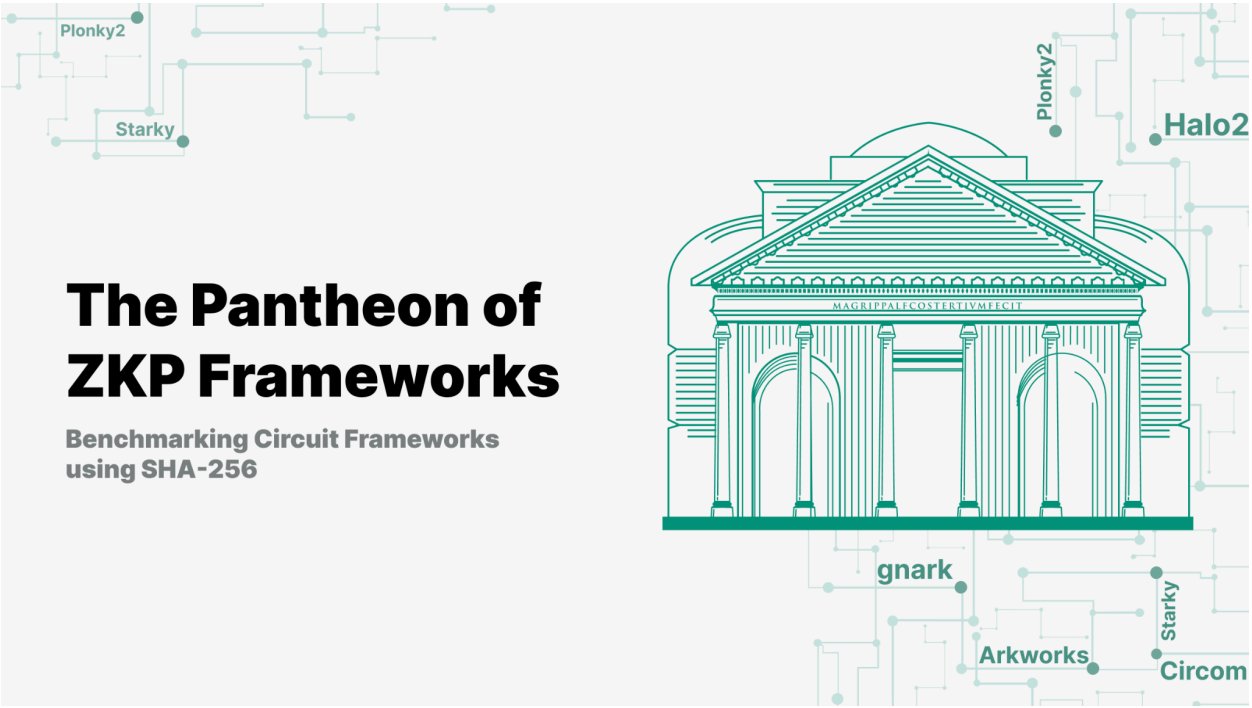
Pantheon of Zero Knowledge Proof Development Frameworks
万神殿 Pantheon：零知识证明开发框架评测平台

Benchmarking Circuit Frameworks using SHA-256

使用 SHA-256 对电路框架进行对比测试



We'd like to thank the teams from Polygon Zero, gnark project at Consensys, Pado Labs, and Delphinus Lab for their valuable review and feedback on this blog.

我们要感谢 *Polygon Zero*、*Consensys gnark*、*Pado Labs* 和 *Delphinus Lab* 团队对本篇文章的宝贵建议和反馈。

The Pantheon of Zero Knowledge Proof

零知识证明开发框架评测平台「万神殿 Pantheon」

Over the past several months, we've dedicated a significant amount of time and effort into developing cutting-edge infrastructure that leverages zk-SNARK succinct proofs. This innovative next-generation platform enables developers to create a new paradigm of blockchain applications that were previously impossible to build.
过去几个月，我们投入了大量时间和精力，开发了利用 zk-SNARK 简洁证明构建的前沿基础设施。这个下一代创新平台使开发者能够构建前所未有的区块链应用新范例。

As part of our development efforts, we have tested and used a wide variety of Zero-Knowledge-Proof (ZKP) development frameworks. While this journey has been rewarding, we do realize that the abundance of available ZKP frameworks often creates a challenge for new developers who are trying to find the best fit for their specific use cases and performance requirements. With this pain point in mind, we believe a community evaluation platform that is able to provide comprehensive benchmark results is needed and will greatly aid in the development of these new applications.
在开发工作中，我们测试并使用了多种零知识证明 (ZKP) 开发框架。虽然这段旅程收获颇丰，但我们也确实意识到，当新的开发者试图找到最适合其特定用例和性能要求的框架时，多种多样的 ZKP 框架通常会给他们带来挑战。 考虑到这一痛点，我们认为需要一个能够提供全面性能测试结果的社区评估平台，这将极大地促进这些新应用的开发。

