

Benchmarking the Plonk, TurboPlonk, and UltraPlonk Proving Systems

Lehigh University

Tal Derei, Caleb Geren, Michael Kaufman, Jon Klein, Rishad Islam Shantho, Victor
Carolino

February 2023

1 Abstract

PlonK (Permutations over Lagrange-bases for Oecumenical Non-Interactive Arguments of Knowledge) [\[1\]](#) is a zero-knowledge proving system that allows a prover to convince a verifier they possess certain information without revealing that information. We evaluate the performance and scalability of the Plonk, TurboPlonk and UltraPlonk zero-knowledge proof systems using custom gates based on Pedersen hashes and lookup arguments. This expands on our prior performance benchmarks of standard Plonk [\[2\]](#). All measurements are collected from Aztec’s Barretenberg cryptographic library and backend running on CPUs [\[3\]](#).

2 Contribution

We extended Aztec’s benchmarks [\[4\]](#) for larger circuit sizes and more hash computations. This demands a performant machine with expansive memory and many processor cores. The circuits in our experiments are limited to a maximum constraint size of 2^{26} and 128K hashes. The transcript from the trusted setup ceremony restricts circuits to an upper bound of **~100 million** constraints [\[5\]](#). In order to support larger circuits with more constraints, a trusted setup with a larger structured proving key is required.

Our benchmarks make a distinction between **Task Size** (the number of hash computations) and **Problem Size** (the circuit size). These techniques measure the execution time and memory consumption for these proving systems. The results show execution time and memory requirements growing log-linearly, $O(n \log n)$ time complexity, with respect to the number of constraints in the circuit. This is consistent with the asymptotic growth rates described in literature [6]. The following charts display only the x-axis in logarithmic scale to emphasize the execution time.

3 Background

The following section reviews the basics of the TurboPlonk and UltraPlonk proof constructions, and highlights the importance of zero-knowledge friendly hash functions in production blockchain systems.

3.1 TurboPlonk and UltraPlonk

Plonk circuits are typically expressed with a fan-in-2 and fan-out-1 gate structure. This means a logical gate, representing either addition or multiplication operations, can handle two inputs and a single output. TurboPlonk generalizes the constraint system by introducing custom gates that represent complicated statements with fewer gates, for expressing the same computation, in a circuit [7]. Custom gates represent more complex operations in a single gate, reducing the number of total gates in the circuit. For instance, cryptographic primitives like fixed-base elliptic curve scalar-multiplication, elliptic curve point arithmetic, and bitwise XOR / AND can be expressed and evaluated with a single custom gate. Consequently, it uses an additional wire, requiring an additional commitment to be computed. This will be described in further detail below.

UltraPlonk extends this construction with precomputed lookup tables, which represent efficient key-value mappings [8]. This enables a prover to prove a witness exists in

a table instead of proving the computation itself. This ultimately reduces the circuit size. The protocol preprocesses a table $T = \{0, \dots, 2^n - 1\}$ and devises a way to check $x \in T$. These lookup tables are commonly used to avoid expensive bit-decompositions for bitwise operations. For example, rather than computing a XOR operation bit by bit, the 8-bit result is encoded in the table and accessed with a lookup operation. Bit-decompositions are expensive because each bit is represented as a finite field element.

3.2 SNARK-Friendly Hashing Algorithms

Computing a SHA-256 or Keccak hash is a frequent and expensive operation in a circuit. SNARK-friendly hashing algorithms like Pedersen [\[9\]](#) make this more efficient by performing the same computation with fewer constraints. They are primarily used for providing collision resistance for Merkle-trees, where hashing dominates ~99% of the runtime for merkle-tree based operations. For example, in UTXO-based chains like Bitcoin, assets are recorded in the form of ‘notes’ of ownership. These notes are stored in binary hash trees known as Merkle-trees. The two types of Merkle-trees used are:

1. **Note Tree:** 2^{30} dense merkle-tree that stores all output notes created.
2. **Nullifier Tree:** 2^{256} sparse merkle-tree that stores copies of spent notes.

A transaction involves adding a single note to both Merkle-trees, costing 60 hashes in total. Each SHA-256 hash in standard Plonk requires ~27,000 gates, and a single transaction consumes a total of 1.6m gates [\[10\]](#).

3 Cloud Computing Environment

We measured the relative speed up in performance on the following bare-metal machine instantiated on Oracle Cloud: 32-core Intel(R) Xeon(R) Platinum 8358 CPU @ 2.60GHz, 1024 GB DDR4 DRAM, 128 GB SSD.

4 Experimental Results

The raw benchmarks can be found here [\[11\]](#).

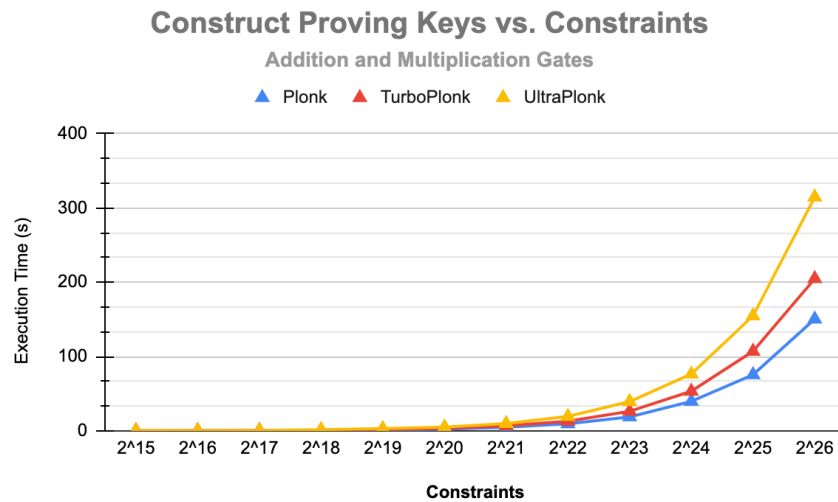
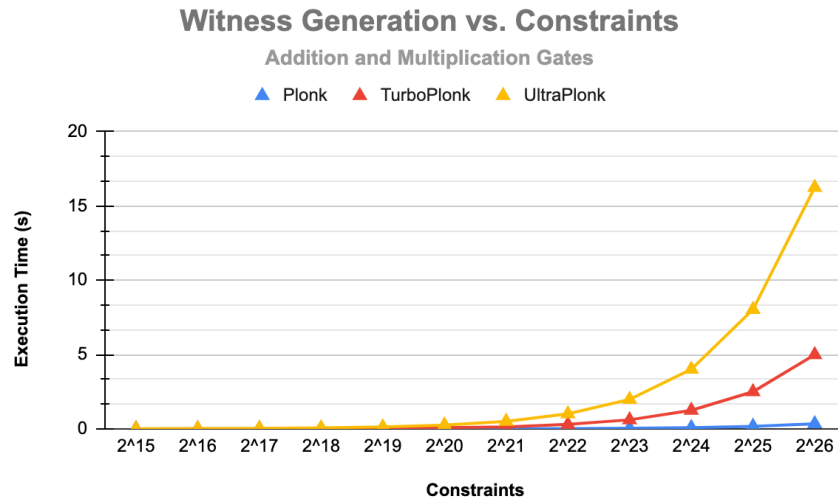
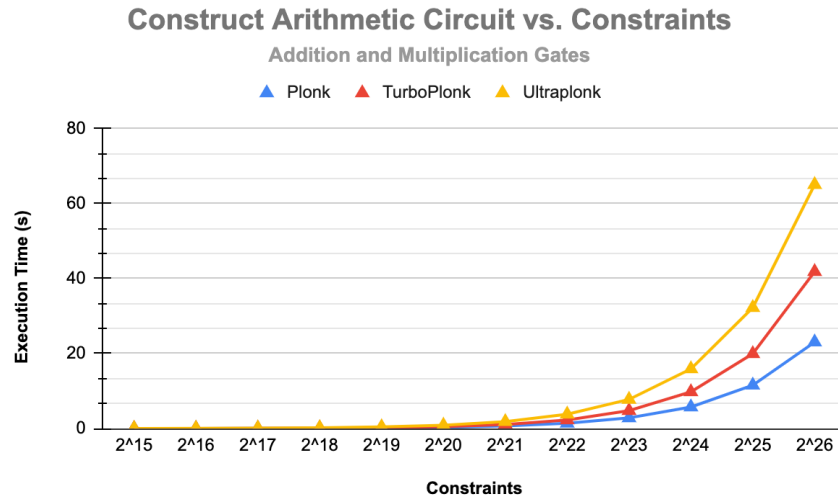
We measured performance using the Oracle Cloud Infrastructure (OCI) service and real-time process monitoring tools. We enabled multithreading using the OpenMP API, BMI2 x86-64 assembly instructions, and Clang compiler optimizations.

