Benchmarking the Plonk, TurboPlonk, and UltraPlonk Proving Systems

Lehigh University

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

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 from Aztec's Barretenberg cryptographic library and backend using CPUs [3].

2 Contribution

We extend Aztec's benchmarks [4] for larger circuit sizes and a greater number of hash computations. This demands a performant machine with access to a large number of processor cores and memory. The circuits in our experiments are limited to a maximum constraint size of 2^{26}, and the maximum number of hashes are 128K. 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 **1. Task Size**, defined by the number of hash computations and **2. Problem Size**, defined by the circuit size. These techniques are

used to measure the execution time and memory consumption for these proving systems. The results show that execution time and memory requirements grow log-linearly, **O(n log n)** complexity, with respect to the number of constraints in the circuit. This is consistent with the asymptotic growth rates described in literature [6].

3 Background

The following section provides background into 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 in terms of a fan-in-2 and fan-out-1 gate structure. This means a logical gate, representing either an addition or multiplication gate, 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 a fixed-base elliptic curve scalar-multiplication, elliptic curve point arithmetic, and bitwise XOR and 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 in later sections.

UltraPlonk extends this construction with precomputed lookup tables, which represent efficient key-value mappings [8]**.** This enables a prover to prove that a witness is in a table instead of proving the computation itself. This results in a reduction in the circuit size. The protocol preprocesses a table $\mathbf{T = \{0, \ldots, 2^n - 1\}}$, and devises a way to check that $x \in T$. Lookup tables in this context are commonly used to avoid expensive

bit-decompositions for bitwise operations. For example, rather than computing an XOR operation bit by bit, you can instead encode the 8-bit result in the table and perform 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 in the context of polynomial commitment schemes and providing collision resistance for Merkle-trees. Hashing further dominates **~99%** of the runtime for merkle-tree operations. For example, in UTXO-based chains like Bitcoin, assets are recorded in the form of 'notes' of ownership. These notes of ownership 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. A SHA-256 hash in standard Plonk requires **~27,000** gates, consuming **1.6m** gates [10].

3 Cloud Computing Environment

We measure 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 Base Frequency, 1024 GB DDR4 DRAM, 128 GB SSD.

4 Experimental Results

The raw benchmarks can be found here [11].

We measure 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.

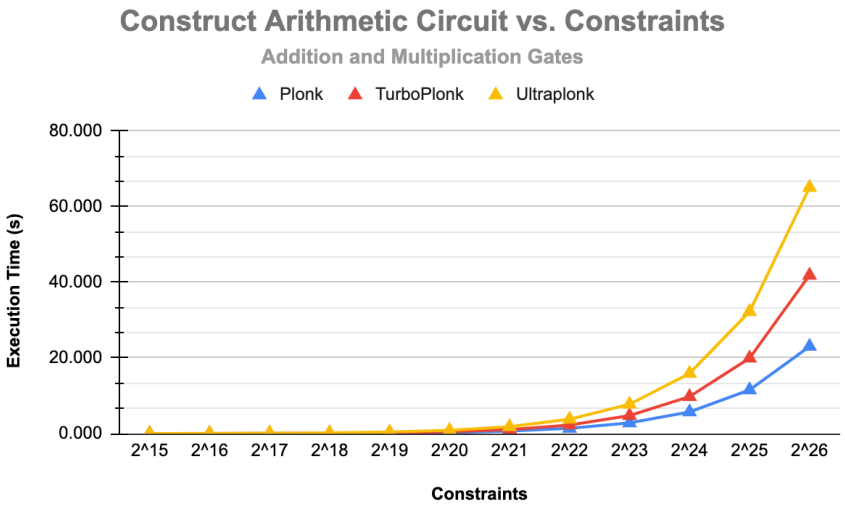
The workload for the prover is divided into multiple tasks:

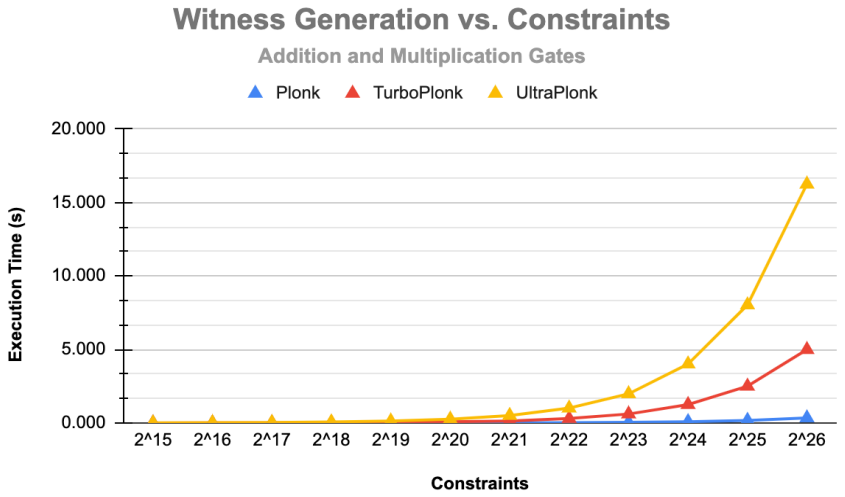
1. Construct the arithmetic circuit
2. Calculate witness polynomials
3. Compute the proving key (including q_l, q_r, etc. and sigma polynomials)
4. Compute the verifier key
5. Generate proof
6. Verify proof