To fulfill this need, we are launching the **Pantheon of Zero Knowledge Proof** as a public good community initiative. The first step will be to encourage the community to share **reproducible benchmarking** results from various ZKP frameworks. Our ultimate goal is to collectively and collaboratively create and maintain a **universally recognized evaluation testbed** that covers low-level circuit development frameworks, high-level zkVMs and compilers, and even hardware acceleration providers. We hope that this initiative will expedite the adoption of ZKPs by facilitating informed decision-making, while also encouraging the evolution and iteration of the ZKP frameworks themselves by providing a set of commonly referenceable benchmarking results. We are committed to investing in this initiative and invite all like-minded community members to join us and contribute to this effort together!
为了满足这一需求，我们推出了零知识证明开发框架评测平台「万神殿 **Pantheon**」这一公益社区倡议。 倡议的第一步将鼓励社区分享各种 ZKP 框架的可复现性能测试结果。我们的最终目标是共同协作创建并维护一个广受认可的测试平台，评估低级电路开发框架、高级 zkVM 和编译器，甚至硬件加速提供商。我们希望这一举措能够让开发者们在选用框架时能有更多性能比较的参考，从而加快 ZKP 的推广。同时，我们希望通过提供一组普遍可参考的性能测试结果，促进 ZKP

框架本身的升级和迭代。我们将大力投入这项计划，并邀请所有志同道合的社区成员加入我们，共同为这项工作做出贡献！

A First Step: Benchmarking Circuit Frameworks using SHA-256

第一步：使用 SHA-256 对电路框架进行性能测试

In this blog post, we take the first step towards building the Pantheon of ZKP by providing a reproducible set of benchmark results using SHA-256 across a range of low-level circuit development frameworks. While we acknowledge that other benchmarking granularities and primitives possible, we selected SHA-256 due to its applicability to a wide range of ZKP use cases, including blockchain systems, digital signatures, zkDID and more. It's also worth mentioning that we also leverage SHA-256 in our own system, so it is quite convenient for us as well! 😂
在这篇文章中，我们迈出了构建 ZKP Pantheon 的第一步，在一系列低级电路开发框架中使用 SHA-256 提供一组可复现的性能测试结果。虽然我们承认其他性能测试粒度和原语或许也是可行的，但我们选择 SHA-256 是因为它适用于广泛的 ZKP 用例，包括区块链系统、数字签名、zkDID 等。另外值得一提的是，我们在自己的系统中也使用了 SHA-256，所以这对我们来说也很方便！ 😂

Our benchmark evaluates the performance of SHA-256 on various zk-SNARK and zk-STARK circuit development frameworks. Through this comparison, we seek to provide developers with insights into the efficiency and practicality of each framework. Our goal is that these findings will enable developers to make informed decisions when selecting the most suitable framework for their projects.
我们的性能测试评估了 SHA-256 在各种 zk-SNARK 和 zk-STARK 电路开发框架上的性能。通过比较，我们力求为开发者提供关于每个框架的效率和实用性的见解。我们的目标是，希望本次性能测试结果能够为开发者在选择最佳框架时提供参考，使之做出明智的决定。

Proving Systems

证明系统

In recent years, we've observed a proliferation of zero-knowledge proving systems. While it is challenging to keep up with all of the exciting advances in the space, we've carefully selected the following proving systems based on their maturity and developer adoption. Our aim is to present a representative sample of different frontend/backend combinations.

近年来，我们观察到零知识证明系统激增。跟上该领域所有激动人心的进步是具有挑战性的，我们根据成熟度和开发者采用情况精心挑选了以下证明系统作为测试对象。我们的目标是提供不同前端/后端组合的代表性样本。

1. **Circom** + **snarkjs** / **rapidsnark**: Circom is a popular DSL for writing circuits and generating R1CS constraints while snarkjs is able to generate Groth16 or Plonk proof for Circom. Rapidsnark is also a prover for Circom that generates Groth16 proof and is usually much faster than snarkjs due to the use of the ADX extension, which parallelizes proof generation as much as possible.
2. **gnark**: gnark is a comprehensive Golang framework from Consensys that supports Groth16, Plonk, and many more advanced features.
3. **Arkworks**: Arkworks is a comprehensive Rust framework for zk-SNARKs.
4. **Halo2 (KZG)**: Halo2 is Zcash's zk-SNARK implementation with Plonk. It is equipped with the highly-flexible Plonkish arithmetization that supports many useful primitives, such as custom gates and lookup tables. We use a Halo2 fork with KZG support from the Ethereum Foundation and Scroll.
5. **Plonky2**: Plonky2 is a SNARK implementation based on techniques from PLONK and FRI from Polygon Zero. Plonky2 uses a small Goldilocks field and supports efficient recursion. In our benchmarking, we targeted 100-bit conjectured security and used the parameters that yielded the best proving time for the benchmark job. Specifically, we used 28 Merkle queries, a blowup factor of 8, and a 16-bit proof-of-work grinding challenge. Moreover, we set num_of_wires = 60 and num_routed_wires = 60.
6. **Starky**: Starky is a highly performant STARK framework from Polygon Zero. In our benchmarking, we targeted 100-bit conjectured security and used the parameters that yielded the best proving time. Specifically, we used 90 Merkle queries, a blowup factor of 2, and a 10-bit proof-of-work grinding challenge.