The workload for the prover is divided into the following tasks:

1. Construct the arithmetic circuit
2. Calculate witness polynomials
3. Compute the proving key (including q_l , q_r , q_o , sigma polynomials, etc.)
4. Compute the verifier key
5. Generate proof
6. Verify proof

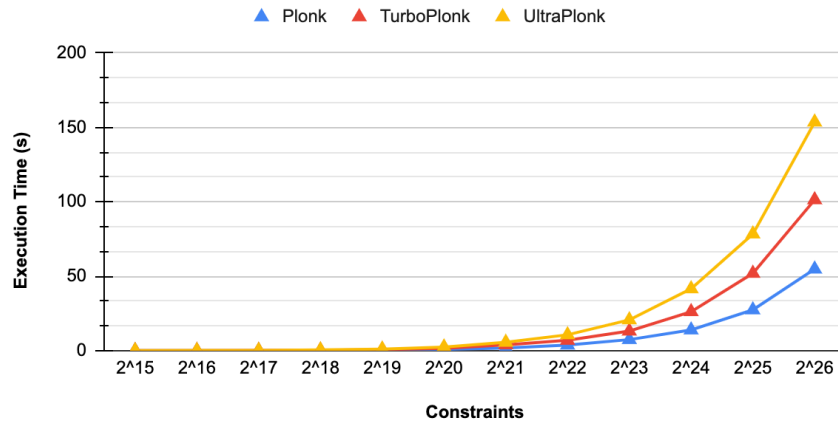
5.1 Addition and Multiplication Gates

The following charts highlight the performance of Plonk, TurboPlonk, and UltraPlonk using *only* addition and multiplication gates. TurboPlonk and UltraPlonk exhibit worse performance in this setting because they are structured to optimize performance in the presence of custom gates. In general, performance and memory are proportional to the number of gates in the arithmetic circuit. The distribution of gates in our circuit is 75% multiplication and 25% addition gates. For simplicity, we assume addition and multiplication gates contribute similarly to the prover cost.