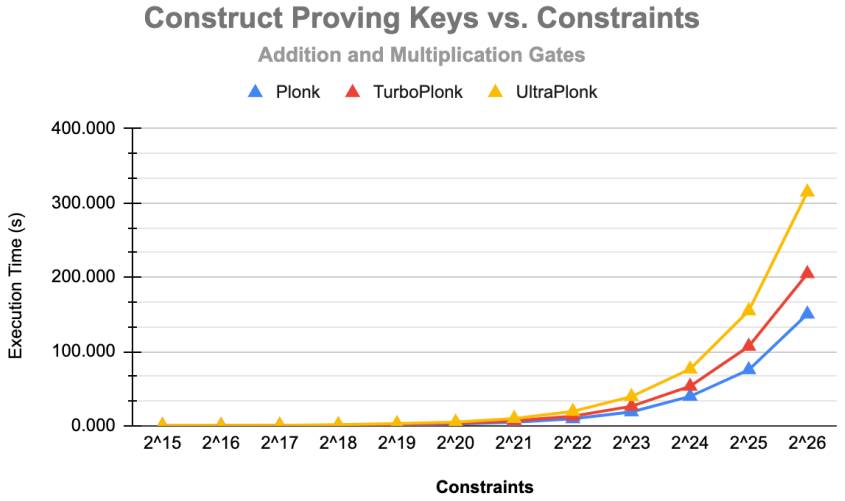
5.1 Addition and Multiplication Gates

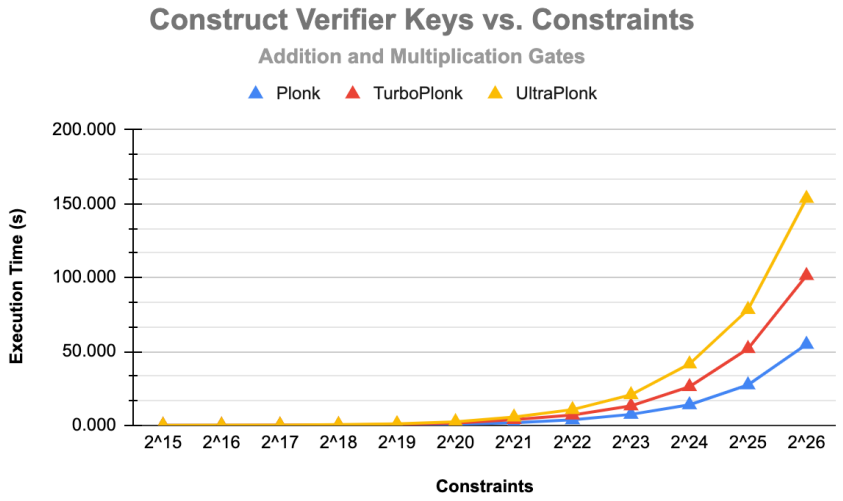
The following charts highlight the performance of Plonk, TurboPlonk, and UltraPlonk using only addition and multiplication gates. TurboPlonk and UltraPlonk exhibit worse performance here because they are structured to optimize performance in the presence of custom gates, which weren't used. In general, performance and memory are proportional to the number of gates in the arithmetic circuit, and the distribution of gates in our circuit is 75% multiplication gates and 25% addition gates. For simplicity, we assume addition and multiplication gates contribute similarly to the prover cost.

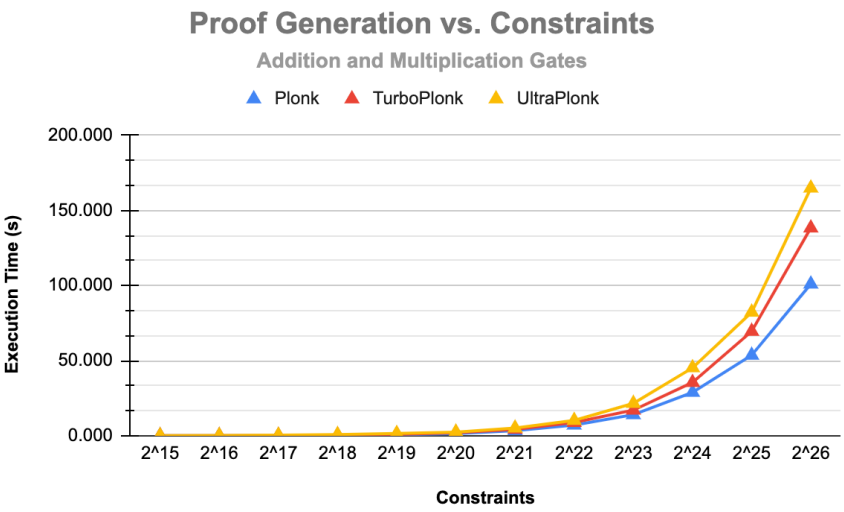
[See Charts in the Next Section]

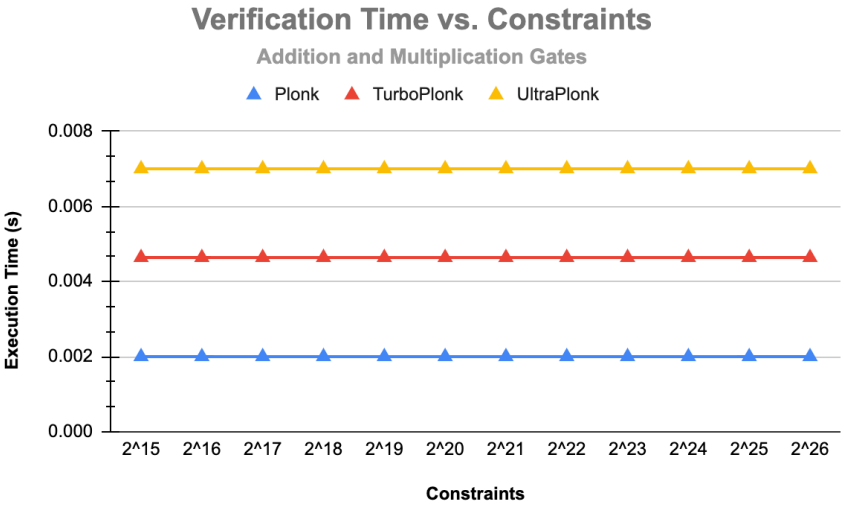

Construct Arithmetic Circuit vs. Constraints
Addition and Multiplication Gates
Plonk
TurboPlonk
Ultraplonk
80.000
60.000
40.000
20.000
0.000
Execution Time (s)
2^15
2^16
2^17
2^18
2^19
2^20
2^21
2^22
2^23
2^24
2^25
2^26
Constraints