1. **Circom** + **snarkjs** / **rapidsnark**: Circom 是一种流行的 DSL，用于编写电路和生成 R1CS 约束，而 snarkjs 能够为 Circom 生成 Groth16 或 Plonk 证明。Rapidsnark 也是 Circom 的证明器，它生成 Groth16 证明，并且由于使用了 ADX 扩展，它通常比 snarkjs 快得多，并尽可能并行化证明生成。
2. **gnark**: gnark 是来自 Consensys 的综合 Golang 框架，支持 Groth16、Plonk 和许多更高级的功能。
3. **Arkworks**: Arkworks 是一个用于 zk-SNARKs 的综合 Rust 框架。
4. **Halo2 (KZG)**: Halo2 是 Zcash 与 Plonk 的 zk-SNARK 实现。 它配备了高度灵活的 Plonkish 算术，支持许多有用的原语，例如自定义网关和查找表。我们使用具有以太坊基金会和 Scroll 支持的 KZG 的 Halo2 分叉。
5. **Plonky2**: Plonky2 是基于来自 Polygon Zero 的 PLONK 和 FRI 技术的 SNARK 实现。 Plonky2 使用小的 Goldilocks 字段并支持高效的递归。 在我们的性能测试中，我们以 100 位推测的安全性为目标，并使用为性能测试工作产生最佳证明时间的参数。具体来说，我们使用了 28 Merkle 查询、8 的放大系数和 16 位工作量证明挑战。 此外，我们设置 num_of_wires = 60 和 num_routed_wires = 60。

6. **Starky**: Starky 是 Polygon Zero 的高性能 STARK 框架。在我们的性能测试中，我们以 100 位推测的安全性为目标，并使用产生最佳证明时间的参数。具体来说，我们使用了 90 Merkle 查询、2 倍放大系数和 10 位工作量证明挑战。

The table below summarizes the above frameworks with the relevant configurations used in our benchmarking. This list is by no means exhaustive and many state-of-the-art frameworks/techniques (e.g., Nova, GKR, Hyperplonk) are left for future work.
下表总结了上述框架以及我们性能测试中使用的相关配置。这个列表绝不是详尽的，我们还将在未来研究许多最先进的框架/技术（例如，Nova、GKR、Hyperplonk）。

Please be aware that these benchmark results are only for circuit development frameworks. We plan to publish a separate blog benchmarking different zkVMs (e.g., Scroll, Polygon zkEVM, Consensys zkEVM, zkSync, Risc Zero, zkWasm) and IR compiler frameworks (e.g., Noir, zkLLVM) in the future.
请注意，这些性能测试结果仅适用于电路开发框架。我们计划在未来发布一篇单独的文章，对不同的 zkVM（例如，Scroll、Polygon zkEVM、Consensys zkEVM、zkSync、Risc Zero、zkWasm）和 IR 编译器框架（例如，Noir、zkLLVM）进行性能测试。

Framework	Arithmetization	Algorithm	Field	Other Configs
Circom + snarkjs / rapidsnark	R1CS	Groth16	BN254 scalar	
gnark	R1CS	Groth16	BN254 scalar	
Arkworks	R1CS	Groth16	BN254 scalar	
Halo2 (KZG)	Plonkish	KZG	BN254 scalar	
Plonky2	Plonk	FRI	Goldilocks	blowup factor = 8 proof of work bits = 16 query rounds = 28 num_of_wires = 60 num_routed_wires = 60
Starky	AIR	FRI	Goldilocks	blowup factor = 2 proof of work bits = 10 query rounds = 90

框架	算术化	算法	数域	其他配置
Circom + snarkjs / rapidsnark	R1CS	Groth16	BN254 scalar	
gnark	R1CS	Groth16	BN254 scalar	
Arkworks	R1CS	Groth16	BN254 scalar	
Halo2 (KZG)	Plonkish	KZG	BN254 scalar	
Plonky2	Plonk	FRI	Goldilocks	blowup factor = 8 proof of work bits = 16 query rounds = 28 num_of_wires = 60 num_routed_wires = 60
Starky	AIR	FRI	Goldilocks	blowup factor = 2 proof of work bits = 10 query rounds = 90

Benchmark Methodology

性能评测方法论

To benchmark these various proving systems, we computed the SHA-256 hash for N bytes of data, where we experimented with N = 64, 128, ..., 64K (with one exception being Starky, where the circuit repeats the SHA-256 computation for a fixed 64-byte input but maintains the same total number of message chunks). The benchmark code and SHA-256 circuit implementations can be found in this repository.
为了对这些不同的证明系统进行性能测试，我们计算了 N 字节数据的 SHA-256 哈希值，其中我们对 N = 64、128、...、64K 进行了实验（Starky 是一个例外，其中电路重复 SHA-256 固定 64 字节输入的计算，但保持相同的消息块总数）。可以在此存储库中找到性能代码和 SHA-256 电路配置。

Furthermore, we conducted the benchmarking of each system using the following performance metrics:

- Proof Generation Time (including witness generation time)
- Peak Memory usage during proof generation
- Average CPU Utilization % during proof generation. (This metric reflects the degree of parallelization during proof generation)

此外，我们使用以下性能指标对每个系统进行了性能测试：

- 证明生成时间（包括见证生成时间 ）

- 证明生成期间的内存使用峰值
- 证明生成期间的平均 CPU 使用率百分比。（该指标反映了证明生成过程中的并行化程度）

Please note that we are making some “hand-waving” assumptions regarding the proof size and proof verification cost, as these aspects can be mitigated by composing with Groth16/KZG before going on-chain.
请注意，我们正在对证明大小和证明验证成本做一些“随意”的假设，因为这些方面可以通过在上链之前与 Groth16 或 KZG 组合来减轻。

The Machines

机器

We conducted our benchmarking on two different machines:
- **Linux Server**: 20 Cores @2.3 GHz, 384GB memory
- **Macbook M1 Pro**: 10 Cores @3.2Ghz, 16GB memory

我们在两台不同的机器上进行了性能测试：
- Linux 服务器：20 核 @2.3 GHz，384GB 内存
- Macbook M1 Pro：10 核 @3.2Ghz，16GB 内存