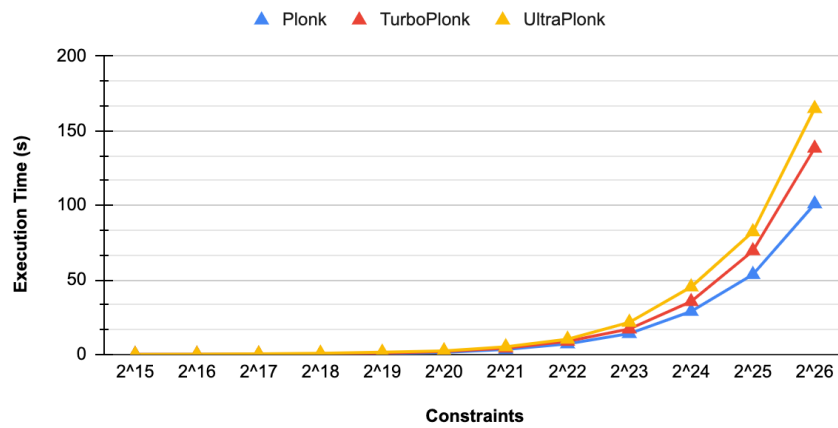
Construct Verifier Keys vs. Constraints

Addition and Multiplication Gates



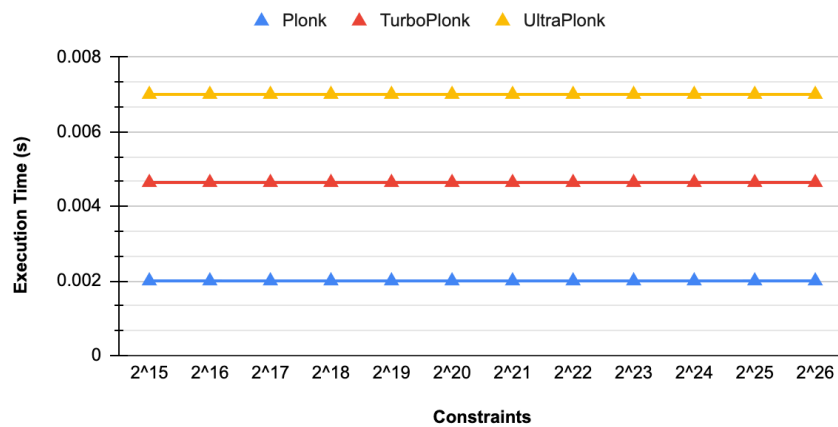
Proof Generation vs. Constraints

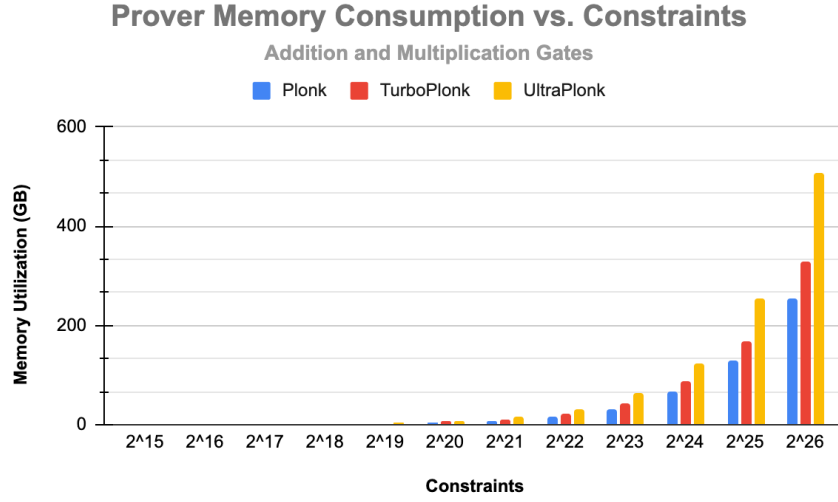
Addition and Multiplication Gates



Proof Verification vs. Constraints

Addition and Multiplication Gates





Figures 1 – 7: Prover workloads for multiplication and addition gates

For 2^{26} constraints, generating a TurboPlonk proof ($\sim 138s$) is 37% slower and UltraPlonk ($\sim 165s$) is 63% slower compared to generating a Plonk proof ($\sim 101s$). UltraPlonk is 19% slower than TurboPlonk in the same setting.

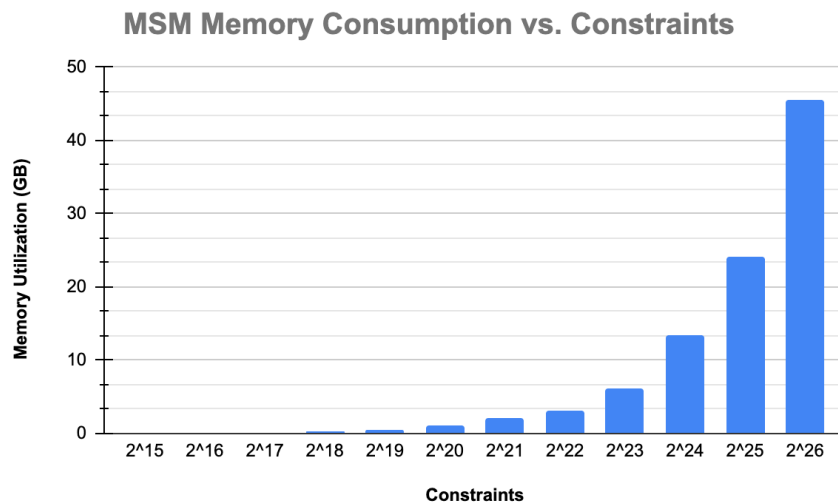
TurboPlonk and UltraPlonk further display larger memory consumption footprints than Plonk. For 2^{26} constraints, generating a TurboPlonk proof consumed $\sim 30\%$ more memory (330 GB) and UltraPlonk consumed $\sim 99\%$ more memory (506 GB) with respect to a Plonk proof (255 GB).

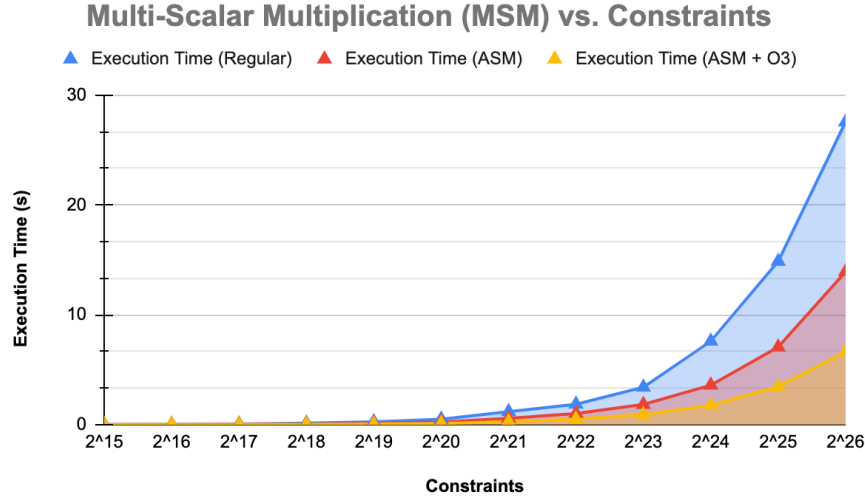
In this setting, with only addition and multiplication gates, Plonk is more efficient because it requires fewer group exponentiations. Consequently, less prover work translates to reduced proof generation time and memory consumption. If ‘ N ’ represents the number of multiplication gates in the circuit, Plonk requires $9N$ scalar multiplications and a proof size of ~ 9 G1 curve elements, TurboPlonk requires $11N$ scalar multiplications and a proof size of ~ 11 G1 curve elements, and UltraPlonk requires $13N$ scalar multiplications and a proof size of ~ 13 G1 curve elements. [1]. Group exponentiations dominate approximately 70–80% of the prover runtime.

Furthermore, a single multi-scalar multiplication for 2^{26} constraints executes in **~6.7** seconds and consumes **45.5 GB** of memory, shown in the chart below. The prover memory usage of **255 GB**, **330 GB**, and **506 GB** are significantly higher than the multi-scalar multiplication memory of **45.5 GB**. All three proving systems compute a collection of polynomials and stores each in 3 forms:

1. Coefficient form: $n * p$
2. Lagrange form: $n * p$
3. Coset-FFT form: $4n * p$

where ' N ' represents the number of constraints in the circuit, ' P ' represents the polynomial. It is possible to run the plonk prover by storing just the coefficient form, which would reduce the memory consumption by a factor of **6**. This aligns with the memory consumption for computing multi-scalar multiplications. This comes at an additional runtime computational cost resulting from not pre-computing and storing these polynomials in all three forms. This variable cost is beyond the scope of our benchmarks.





Executing these workloads on a powerful server with extensive hardware resources merits evaluating the tradeoffs between computational efficiency and memory consumption. Memory is less likely to be a bottleneck compared to processing power. With the decreasing costs of consumer memory chips, it is reasonable for a server to have access to multiple terabytes of DDR4 dynamic random-access memory (DRAM). A larger memory profile can often lead to better performance and computational efficiency through reduced disk IO, faster memory access times, reduced memory swapping, and better caching.

5.2 Custom Gates and Lookup Tables

The following charts highlight the performance of Plonk, TurboPlonk, and UltraPlonk proving systems for executing pedersen hashes. TurboPlonk and UltraPlonk employ custom hash gates and lookup tables for greater efficiency gains and improved performance.