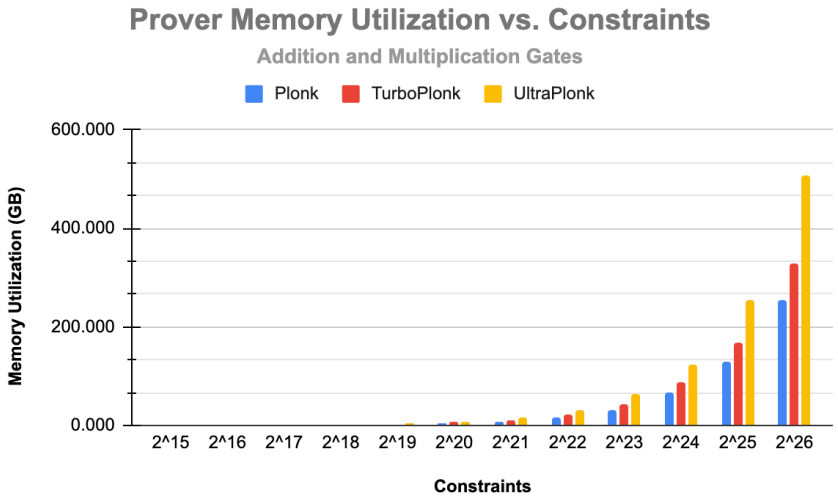

Witness Generation vs. Constraints
Addition and Multiplication Gates
Plonk
TurboPlonk
UltraPlonk
20.000
15.000
10.000
5.000
0.000
Execution Time (s)
2^15
2^16
2^17
2^18
2^19
2^20
2^21
2^22
2^23
2^24
2^25
2^26
Constraints


Construct Proving Keys vs. Constraints
Addition and Multiplication Gates
Plonk
TurboPlonk
UltraPlonk
400.000
300.000
200.000
100.000
0.000
Execution Time (s)
2^15
2^16
2^17
2^18
2^19
2^20
2^21
2^22
2^23
2^24
2^25
2^26
Constraints


Construct Verifier Keys vs. Constraints
Addition and Multiplication Gates
Plonk
TurboPlonk
UltraPlonk
200.000
150.000
100.000
50.000
0.000
Execution Time (s)
2^15
2^16
2^17
2^18
2^19
2^20
2^21
2^22
2^23
2^24
2^25
2^26
Constraints


Proof Generation vs. Constraints
Addition and Multiplication Gates
Plonk
TurboPlonk
UltraPlonk
200.000
150.000
100.000
50.000
0.000
Execution Time (s)
2^15
2^16
2^17
2^18
2^19
2^20
2^21
2^22
2^23
2^24
2^25
2^26
Constraints


Verification Time vs. Constraints
Addition and Multiplication Gates
Plonk
TurboPlonk
UltraPlonk
0.008
0.006
0.004
0.002
0.000
Execution Time (s)
2^15
2^16
2^17
2^18
2^19
2^20
2^21
2^22
2^23
2^24
2^25
2^26
Constraints



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

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

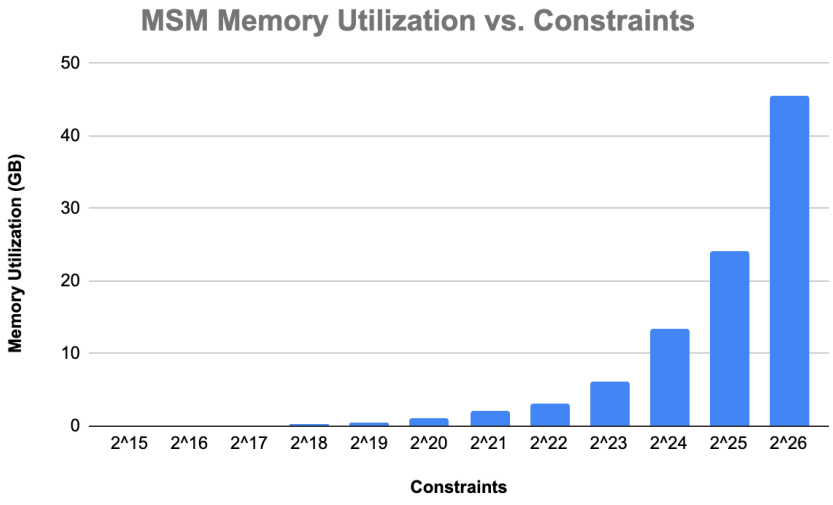
TurboPlonk and UltraPlonk further display larger memory consumption footprints than Plonk. For 2^{26} constraints, generating a TurboPlonk proof consumed **29.5%** more memory **(330 GB)** and UltraPlonk consumed **~99%** more memory **(506 GB)** with respect to generating a Plonk proof **(255 GB)**.

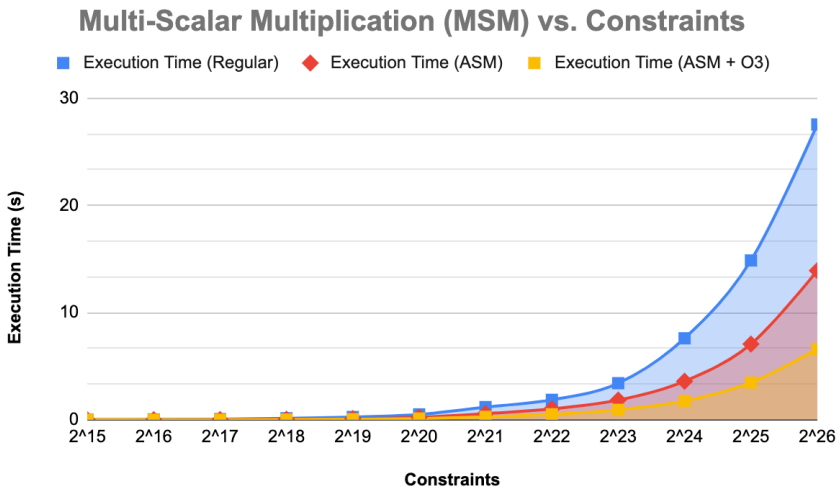
In this setting with only addition and multiplication gates, Plonk is more efficient because it requires fewer group exponentiations. Consequently, less prover work translates to reduced proof generation and memory consumption. Plonk requires **9N** scalar multiplications and a proof size of **~9 G1** 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]. '*N*' represents the number of multiplication gates in the circuit. Group exponentiations dominate approximately **70–80%** of the prover runtime.