The Linux server was used to simulate the scenario with many CPU cores and abundant memory. While the Macbook M1 Pro, which is commonly used for R&D, has a more powerful CPU with fewer cores.
Linux 服务器用于模拟 CPU 核数多、内存充裕的场景。而通常用于研发的 Macbook M1 Pro 拥有更强大的 CPU，但内核较少。

We enabled multithreading where optional, but we did not utilize GPU acceleration in this benchmark. We plan to include GPU benchmarking as part of our future work.
我们启用了可选的多线程，但我们没有在此性能测试中使用 GPU 加速。我们计划在未来进行 GPU 性能测试。

Benchmark Results

性能评测结果

Number of Constraints

约束数量

Before we move on to the detailed benchmarking results, it's useful to first understand the complexity of the SHA-256 by looking at the number of constraints in each proving system. It is important to be aware that the constraint numbers in different arithmetization schemes are not directly comparable.
在我们继续讨论详细的性能测试结果之前，首先通过查看每个证明系统中的约束数量来了解 SHA-256 的复杂性是很有用的。重要的是要注意不能直接比较不同算术方案中的约束数量。

The results below correspond to a pre-image size of 64KB. While results may vary with other pre-image sizes, they can be roughly scaled linearly.
下面的结果对应 64KB 的原像尺寸。虽然结果可能因其他原像尺寸而异，但它们可以粗略地线性缩放。

- Circom, gnark, and Arkworks all use the same R1CS arithmetization, and the number of R1CS constraints for computing 64KB SHA-256 is roughly 30M to 45M. The difference among Circom, gnark, and Arkworks is likely due to implementation differences.
- Halo2 and Plonky2 both use Plonkish arithmetization, where the number of rows range from 2^22 to 2^23. Halo2's implementation of SHA-256 is much more efficient than that in Plonky2 due to the use of lookup tables.
- Starky uses AIR arithmetization where the execution trace tables require 2^16 transition steps.

- Circom、gnark、Arkworks 都使用相同的 R1CS 算法，计算 64KB SHA-256 的 R1CS 约束数量大致在 30M 到 45M 之间。 Circom、gnark 和 Arkworks 之间的差异可能是由于配置差异造成的。
- Halo2 和 Plonky2 都使用 Plonkish 算术，其中行数范围从 2^22 到 2^23。由于使用查找表，Halo2 的 SHA-256 实现效率比 Plonky2 的高得多。
- Starky 使用 AIR 算法，其中执行跟踪表需要 2^16 个转换步骤。

Proving System	Number of Constraints (64KB SHA-256)
Circom	32M
gnark	45M

Arkworks	43M
Halo2	4M rows (K=22)
Plonky2	8M rows (K=23)
Starky	2^16 transition steps

证明系统	约束数量 **(64KB SHA-256)**
Circom	32M
gnark	45M
Arkworks	43M
Halo2	4M rows (K=22)
Plonky2	8M rows (K=23)
Starky	2^16 transition steps

Proof Generation Time

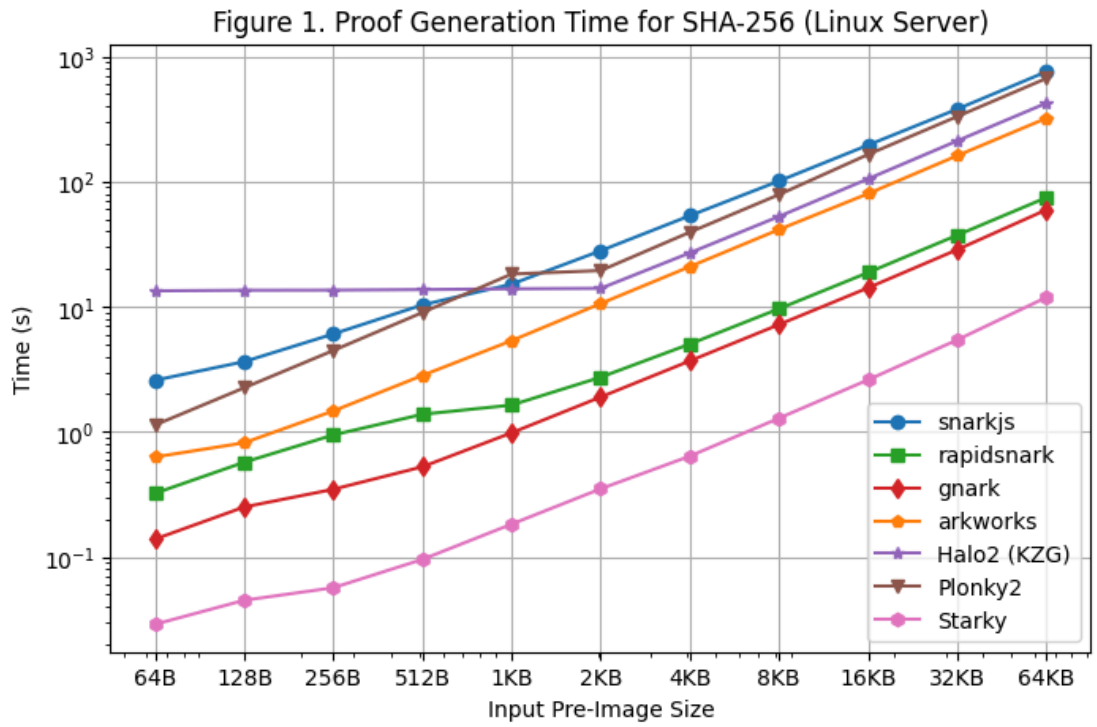
证明生成时间

[Figure 1] illustrates the proof generation time of each framework for SHA-256 over the various pre-image sizes, using the Linux Server. We are able to make the following observations:
[图 1] 使用 Linux 服务器测试了 SHA-256 的每个框架在各种原图像尺寸上的证明生成时间。 我们可以得到以下发现：