We expect performance to improve for TurboPlonk and UltraPlonk, compared to **section 5.1**, because the custom gates we are using replace gates used in the circuit. We expect further improvement in UltraPlonk since repeated computations are replaced with a table lookup. For example, suppose we have a custom gate to calculate $y = f(x)$ for some complex function, or a lookup table to encode a $y = g(x)$ mapping. If these operations are

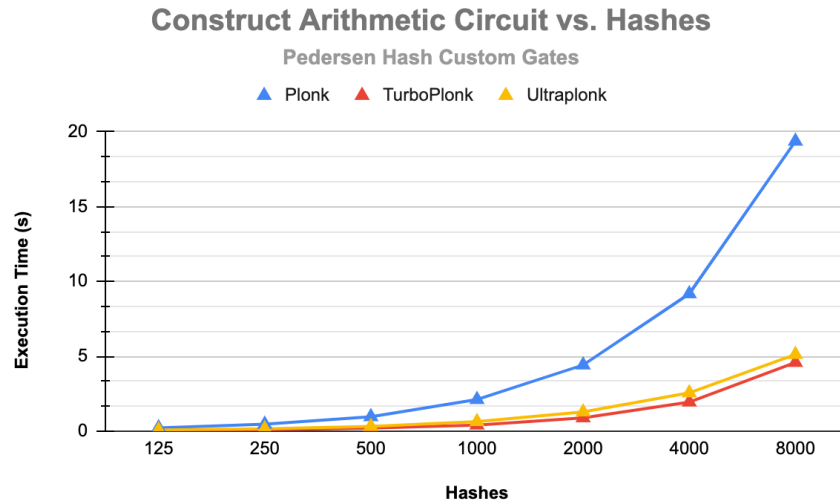
sparsely used in large circuits, then TurboPlonk and UltraPlonk will be slower compared to Plonk. This is supported by the results of the previous section.

5.2.1 Comparison of Pedersen Hashes

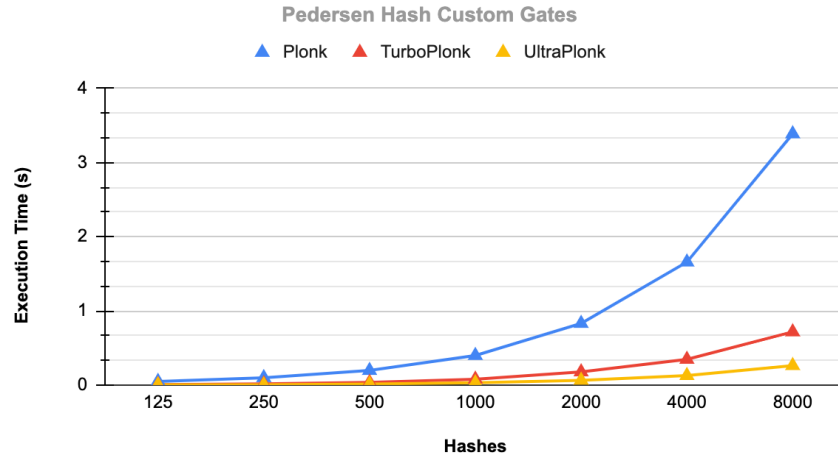
We compare the execution time and memory usage for 125 – 8K Pedersen hashes. Pedersen hashes are computationally expensive operations in terms of prover workload, and are designed to stress the system. Past 8K hashes, the performance of Plonk dramatically degrades. Computing arithmetic hashes in Plonk is expensive in terms of the number of gates per hash. This dramatically blows up the circuit size, resulting in slower proof generation. We expand on this relationship in [section 5.2.3](#).

It's worth noting that without assembly instructions and compiler optimizations enabled, these workloads would not be practical and proof generation would generally be **~4x slower**. The results show that execution time and memory consumption grow log-linearly with respect to the number of constraints in the circuit.

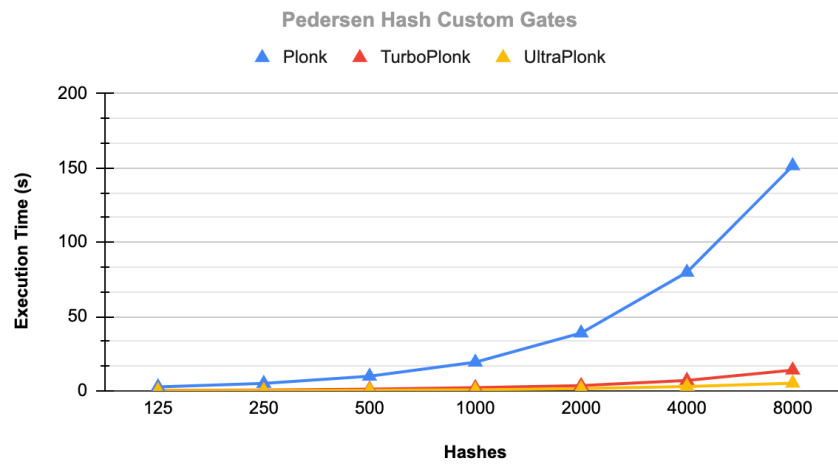
Figures 10 – 16: Prover workloads for custom gates based on pedersen hashes



