X)$ is derived from *round 0*'s witness.

UltraGroth Overall Idea

- Do not touch $a(X)$ and $b(X)$.
- Split R1CS into two rounds: *round 0* computes the circuit without lookup check, *round 1* imposes lookup check.
- Split $c(X)$ into $c_0(X)$ and $c_1(X)$.
- $c_0(X)$ is derived from *round 0*'s witness.
- Form point $\pi_C^{\langle 0 \rangle} \leftarrow g_1^{c_0(\tau)}$ and sample randomness $\gamma \leftarrow \mathcal{H}(\pi_C^{\langle 0 \rangle})$.

UltraGroth Overall Idea

- Do not touch $a(X)$ and $b(X)$.
- Split R1CS into two rounds: *round 0* computes the circuit without lookup check, *round 1* imposes lookup check.
- Split $c(X)$ into $c_0(X)$ and $c_1(X)$.
- $c_0(X)$ is derived from *round 0*'s witness.
- Form point $\pi_C^{\langle 0 \rangle} \leftarrow g_1^{c_0(\tau)}$ and sample randomness $\gamma \leftarrow \mathcal{H}(\pi_C^{\langle 0 \rangle})$.
- Compute witness for *round 1* using γ, form $c_1(X)$ and thus compute $\pi_C^{\langle 1 \rangle} \leftarrow g_1^{c_1(\tau)}$.

UltraGroth Overall Idea

- Do not touch $a(X)$ and $b(X)$.
- Split R1CS into two rounds: *round 0* computes the circuit without lookup check, *round 1* imposes lookup check.
- Split $c(X)$ into $c_0(X)$ and $c_1(X)$.
- $c_0(X)$ is derived from *round 0*'s witness.
- Form point $\pi_C^{\langle 0 \rangle} \leftarrow g_1^{c_0(\tau)}$ and sample randomness $\gamma \leftarrow \mathcal{H}(\pi_C^{\langle 0 \rangle})$.
- Compute witness for *round 1* using γ, form $c_1(X)$ and thus compute $\pi_C^{\langle 1 \rangle} \leftarrow g_1^{c_1(\tau)}$.
- Output proof as $\pi \leftarrow (\pi_A, \pi_C^{\langle 0 \rangle}, \pi_C^{\langle 1 \rangle}, \pi_B)$.

UltraGroth Overall Idea

- Do not touch $a(X)$ and $b(X)$.
- Split R1CS into two rounds: *round 0* computes the circuit without lookup check, *round 1* imposes lookup check.
- Split $c(X)$ into $c_0(X)$ and $c_1(X)$.
- $c_0(X)$ is derived from *round 0*'s witness.
- Form point $\pi_C^{\langle 0 \rangle} \leftarrow g_1^{c_0(\tau)}$ and sample randomness $\gamma \leftarrow \mathcal{H}(\pi_C^{\langle 0 \rangle})$.
- Compute witness for *round 1* using γ, form $c_1(X)$ and thus compute $\pi_C^{\langle 1 \rangle} \leftarrow g_1^{c_1(\tau)}$.
- Output proof as $\pi \leftarrow (\pi_A, \pi_C^{\langle 0 \rangle}, \pi_C^{\langle 1 \rangle}, \pi_B)$.

The **verification equation** is:

$$e(\pi_A, \pi_B) = e(g_1^{\alpha}, g_2^{\beta}) \cdot e(g_1^{i(\tau)}, g_2^{\gamma}) \cdot e(\pi_C^{\langle 0 \rangle}, g_2^{\delta_0}) \cdot e(\pi_C^{\langle 1 \rangle}, g_2^{\delta}).$$

UltraGroth Overall Idea

- Do not touch $a(X)$ and $b(X)$.
- Split R1CS into two rounds: *round 0* computes the circuit without lookup check, *round 1* imposes lookup check.
- Split $c(X)$ into $c_0(X)$ and $c_1(X)$.
- $c_0(X)$ is derived from *round 0*'s witness.
- Form point $\pi_C^{\langle 0 \rangle} \leftarrow g_1^{c_0(\tau)}$ and sample randomness $\gamma \leftarrow \mathcal{H}(\pi_C^{\langle 0 \rangle})$.
- Compute witness for *round 1* using γ, form $c_1(X)$ and thus compute $\pi_C^{\langle 1 \rangle} \leftarrow g_1^{c_1(\tau)}$.
- Output proof as $\pi \leftarrow (\pi_A, \pi_C^{\langle 0 \rangle}, \pi_C^{\langle 1 \rangle}, \pi_B)$.

The **verification equation** is:

$$e(\pi_A, \pi_B) = e(g_1^{\alpha}, g_2^{\beta}) \cdot e(g_1^{i(\tau)}, g_2^{\gamma}) \cdot e(\pi_C^{\langle 0 \rangle}, g_2^{\delta_0}) \cdot e(\pi_C^{\langle 1 \rangle}, g_2^{\delta}).$$

Note: This construction can be easily generalized for $d > 1$ rounds.

Our Contribution

- Implemented a single-round UltraGroth (essentially, a Mirage protocol). Credits to Artem Sdobnov, Vitalii Volovyk, Yevhenii Sekhin, and Illia Dovgopoly.

Our Contribution

- Implemented a single-round UltraGroth (essentially, a Mirage protocol). Credits to Artem Sdobnov, Vitalii Volovyk, Yevhenii Sekhin, and Illia Dovgopoly.
  - Forked `rapidsnark`.

Our Contribution

- Implemented a single-round UltraGroth (essentially, a Mirage protocol). Credits to Artem Sdobnov, Vitalii Volovyk, Yevhenii Sekhin, and Illia Dovgopoly.
  - Forked `rapidsnark`.
  - Forked `snarkjs` for witness export/verify functions and smart-contract autogeneration.

Our Contribution

- Implemented a single-round UltraGroth (essentially, a Mirage protocol). Credits to Artem Sdobnov, Vitalii Volovyk, Yevhenii Sekhin, and Illia Dovgopoly.
    - Forked `rapidsnark`.
    - Forked `snarkjs` for witness export/verify functions and smart-contract autogeneration.
    - Thanks to Ivan Lele, we even have a Swift SDK for that!

Our Contribution

- Implemented a single-round UltraGroth (essentially, a Mirage protocol). Credits to Artem Sdobnov, Vitalii Volovyk, Yevhenii Sekhin, and Illia Dovgopoly.
  - Forked `rapidsnark`.
  - Forked `snarkjs` for witness export/verify functions and smart-contract autogeneration.
  - Thanks to Ivan Lele, we even have a Swift SDK for that!
- Proved completeness, soundness, and zero-knowledge for general d-round UltraGroth. Formalized everything properly.

Our Contribution

- Implemented a single-round UltraGroth (essentially, a Mirage protocol). Credits to Artem Sdobnov, Vitalii Volovyk, Yevhenii Sekhin, and Illia Dovgopoly.
  - Forked `rapidsnark`.
  - Forked `snarkjs` for witness export/verify functions and smart-contract autogeneration.
  - Thanks to Ivan Lele, we even have a Swift SDK for that!
- Proved completeness, soundness, and zero-knowledge for general d-round UltraGroth. Formalized everything properly.
- Applied UltraGroth to Bionetta and obtained incredible results.

Any Questions?

♥

distributedlab.com

github.com/rarimo/ultragroth