- For SHA-256, Groth16 frameworks (rapidsnark, gnark, and Arkworks) generate proofs faster than Plonk frameworks (Halo2 and Plonky2). This is because SHA-256 mainly consists of bitwise operations where wire values are either 0 or 1. For Groth16, this reduces most of the computations from elliptic curve scalar multiplication to elliptic curve point addition. However, wire values are not directly used in Plonk's computations, so the special wire structure in SHA-256 does not reduce the amount of computation needed in Plonk frameworks.
- Among all of the Groth16 frameworks, gnark and rapidsnark are 5x-10x faster than Arkworks and snarkjs. This is thanks to their superior capability of utilizing multiple cores to parallelize proof generation. Gnark is 25% faster than rapidsnark.

- For the Plonk frameworks, Plonky2 is 50% slower than Halo2 for SHA-256 when using a larger pre-image size of >= 4KB. This is because Halo2's implementation heavily utilizes a lookup table to speed up bitwise operations, resulting in 2x fewer rows than Plonky2. However, if we compare Plonky2 and Halo2 with the same number of rows (e.g., SHA-256 over 2KB in Halo2 vs. SHA-256 over 4KB in Plonky2), Plonky2 is 50% faster than Halo2. If we implement SHA-256 with a lookup table in Plonky2, we should expect Plonky2 to be faster than Halo2, although the Plonky2 proof size is larger.
- On the other hand, when the input pre-image size is small (<=512 bytes), Halo2 is slower than Plonky2 (and other frameworks) due to the fixed setup cost of the lookup table that accounts for the majority of the constraints. However, as the pre-image increases, Halo2's performance becomes more competitive, with a proof generation time that remains constant for pre-image sizes up to 2KB and then scales almost linearly, as can be observed in the graph.
- As expected, Starky's proof generation time is significantly shorter (5x-50x) than any SNARK framework, but this comes at the cost of a much larger proof size.
- An additional note is that even if the circuit size is linear in the size of the pre-image, the proof generation grows super-linearly for SNARKs due to the O(nlogn) FFT (although this is not obvious on the graph due to the log scale).

- 对于 SHA-256，Groth16 框架（rapidsnark、gnark 和 Arkworks）生成证明的速度比 Plonk 框架（Halo2 和 Plonky2）快。这是因为 SHA-256 主要由位运算组成，其中线值为 0 或 1。对于 Groth16，这减少了从椭圆曲线标量乘法到椭圆曲线点加法的大部分计算。但是，连线值并不直接用于 Plonk 的计算，因此 SHA-256 中的特殊连线结构不会减少 Plonk 框架中所需的计算量。
- 在所有 Groth16 框架中，gnark 和 rapidsnark 比 Arkworks 和 snarkjs 快 5 到 10 倍。这要归功于它们利用多个内核并行化生成证明的卓越能力。Gnark 比 rapidsnark 快 25%。
- 对于 Plonk 框架，当使用 >= 4KB 的较大原像尺寸时，Plonky2 的 SHA-256 比 Halo2 的慢 50%。这是因为 Halo2 的实现主要使用查找表来加速按位运算，导致行数比 Plonky2 少 2 倍。但是，如果我们比较具有相同行数的 Plonky2 和 Halo2（例如，Halo2 中超过 2KB 的 SHA-256 与 Plonky2 中超过 4KB 的 SHA-256），Plonky2 比 Halo2 快 50%。如果我们在 Plonky2 中使用查找表实现 SHA-256，我们应该期望 Plonky2 比 Halo2 更快，尽管 Plonky2 的证明尺寸更大。
- 另一方面，当输入原像尺寸较小（<=512 字节）时，由于查找表的固定设置成本占大部分约束，Halo2 比 Plonky2（和其他框架）慢。然而，随着原像的增加，Halo2 的性能变得更具竞争力，对于高达 2KB 的原像大小，其证明生成时间保持不变，如图所示，其几乎呈线性扩展。
- 正如预期的那样，Starky 的证明生成时间比任何 SNARK 框架都短得多 (5倍-50倍)，但这是以更大的证明大小为代价的。
- 另外需要注意的是，即使电路大小与原像大小成线性关系，由于 O(nlogn) FFT，对于 SNARKs 的证明生成也是呈超线性增长的（尽管由于对数刻度，这一现象在图表上并不明显）。



We also conducted a proof generation time benchmark on the Macbook M1 Pro, as illustrated in [Figure 2]. However, it is important to note that rapidsnark was not included in this benchmark due to its lack of support for arm64 architecture. In order to use snarkjs on arm64, we had to generate the witness using webassembly, which is slower than the C++ witness generation used on the Linux Server.
我们还在 Macbook M1 Pro 上进行了证明生成时间性能测试，如 [图 2] 所示。但是，需要注意的是，由于缺乏对 arm64 架构的支持，rapidsnark 未包含在该性能测试中。为了在 arm64 上使用 snarkjs，我们必须使用 webassembly 生成见证，这比 Linux 服务器上使用的 C++ 见证生成要慢。