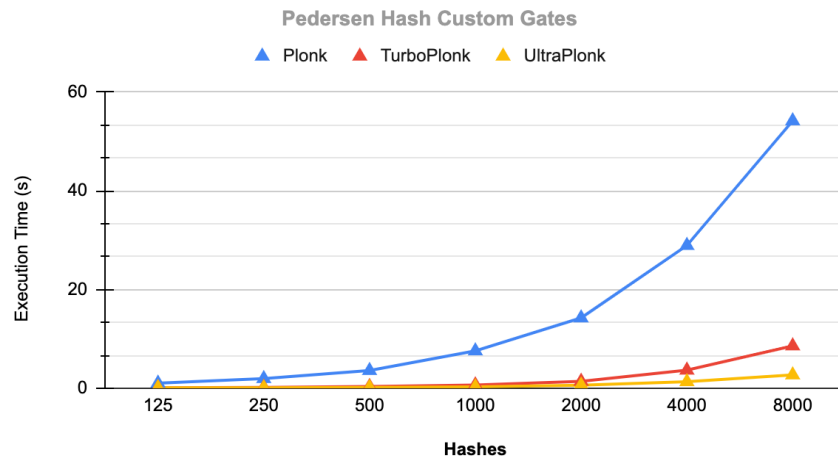
Witness Generation vs. Hashes



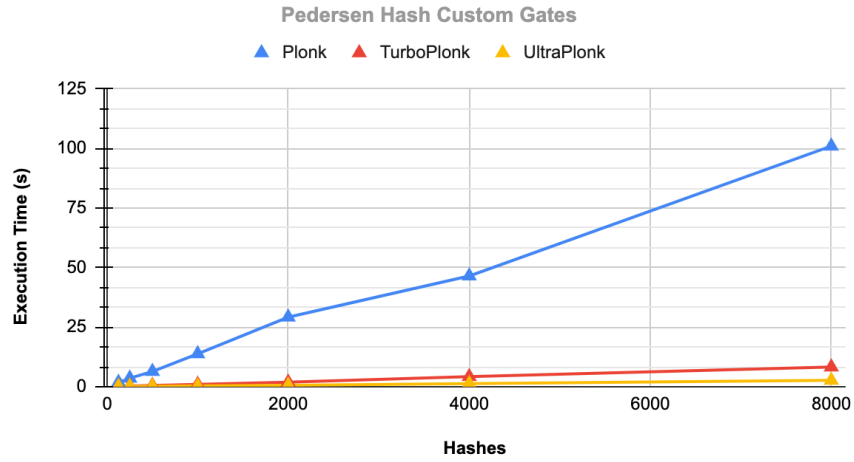
Construct Proving Keys vs. Hashes



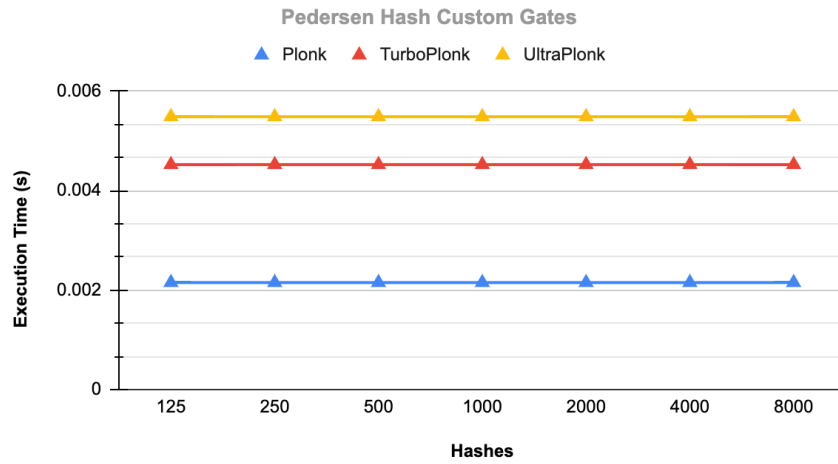
Construct Verifier Keys vs. Hashes



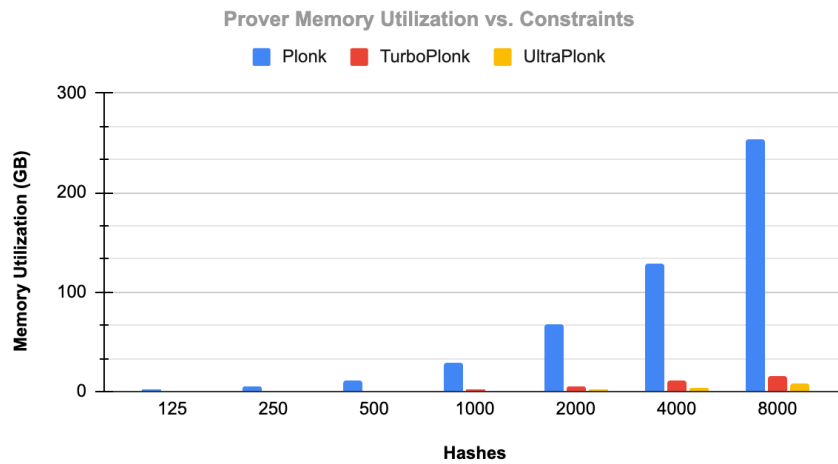
Proof Generation vs. Hashes



Proof Verification vs. Hashes



Prover Memory Consumption vs. Hashes



For **8K** hashes, generating a TurboPlonk proof (**~8s**) is **~12x** faster and UltraPlonk (**~3s**) is **~38x** faster compared to a standard Plonk proof (**~100s**). UltraPlonk is **~3x** faster than TurboPlonk in the same setting.

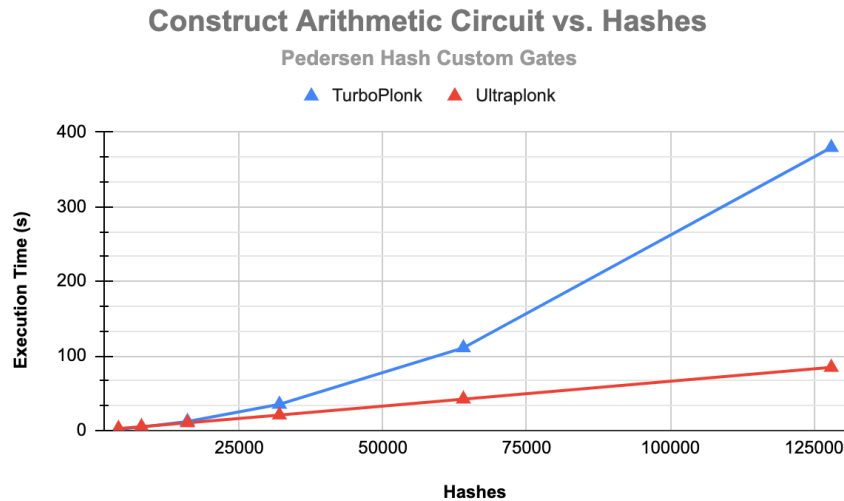
Plonk further displays larger memory consumption. Generating a TurboPlonk proof consumed **~16x** less memory (**~16 GB**) and UltraPlonk **~32x** less memory (**~8 GB**) compared to a standard Plonk proof (**254 GB**).

The memory consumption of Plonk is greater than TurboPlonk, as expected. But the memory of TurboPlonk is greater compared to UltraPlonk, which is contrary to our initial expectations. This is due to us evaluating the proof systems based on the same task size, i.e. **8K** hashes, rather than the same problem size, i.e. 2^{25} constraints. In practice, determining which is more reasonable depends on the specific workload or application. **Section 5.2.2** describes this distinction in greater detail for **128K** hashes.

5.2.2 Comparison of Pedersen Hashes

This extends section 4.2.1, comparing TurboPlonk and UltraPlonk for **4k - 128k** Pedersen hashes.