Furthermore in the chart below, a single multi-scalar multiplication for 2^{26} constraints executes in **~6.7** seconds and consumes **45.5 GB** of memory. 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**

It's 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 the multi-scalar multiplication. This comes at an additional runtime computational cost resulting from not pre-computing and storing the 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 computation 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 and faster access times in memory, 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 y = g(x) mapping. If these operations are

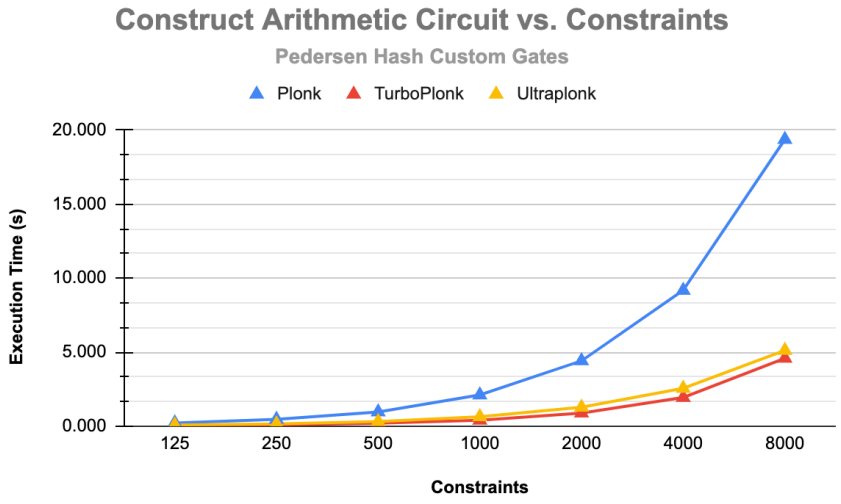
sparsely used in large circuits, then TurboPlonk and UltraPlonk will be slower compared to Plonk. This is supported by the results in the previous section.

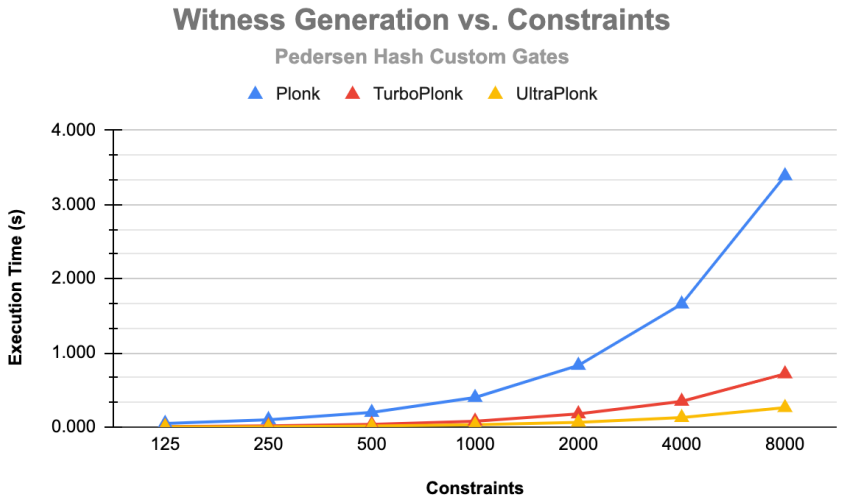
5.2.1 Comparison of Pedersen Hashes

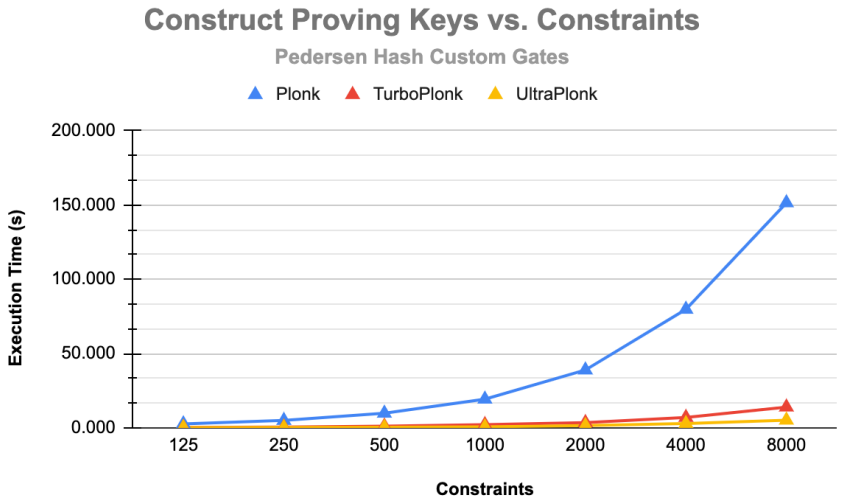
We compare the execution time and memory usage for **125 - 8K** Pedersen hashes. Pedersen hashes are computationally expensive operations in terms of prover workload, and are designed to stress the system. Past **8K** hashes, the performance of Plonk dramatically degrades. Computing arithmetic hashes 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**.

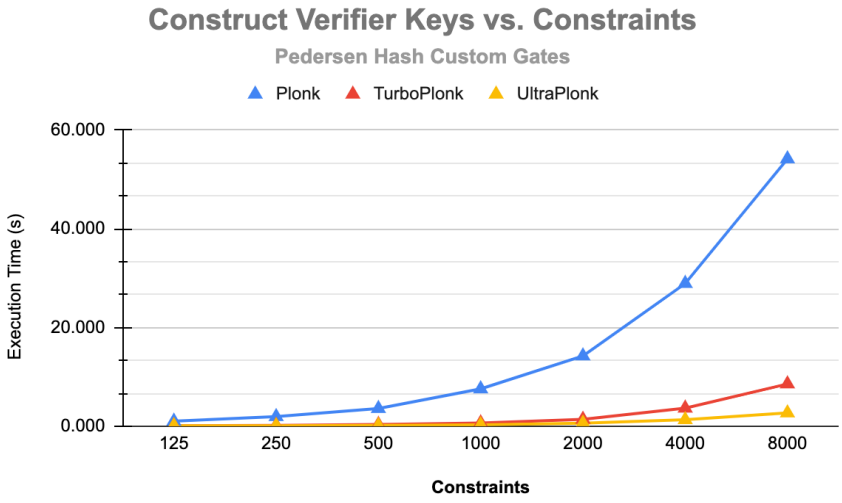
It's worth noting that without assembly instructions and compiler optimizations enabled, these workloads wouldn't be practical and the proof generation would generally be **~4x slower.** The results show that execution time and memory consumption grow log-linearly with respect to the number of constraints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