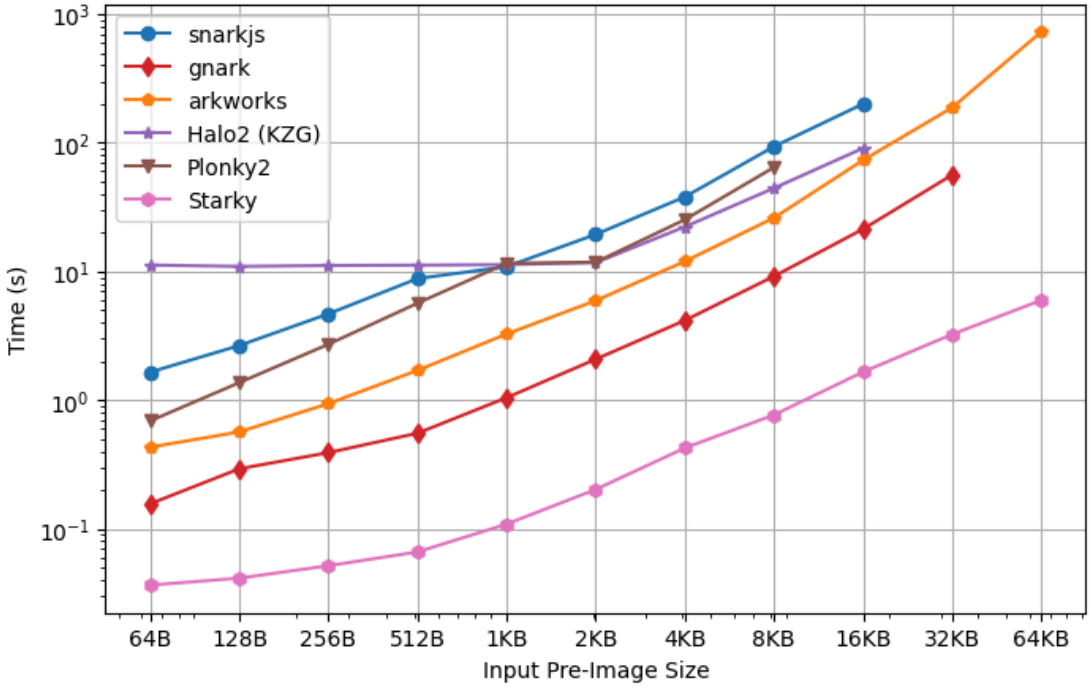
There were several additional observations when running the benchmark on Macbook M1 Pro:
在 Macbook M1 Pro 上运行性能测试时还有几个额外的观察结果：

- Except for Starky, all SNARK frameworks encountered out-of-memory (OOM) errors or used swap memory (resulting in slower proving time) when the pre-image size became large. Specifically, Groth16 frameworks (snarkjs, gnark, Arkworks) began to use swap memory when the pre-image size was greater than or equal to 8KB, and gnark encountered OOM for 64KB. Halo2 encountered a memory limit when the pre-image size was greater than or equal to 32KB. Plonky2 begins to use swap memory when pre-image size was greater than or equal to 8KB.
- FRI-based frameworks (Starky and Plonky2) were about 60% faster on the Macbook M1 Pro than on the Linux Server, while other frameworks saw similar proving times as compared to those on the Linux Server. As a result, Plonky2 achieved almost the same proving time as Halo2 on the Macbook M1 Pro, even though lookup table was not used

in Plonky2. The main reason for this is that the Macbook M1 Pro has a more powerful CPU but with fewer cores. FRI mainly conducts hash operations, which are more sensitive to CPU clock cycles but not as parallelizable as KZG/Groth16.

- 除了 Starky 之外，所有 SNARK 框架在原像尺寸变大时都会遇到内存不足 (OOM) 错误或使用交换内存（导致证明时间变慢）现象。具体来说，Groth16 框架（snarkjs、gnark、Arkworks）在原像尺寸>= 8KB 时就开始使用交换内存，而 gnark 在原像尺寸>= 64KB 时出现内存不足。当原像尺寸>= 32KB 时，Halo2 遇到了内存限制。当原像尺寸>= 8KB 时，Plonky2 开始使用交换内存。
- 基于 FRI 的框架（Starky 和 Plonky2）在 Macbook M1 Pro 上比在 Linux 服务器上快大约 60%，而其他框架在两台机器上面的证明时间相似。因此即使在 Plonky2 中没有使用查找表，它在 Macbook M1 Pro 上实现了与 Halo2 几乎相同的证明时间。 主要原因是 Macbook M1 Pro 拥有更强大的 CPU，但内核更少。FRI 主要进行哈希运算，对 CPU 时钟周期比较敏感，但并行性不如 KZG 或 Groth16。

Figure 2. Proof Generation Time for SHA-256 (MBP M1)



Peak Memory Usage

内存使用峰值

The peak memory usage during proof generation on the Linux Server and Macbook M1 Pro is shown in [Figure 3] and [Figure 4], respectively. The following observations can be made based on these benchmarking results:

[图 3] 和 [图 4] 分别显示了在 Linux Server 和 Macbook M1 Pro 上生成证明期间的内存使用峰值。 根据这些性能测试结果可以得出以下观察结果：

- Among all of the SNARK frameworks, rapidsnark is the most memory efficient. We also see that Halo2 uses more memory when the pre-image size is smaller due to the fixed setup cost of the lookup table, but consumes less memory overall when the pre-image size is larger.
- Starky is more than 10x more memory efficient than SNARK frameworks. This is in part due to its use of fewer rows.
- It should be noted that the peak memory usage remains relatively flat on the Macbook M1 Pro as the pre-image size becomes large due to the usage of swap memory.
- 在所有 SNARK 框架中，rapidsnark 是内存效率最高的。我们还看到，由于查找表的固定设置成本，当原像尺寸较小时，Halo2 使用更多内存，但当原像尺寸较大时，整体消耗的内存较少。
- Starky 的内存效率比 SNARK 框架高 10 倍以上。部分原因是它使用了更少的行。
- 应该注意的是，由于使用交换内存，原像尺寸变大，因此 Macbook M1 Pro 上的内存使用量峰值保持相对平稳。