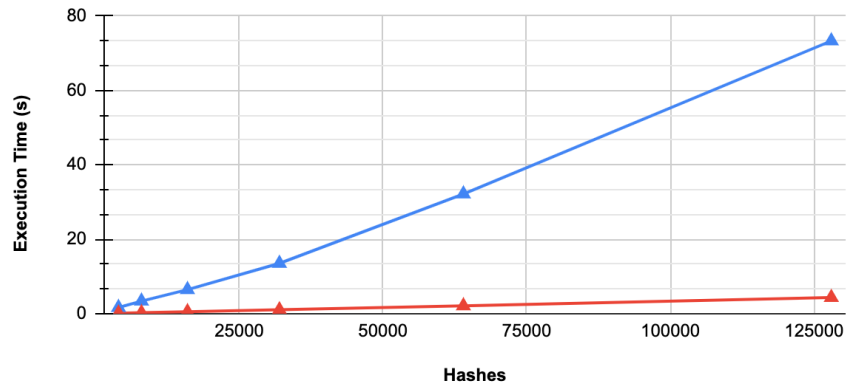
Figures 17 - 23: Prover workloads for custom gates based on Pedersen hashes



Witness Generation vs. Hashes

Pedersen Hash Custom Gates

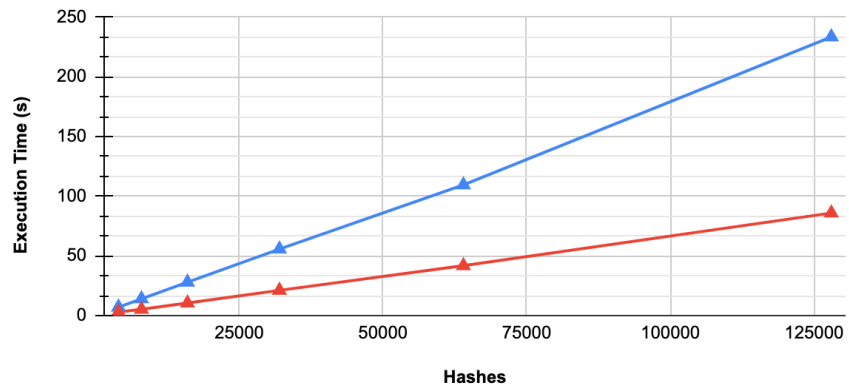
▲ TurboPlonk ▲ UltraPlonk



Construct Proving Keys vs. Hashes

Pedersen Hash Custom Gates

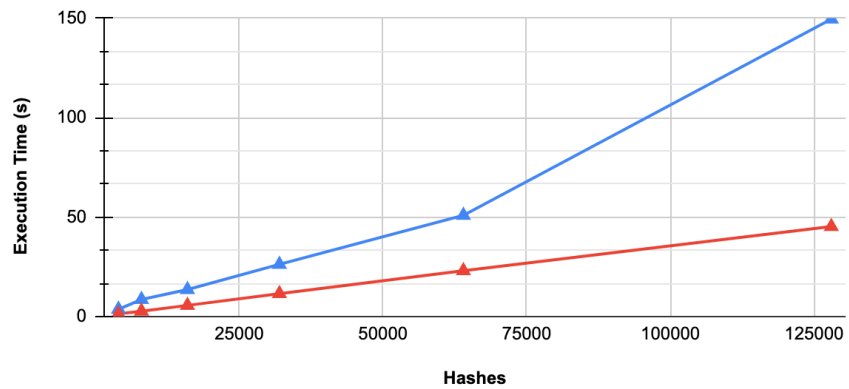
▲ TurboPlonk ▲ UltraPlonk



Construct Verifier Keys vs. Hashes

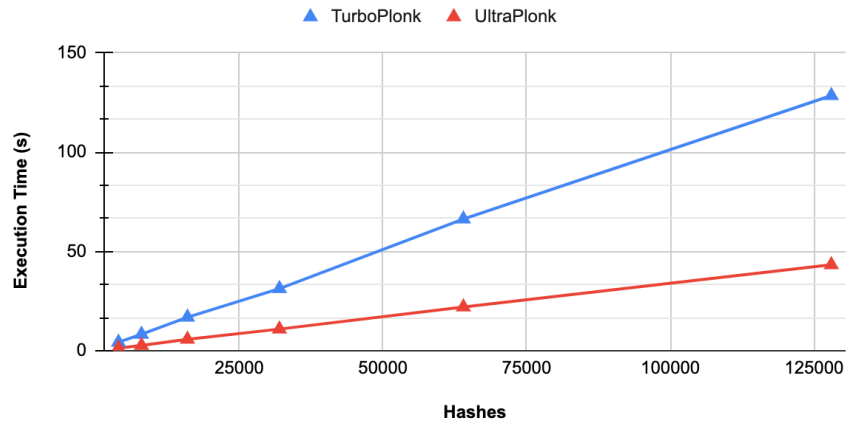
Pedersen Hash Custom Gates

▲ TurboPlonk ▲ UltraPlonk



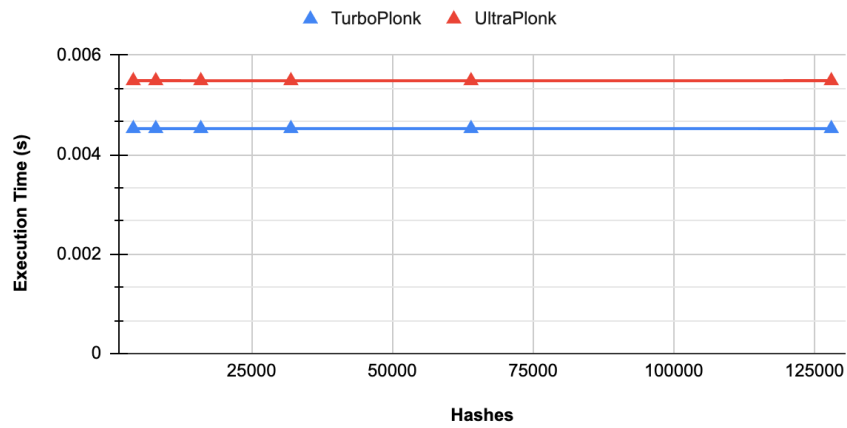
Proof Generation vs. Hashes

Pedersen Hash Custom Gates



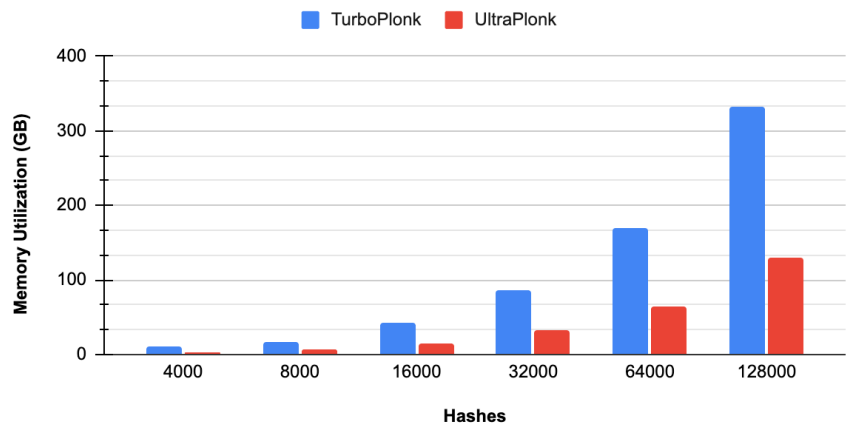
Proof Verification vs. Hashes

Pedersen Hash Custom Gates



Prover Memory Consumption vs. Hashes

Pedersen Hash Custom Gates



Prover times for UltraPlonk (~42s) were ~3x faster than TurboPlonk (~125s), and the memory consumption for UltraPlonk (~130 GB) was ~2.5x smaller than TurboPlonk (~330 GB). The verification time for UltraPlonk was slightly slower than TurboPlonk since there is an inverse tradeoff between proof generation and verification. In general, fast provers have large proofs with slow verifiers, while slow provers have small proofs with fast verifiers.