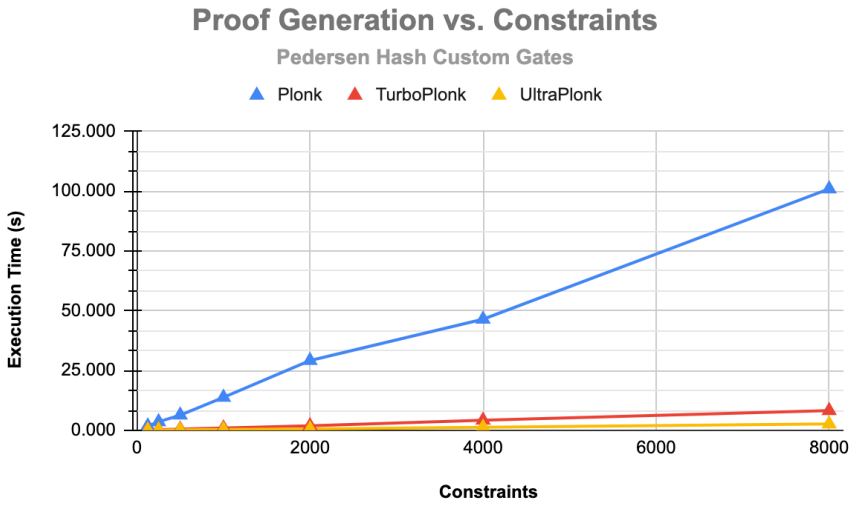
Figures 10 - 16: Prover workloads for custom gates based on pedersen hashes

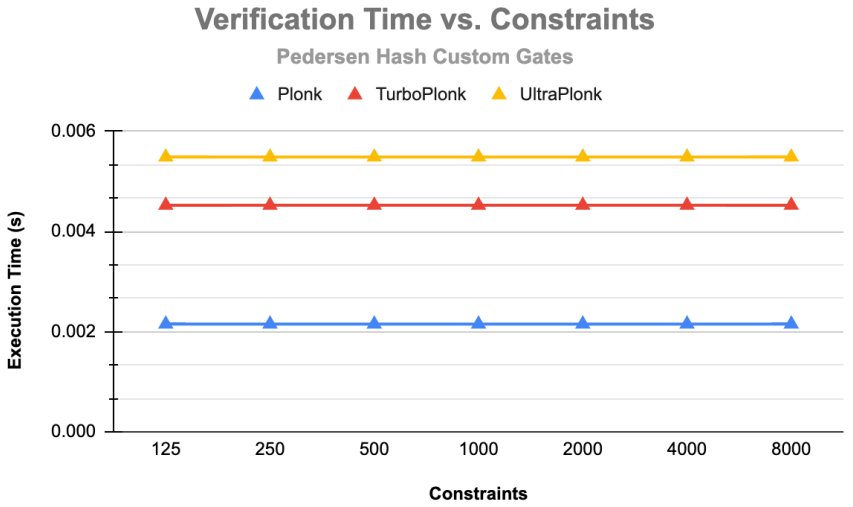


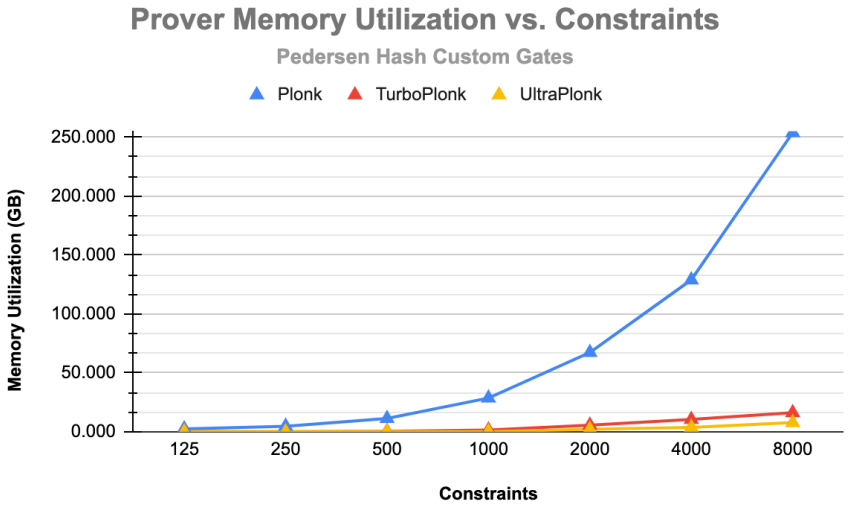

Witness Generation vs. Constraints
Pedersen Hash Custom Gates
Plonk
TurboPlonk
UltraPlonk
4.000
3.000
2.000
1.000
0.000
Execution Time (s)
125
250
500
1000
2000
4000
8000
Constraints


Construct Proving Keys vs. Constraints
Pedersen Hash Custom Gates
Plonk
TurboPlonk
UltraPlonk
200.000
150.000
100.000
50.000
0.000
Execution Time (s)
125
250
500
1000
2000
4000
8000
Constraints


Construct Verifier Keys vs. Constraints
Pedersen Hash Custom Gates
Plonk
TurboPlonk
UltraPlonk
60.000
40.000
20.000
0.000
Execution Time (s)
125
250
500
1000
2000
4000
8000
Constraints


Proof Generation vs. Constraints
Pedersen Hash Custom Gates
Plonk
TurboPlonk
UltraPlonk
125.000
100.000
75.000
50.000
25.000
0.000
Execution Time (s)
0
2000
4000
6000
8000
Constraints


Verification Time vs. Constraints
Pedersen Hash Custom Gates
Plonk
TurboPlonk
UltraPlonk
0.006
0.004
0.002
0.000
Execution Time (s)
125
250
500
1000
2000
4000
8000
Constraints


Prover Memory Utilization vs. Constraints
Pedersen Hash Custom Gates
Plonk
TurboPlonk
UltraPlonk
250.000
200.000
150.000
100.000
50.000
0.000
Memory Utilization (GB)
125
250
500
1000
2000
4000
8000
Constraints