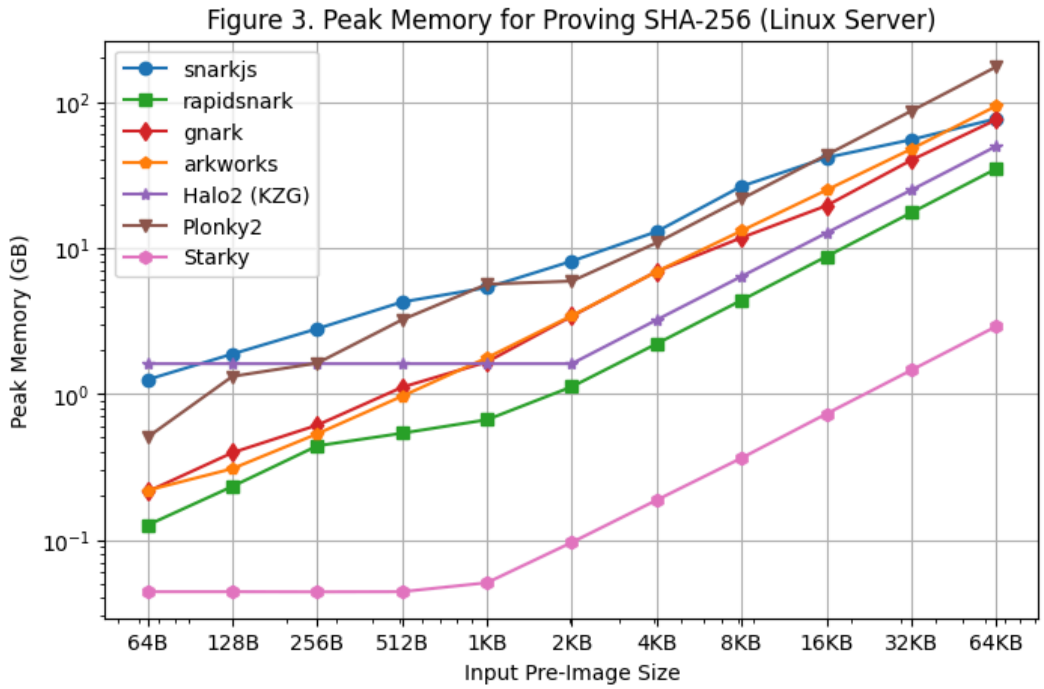
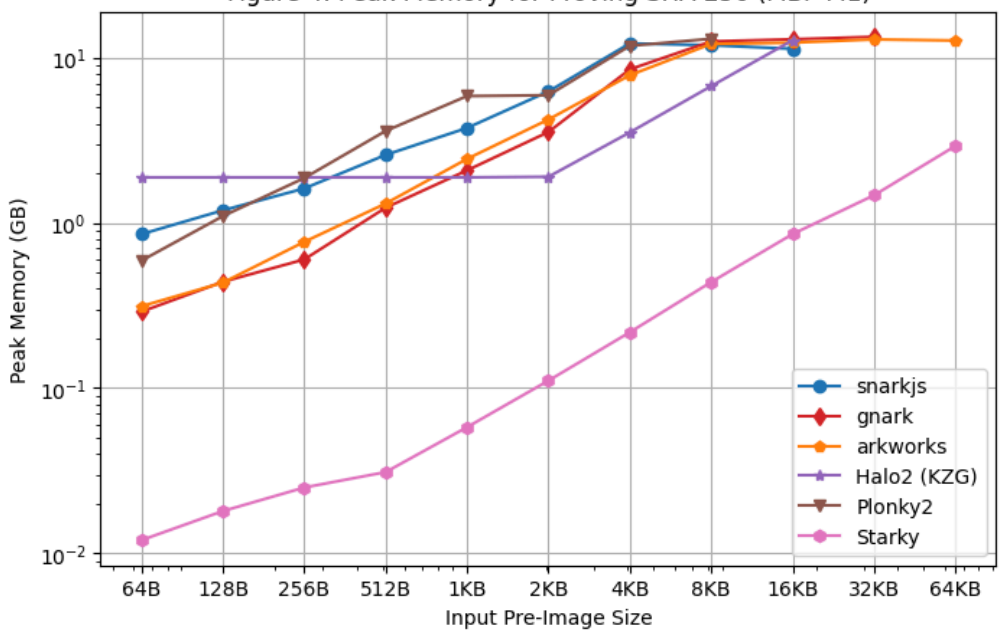


Figure 4. Peak Memory for Proving SHA-256 (MBP M1)



CPU Utilization

CPU 利用率

We evaluated the degree of parallelization for each proving system by measuring the average CPU utilization during proof generation for SHA-256 over a 4KB pre-image input. The table below shows the average CPU utilization (and the average utilization per core in the parentheses) on both the Linux Server (with 20 cores) and the Macbook M1 Pro (with 10 cores).
我们通过测量 SHA-256 在 4KB 原像输入的证明生成期间的平均 CPU 利用率来评估每个证明系统的并行化程度。下表显示了 Linux Server（20 核）和 Macbook M1 Pro（10 核）上的平均 CPU 利用率（括号中为每个内核的平均利用率）。

The key observations are as follows:
主要观察结果如下：

- Gnark and rapidsnark show the highest CPU utilization on the Linux Server, indicating their ability to efficiently use multiple cores and parallelize proof generation. Halo2 also demonstrates good parallelization performance.
- Most frameworks demonstrate 2x CPU utilization on the Linux Server as compared to the Macbook Pro M1, the exception to this is snarkjs.
- Despite initial expectations that FRI-based frameworks (Plonky2 and Starky) might struggle to use multiple cores effectively, they perform no worse than some Groth16/KZG frameworks in our benchmarks. It remains to be seen if there will be any differences in CPU utilization on a machine with even more cores (e.g., 100 cores).