In the figures shown, generating a proof for 128K hashes requires radically different circuit sizes between TurboPlonk and UltraPlonk. For instance, TurboPlonk requires 13,191,211 constraints, and UltraPlonk requires 44,184,152 constraints:

→ **UltraPlonk:** 13,191,211 constraints / 103 gates per hash = **128K hashes**

→ **TurboPlonk:** 44,184,152 constraints / 345 gates per hash = **128K hashes**

These figures are comparable because they compute the same number of hash operations. TurboPlonk ultimately has ~2.5x increased memory consumption than UltraPlonk when evaluating 128K hashes because it exhibits a ~3.3x increase in circuit size. TurboPlonk has more gates per hash than UltraPlonk, since UltraPlonk employs lookup tables that reduce the circuit's constraint size. Therefore, 128K hashes in TurboPlonk will be more expensive than 128K hashes in UltraPlonk relative to the number of gates, execution time and memory consumption.

We can approximate these memory figures through rough calculations. Since each polynomial is stored in 3 forms, as described in **Section 5.1**, this yields a total memory usage of $6n * p$. We can decompose the total memory by recognizing each polynomial costs $6 * n$. Plonk has 4 witnesses and 5 selectors (9 polynomials), TurboPlonk has 4 witnesses and 11 selectors (15 polynomials), and UltraPlonk has 4 witnesses and 15 selectors (19 polynomials). We ignore the permutation and linearization polynomials for this rough calculation, which

adds a couple more polynomials. The estimated memory consumed by these systems would be:

Plonk: $9 * 6 * n$

TurboPlonk: $15 * 6 * n$

UltraPlonk: $19 * 6 * n$

Recall, the number of constraints ‘ n ’ will be different for each proof system when evaluating 128K hashes. The memory consumption ratio of TurboPlonk to UltraPlonk is $(15 * 6 * n) / (19 * 6 * n) = 3,976,573,680 / 1,503,798,054 = 2.64$. This estimation is consistent with the 2.5x increase in memory consumption from the charts.

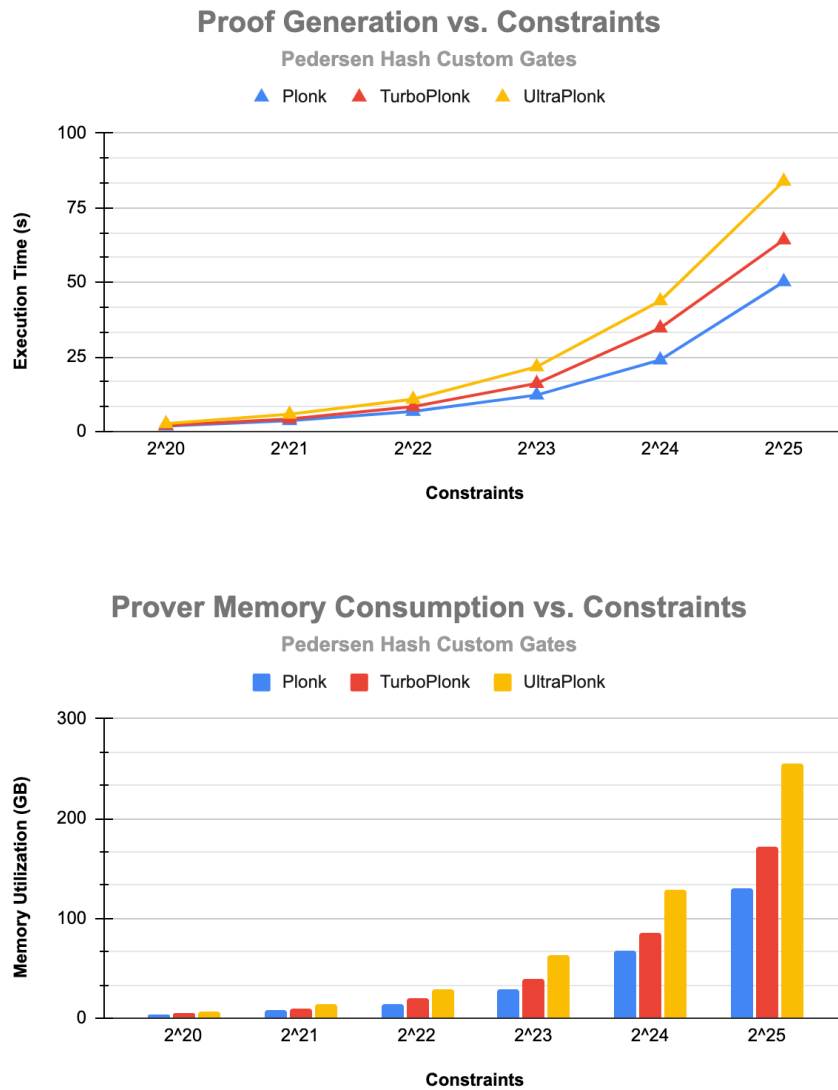
In general, custom gates increase the degree of the identity to be proven, which increases the computational work for the prover. This is in tandem with the reduction in circuit size. For example, the degree of the identity may double, leading to increased computational work for the prover, while the circuit size may be cut in half [\[12\]](#). This simultaneously increases prover work for the custom gate operation and reduces the circuit size. If the latter has a greater effect on reducing the prover work than the increase in prover time incurred by the former, there is a net positive effect. Although we did not benchmark this relationship directly, the tradeoffs need to be considered for different applications and use-cases.

In **section 5.2.3**, the memory consumption for UltraPlonk will be higher than TurboPlonk because we are evaluating the proof systems based on the same problem size, i.e. 2^{25} constraints, rather than the same task size, i.e. **8K** or **128K** hashes, as in **sections 5.2.1** and **5.2.2** respectively.

5.2.3 Comparison of Circuit Size

In this section, we compare the proving systems based on circuit size rather than the number of hashes. The following charts highlight that for 2^{25} constraints, proof generation took 50s (Plonk), 64s (TurboPlonk), and 84s (UltraPlonk). The memory consumption was 129 GB (Plonk), 171 GB (TurboPlonk), and 256 GB (UltraPlonk). The verification time was constant at approximately 2 – 5 ms.

Figures 24 –25: Proof generation and memory consumption for workloads based on circuit size



When fixing the circuit size and evaluating the throughput as a function of the number of hashes it can process: Plonk requires 5113 gates per hash, TurboPlonk requires 345 gates per

hash, and UltraPlonk requires 103 gates per hash. For circuits with 2^{25} constraints, Plonk processed 6,562 hashes, TurboPlonk processed 97,259 hashes, and UltraPlonk processed 325,771 hashes. TurboPlonk and UltraPlonk were able to prove 14.8x and 49.6x more hashes than Plonk respectively for the same circuit size. UltraPlonk was able to prove ~3.4x more hashes than TurboPlonk. In summary, if the circuit sizes of TurboPlonk and UltraPlonk are the same, they must compute a different number of hashes.