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

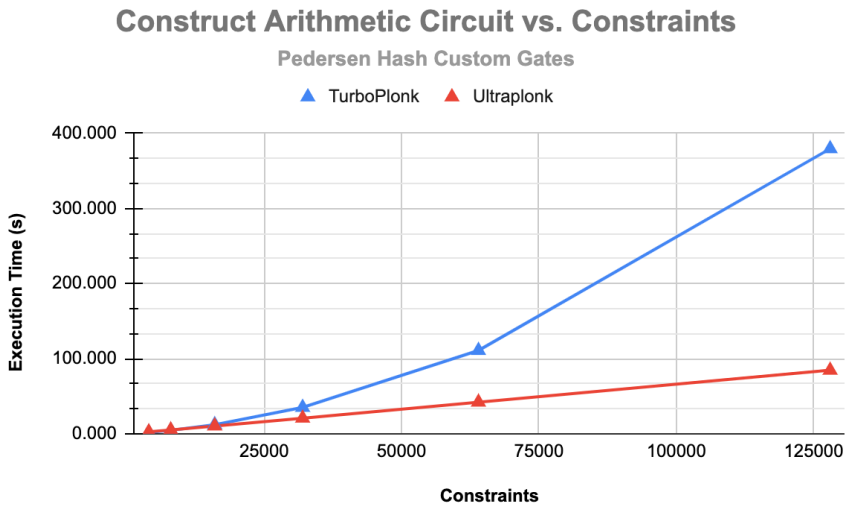
Plonk further displays a larger memory consumption. Generating a TurboPlonk proof consumed **~16x** less memory **(~16 GB)** and UltraPlonk consumed **~32x** less memory **(~8 GB)** compared to generating a standard Plonk proof **(254 GB)**.

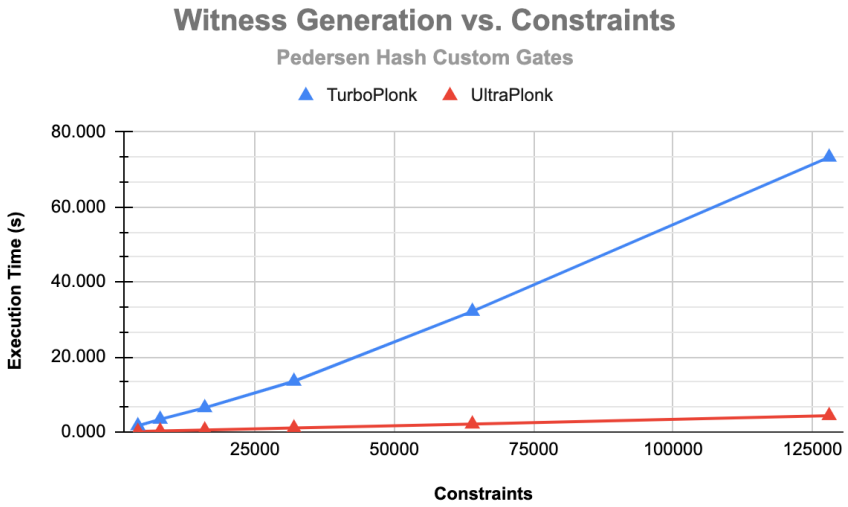
The memory of Plonk is greater than TurboPlonk, as expected. But the memory of TurboPlonk is greater compared to UltraPlonk, which is contrary to our initial expectations. This is due to the fact that we are evaluating the proof systems based on the same task size, i.e. **8K** hashes, rather than the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

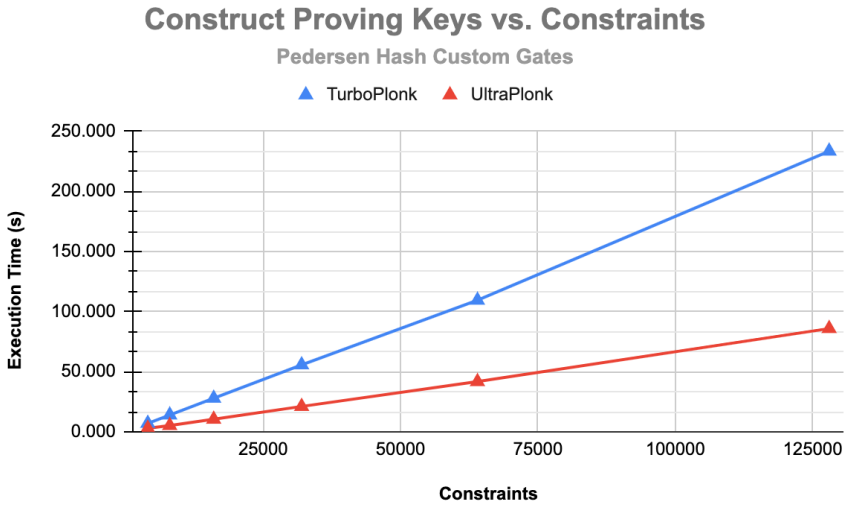
5.2.2 Comparison of Pedersen Hashes

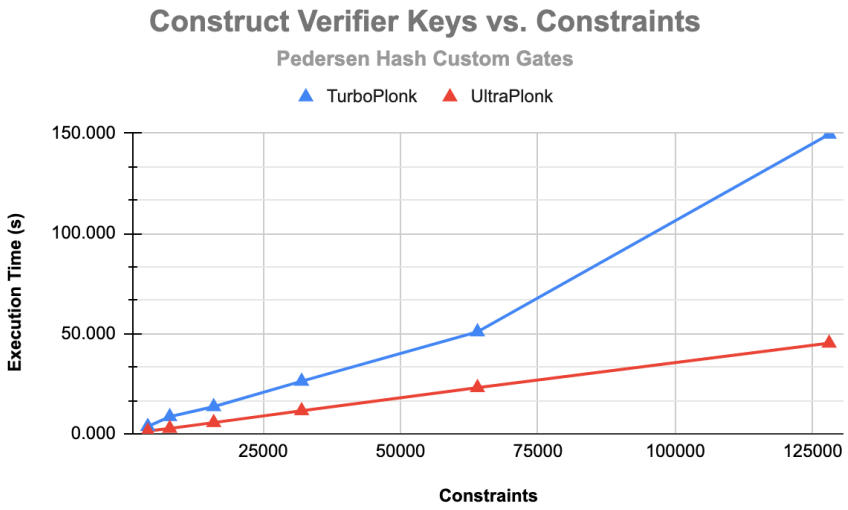
This extends section **4.2.1**, comparing TurboPlonk and UltraPlonk for **4k - 128k** Pedersen hashes.