- Gnark 和 rapidsnark 在 Linux 服务器上表现出最高的 CPU 利用率，表明它们能够有效地使用多核且并行化生成证明。Halo2 也展现了良好的并行化性能。
- 大多数框架在 Linux 服务器上的 CPU 利用率是在 Macbook Pro M1 的 2 倍，只有 snarkjs 例外。
- 尽管最初预计基于 FRI 的框架（Plonky2 和 Starky）可能难以有效地使用多核，但它们在我们的性能测试中的表现并不比某些 Groth16 或 KZG 框架差。在具有更多内核（例如 100 个内核）的机器上，CPU 利用率是否会有差异还有待观察。

Proving System	**CPU Usage (avg per-core usage) (Linux Server)**	**CPU Usage (avg per-core usage) (MBP M1)**
snarkjs	557% (27.85%)	486% (48.6%)
rapidsnark	1542% (77.1%)	N/A
gnark	1624% (81.2%)	720% (72%)
Arkworks	935% (46.75%)	504% (50.4%)
Halo2 (KZG)	1227% (61.35%)	588% (58.8%)
Plonky2	892% (44.6%)	429% (42.9%)
Starky	849% (42.45%)	335% (33.5%)

证明系统	**CPU** 利用率（平均每核利用率） **(Linux** 服务器**)**	**CPU** 利用率（平均每核利用率） **(MBP M1)**
snarkjs	557% (27.85%)	486% (48.6%)
rapidsnark	1542% (77.1%)	N/A
gnark	1624% (81.2%)	720% (72%)
Arkworks	935% (46.75%)	504% (50.4%)
Halo2 (KZG)	1227% (61.35%)	588% (58.8%)
Plonky2	892% (44.6%)	429% (42.9%)
Starky	849% (42.45%)	335% (33.5%)

Conclusion and Future Work

结论及未来研究

This blog post presents a comprehensive comparison of the performance of SHA-256 on various zk-SNARK and zk-STARK development frameworks. Through the benchmark results,

we've gained insights into the efficiency and practicality of each framework for developers who require succinct proofs for SHA-256 operations. Groth16 frameworks (e.g., rapidsnark, gnark) are found to be faster in generating proofs than Plonk frameworks (e.g., Halo2, Plonky2). The lookup table in Plonkish arithmetization significantly reduces the constraints and proving time for SHA-256 when using a larger pre-image size. Furthermore, gnark and rapidsnark demonstrate an excellent capability to utilize multiple cores for parallelization. Starky, on the other hand, shows a much shorter proof generation time but at the cost of a much larger proof size. In terms of memory efficiency, rapidsnark and Starky outperform other frameworks.
这篇文章全面比较了 SHA-256 在各种 zk-SNARK 和 zk-STARK 开发框架上的性能测试结果。通过比较，我们深入了解了每种框架的效率和实用性，以期可以帮助需要为SHA-256 操作生成简洁证明的开发者。我们发现 Groth16 框架（例如 rapidsnark、gnark）在生成证明方面比 Plonk 框架（例如 Halo2、Plonky2）更快。Plonkish 算术化中的查找表在使用较大的原像尺寸时显着减少了 SHA-256 的约束和证明时间。 此外，gnark 和 rapidsnark 展示了利用多核以并行化运作的出色能力。另一方面，Starky 的证明生成时间要短得多，但代价是证明大小要大得多。在内存效率方面，rapidsnark 和 Starky 优于其他框架。

As the first steps to building the Pantheon of ZKP, we acknowledge that this benchmark result is far from being the final comprehensive testbed we aim for it to one day be. We welcome and are open to feedback and criticism and invite everyone to contribute to this initiative of making ZKP easier and more accessible for developers to use. We are also willing to provide grants for individual contributors to cover the costs of computational resources for large-scale benchmarking. Together, we can improve the efficiency and practicality of ZKP for the benefit of the wider community.
作为构建零知识证明评测平台「万神殿 Pantheon」的第一步，我们承认本次性能测试结果远不足以成为最终我们希望构建的一个综合测试平台。我们欢迎并乐于接受反馈和批评，并邀请所有人为这项倡议做出贡献，以便开发者更容易、低门槛地使用零知识证明。我们也愿意为个人独立贡献者提供资助，以支付大规模性能测试的计算资源成本。我们希望可以共同提高 ZKP 的效率和实用性，更为广泛地造福社区。

Finally, we'd like to thank the teams from Polygon Zero, the gnark team at Consensys, Pado Labs, and Delphinus Lab for their valuable review and feedback on the benchmarking results.
最后，我们要感谢 Polygon Zero 、Consensys gnark 、Pado Labs 以及 Delphinus Lab 团队，感谢他们对性能测试结果的宝贵审查和反馈。