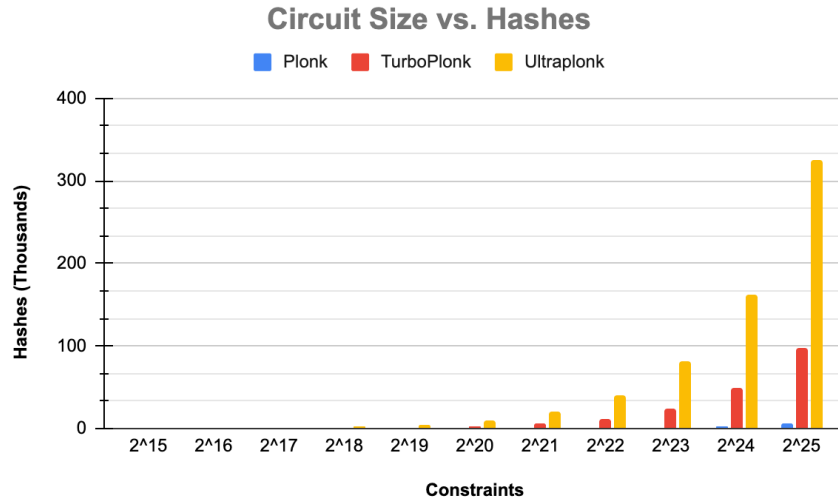


Figure 26: Comparing number of hashes and circuit size

In this setting, UltraPlonk has the highest memory profile because it has more selector polynomials than TurboPlonk, and each selector polynomial is stored in three forms. The increase in memory consumption is not due to the precomputed lookup table because the table sizes do not exceed 2^{16} constraints in the current configuration.

Exceptionally large lookup tables, used primarily for performing efficient range queries where x is in the range $0 \leq x \leq 2^n$, can add unnecessary overhead. UltraPlonk requires computing extra prover commitments for table operations, and a larger table size can add extra prover work even though conventional indexing has $O(1)$ time complexity. In some cases, the prover may need to add dummy gates into the circuit if the table size is

significantly larger than the number of constraints [13]. This increases the circuit size, requiring more prover computation. Memory consumption also increases as a larger precomputed table needs to be stored in system memory.

For instance, in Plookup [14], the results are amortized and you can check m different x 's are in table T in $O(m + N)$ constraints, where N is the number of entries in the table. Culminating from the progress made in Caulk [15], the techniques developed in Cached Quotients (referenced cq) [16] is such that the prover does not pay for the table size through a preprocessing phase. After $O(n \log n)$ preprocessing, you can check $x \in T$ in $O(1)$ constraints [17]. This means prover complexity is independent of the table size. These protocols are beyond the scope of the results presented in this paper.

6 Conclusion

For the Plonk, TurboPlonk, and UltraPlonk proving systems, we make a distinction between **Task Size** (the number of hash computations) and **Problem Size** (the circuit size with N constraints). These are techniques for measuring the execution time and memory consumption for these systems. In practice, UltraPlonk has the most efficient proof generation time for most applications in the presence of custom gates and lookup tables, but consumes the most memory.

7 Future Research

We obtained our results using a single machine running a 32-core Intel CPU. Our next objective is executing components of the prover on a single NVIDIA GPU using the CUDA programming language, and then distributing the workload across multiple GPUs to take advantage of massive parallelism.

8 Acknowledgements

This work was supported in part by Oracle Cloud and related resources provided by the Oracle for Research program. The benchmarking data shown here was run on Oracle Cloud. Other support for this work includes a gift from Steel Perlot and Google, and a Lehigh CORE grant. We thank Suyash Bagad from the Aztec network for his insights into the TurboPlonk and UltraPlonk proving systems. We thank Maxim Vezenov from Aztec and Benjamin Aulenbach from Macguyver Tech for their review and comments.

9 References

- [1] PlonK: Permutations over Lagrange-bases for Oecumenical Noninteractive Arguments of Knowledge: <https://eprint.iacr.org/2019/953>
- [2] Benchmarking Plonk Proving System:
<https://github.com/TalDerei/Masters-Research/blob/main/Benchmarking%20PlonK%20Proving%20System.pdf>
- [3] Barretenberg:
<https://github.com/AztecProtocol/barretenberg>
- [4] PlonK Benchmarks II — ~5x faster than Groth16 on Pedersen Hashes:
<https://medium.com/aztec-protocol/plonk-benchmarks-ii-5x-faster-than-groth16-on-pedersen-hashes-ea5285353db0>
- [5] Aztec Ignition Trusted-Setup Ceremony
https://www.google.com/url?q=https://github.com/AztecProtocol/ignition-verification&sa=D&source=docs&ust=1677326974436967&usg=AOvVaw0kjIE_pY9K4r7ljWx9gNji
- [6] Implementation and Optimization of Zero-Knowledge Proof Circuit Based on Hash Function SM3
<https://www.mdpi.com/1424-8220/22/16/5951>
- [7] The Turbo-Plonk Program Syntax for Specifying SNARK Programs:
<https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo-plonk.pdf>
- [8] Plookup: A Simplified Polynomial Protocol for Lookup Tables:
<https://eprint.iacr.org/2020/315.pdf>
- [9] SNARK-Friendly Hash Functions:
https://hackmd.io/@yelhousni/snark_hash?utm_source=preview-mode&utm_medium=rec
- [10] Why Hashes Dominate in SNARKs

- <https://medium.com/aztec-protocol/why-hashes-dominate-in-snarks-b20a555f074c>
- [11] Raw Benchmarks:
https://drive.google.com/file/d/1uXLUabgh18XHdz0FWvaZo7jMRkqfa1qz/view?usp=share_link
- [12] Proof Compression:
<https://medium.com/aztec-protocol/proof-compression-a318f478d575>
- [13] zkSummit: Plookup, Speeding up the PLONK Prover - Zac Williamson & Ariel Gabizon:
https://www.youtube.com/watch?v=Vdlc1CmRYRY&ab_channel=ZeroKnowledge
- [14] Plookup: A simplified polynomial protocol for lookup tables
<https://eprint.iacr.org/2020/315.pdf>
- [15] Caulk: Lookup Arguments in Sublinear Time: <https://eprint.iacr.org/2022/621>
- [16] Cq: Cached quotients for fast lookups:
<https://eprint.iacr.org/2022/1763.pdf>
- [17] StarkWare Sessions 23 | Recent Progress on Lookup Protocols | Ariel Gabizon
<https://www.youtube.com/watch?v=bPXPIGBo0G0>