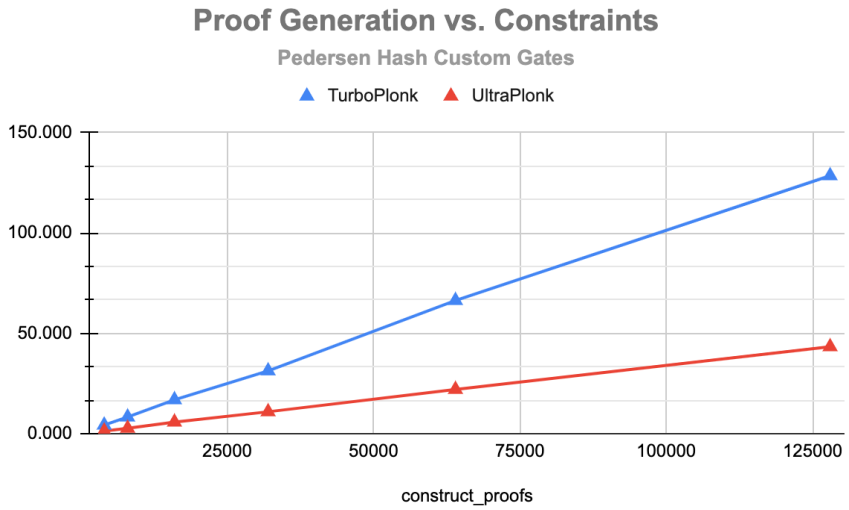
Figures 17 - 23: Prover workloads for custom gates based on Pedersen hashes

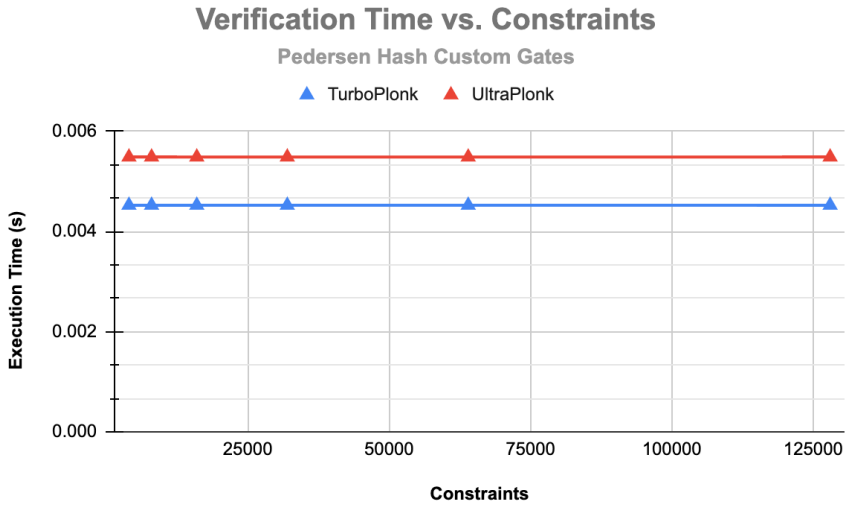


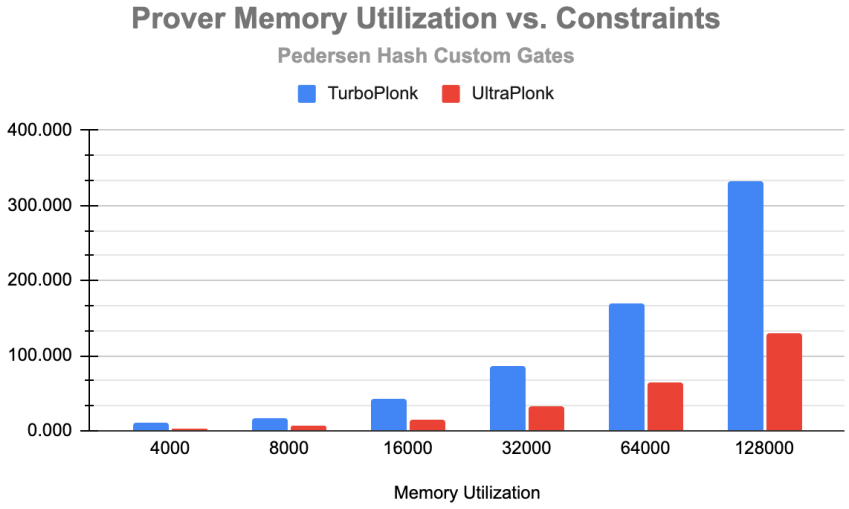

Witness Generation vs. Constraints
Pedersen Hash Custom Gates
TurboPlonk
UltraPlonk
80.000
60.000
40.000
20.000
0.000
Execution Time (s)
25000
50000
75000
100000
125000
Constraints


Construct Proving Keys vs. Constraints
Pedersen Hash Custom Gates
TurboPlonk
UltraPlonk
250.000
200.000
150.000
100.000
50.000
0.000
Execution Time (s)
25000
50000
75000
100000
125000
Constraints


Construct Verifier Keys vs. Constraints
Pedersen Hash Custom Gates
TurboPlonk
UltraPlonk
150.000
100.000
50.000
0.000
Execution Time (s)
25000
50000
75000
100000
125000
Constraints


Proof Generation vs. Constraints
Pedersen Hash Custom Gates
TurboPlonk
UltraPlonk
150.000
100.000
50.000
0.000
25000
50000
75000
100000
125000
construct_proofs


Verification Time vs. Constraints
Pedersen Hash Custom Gates
TurboPlonk
UltraPlonk
0.006
0.004
0.002
0.000
Execution Time (s)
25000
50000
75000
100000
125000
Constraints


Prover Memory Utilization vs. Constraints
Pedersen Hash Custom Gates
TurboPlonk
UltraPlonk
400.000
300.000
200.000
100.000
0.000
4000
8000
16000
32000
64000
128000
Memory Utilization

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

In the figures above, generating a proof for **128K** hashes requires radically different circuit sizes between TurboPlonk and UltraPlonk. For instance, TurboPlonk requires n = 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 in terms of the number of gates, execution time and memory.

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,** where 'n' is the number of constraints and 'p' is the size of the polynomial. We can decompose the total memory by recognizing that each polynomial costs **6 * n**, and 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 linearisation 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 for evaluating 128K hashes. The ratio of the memory consumption of TurboPlonk to UltraPlonk is (**15 * 6 * n) / (19 * 6 * n**) = 3,976,573,680 / 1,503,798,054 = **2.64x**. This estimation is consistent with the **2.5x** increase in memory consumption from the charts.

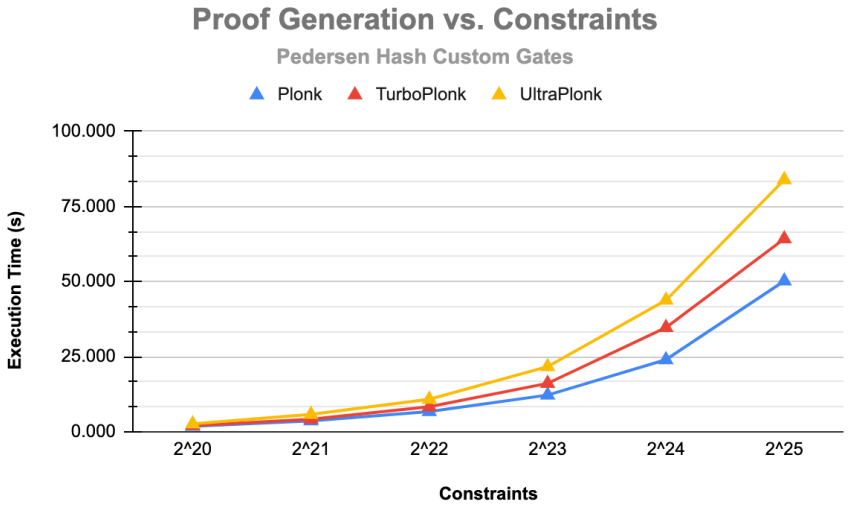
In general, custom gates increase the degree of the identity to be proven, which increases the computational work for the prover. This is in tandem with the reduction in circuit size. For example, the degree of the identity may double, leading to increased computational work for the prover, while the circuit size may be cut by 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, this 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.

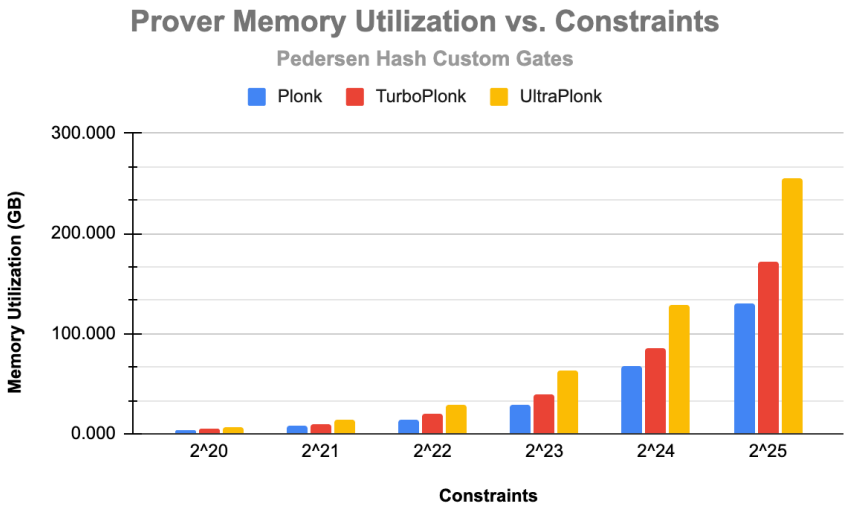
In **section 4.2.3**, the memory consumption for UltraPlonk will be higher than TurboPlonk because we are evaluating the proof systems based on the same problem size, i.e. 2^{25} constraints, rather than on the same task size, i.e. **8K** or **128K** hashes, as in sections **5.2.1 and 5.2.2** respectively.

5.2.3 Comparison of Circuit Size

In this section, we compare the proving systems based on the circuit size rather than 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 examining the throughput based on the circuit size 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. TurbPlonk 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.

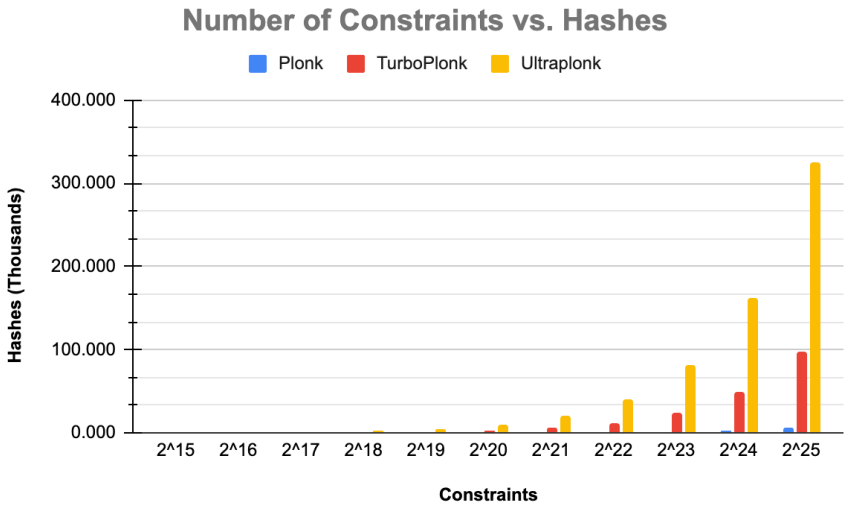


Figure 26: Comparing number of hashes and circuit size

In this setting, UltraPlonk also has the highest memory profile because it has more selector polynomials than TurboPlonk, and each selector polynomial is stored in 3 forms. The increase in memory consumption is not due to the precomputed lookup table because the table sizes don't exceed 2^{16} constraints in the current configuration.

Exceptionally large lookup tables, used primarily for performing efficient range queries for checking x is in the range $0 \leq x \leq 2^n$, can have negative effects. UltraPlonk adds 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 in the circuit [13]. This increases the circuit size, requiring more prover computation. This will also increase the memory consumption 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 doesn't 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 **1. Task Size**, defined by the number of hash computations and **2. Problem Size**, defined by 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 on a 32-core intel CPU. Our next objective is executing components of the prover on a single Nvidia GPU using Cuda, 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 the 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 for his peer review.

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