@abipalli, @alecb355, @wenbozhu24

Surveying Zero-Knowledge SNARKs for Registrar Systems

Overview

Background

Zero-knowledge proofs are systems that allow one party to prove to another that a statement is true without revealing any additional information about the secret witness. There are two main examples: **STARKs** and **SNARKs**. **STARK** stands for Scalable Transparent Argument of Knowledge. **SNARK** stands for Succinct Non-Interactive Argument of Knowledge.

Non-interactive means the prover will produce a static proof string, so the prover can post the proof string in public for others to verify independently instead of requiring any interaction or communication between the prover and the verifier; Succinctness in the context refers to the fact that the proof string will be non-trivially short and fast to verify, at least sublinear with respect to the length of the witness. And *zero-knowledge* means that verifier learns no information about the witness from the proof string, which is what makes it fundamentally different from other crypto protocols that are encryption-centric: In zk-SNARKS, we do not encrypt the secret itself; we bypass it entirely.

The key difference between STARK and SNARK is that a SNARK relies on a trusted setup where the secret keys that generate the proof are subsequently destroyed. STARKs don't require this trusted setup and have more secure protocol, but as a result they are less efficient. STARKs are better in decentralized systems where there are no trusted parties whereas SNARKs are a better fit if there is a centralized trusted entity that generates the proofs.

Objective

Our goal is to design an application for 3rd parties to verify membership status in an organization. The enrollment status for "members" must be verifiable without needing to expose Personal Identifying Information (PII) to the "verifier" parties.

The security requirements of this system led us to consider zero-knowledge proofs. Given that the membership "prover" is a *trusted* entity in our scenario, we decided to use zk-SNARKs for our application. In addition, we are mainly concerned with efficiency of the application as opposed to security, and the proofs generated by SNARKS are smaller and execute faster.

In experimenting the feasibility and scalability of zk-SNARKs in enterprise applications, this paper identifies its usage in attestation services that determine membership status in registrar applications. Goals for this application are:

1. Members should not have to reveal personally-identifiable information (PII)
2. Verifiers should be able to verify which year(s) a student is/was enrolled
3. Verifiers should be able to "reconstruct" the proof at any time
4. The registrar service is responsible for data consistency, integrity, and availability

Methodology

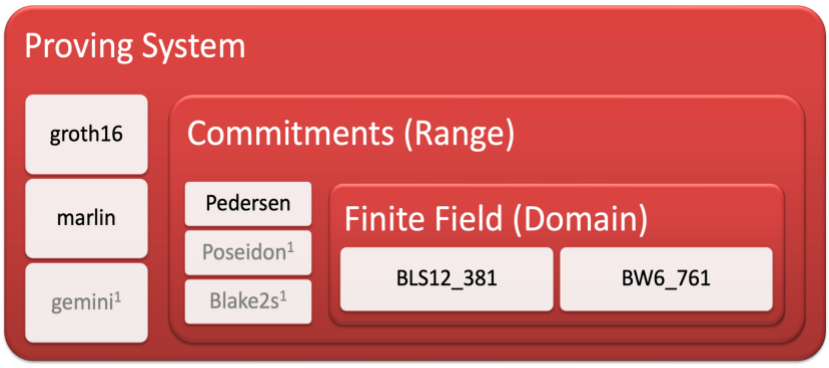


Figure 1. Visualization of zkSNARK construction.

A zk-Circuit is an encapsulation of 3 fundamental primitives: (1) a Proving System, (2) a Commitment Schema, and (3) a Finite Field – visualized in **Figure 1**. The finite field requires transformation of the universal numbers to the constrained domain. The commitment schema defines how values are cryptographically encoded to ensure binding and hiding properties over the witness and inputs. The proving system orchestrates the generation and verification of succinct proofs over the committed values using specific constraint systems and algebraic structures. The leading configuration in most zero-knowledge proofs is groth16 circuits with pedersen commitments (ie. hashes)[1].

This paper evaluates 2 proving systems (i.e. groth16, marlin), 1 commitment schema (i.e. Pedersen commitments), and 2 finite fields (i.e. BLS12_381 and BW6_761). **Figure 1** highlights these decisions and mentions additional alternatives to consider for future experimentation and for a better holistic decision.

Finite Fields

Finite fields are mathematical structures with a fixed number of elements where basic operations—addition, subtraction, multiplication, and division—are always defined (except division by zero). In zero-knowledge systems, they form the arithmetic backbone for operations like polynomial computations and circuit constraints, with the field's size affecting both efficiency and security.

Elliptic Curve

Elliptic curves are sets of points that satisfy a specific algebraic equation over a finite field and possess a natural group structure. In zkSNARKs, they are essential for creating hard problems (like the discrete logarithm problem) and for efficiently implementing pairing operations used in proof verification. The choice of elliptic curve — including its coordinate representation and pairing properties — is key to balancing performance with robust security.

[1] Benarroch, D., Campanelli, M., Fiore, D. *et al.* Zero-knowledge proofs for set membership: efficient, succinct, modular. *Des. Codes Cryptogr.* 91, 3457–3525 (2023). https://doi.org/10.1007/s10623-023-01245-1

In the current implementation, we use BLS12-381 and BW6-761 respectively for our implementations. These curves have different bit lengths and thus cause performance differences. One thing to notice while implementing is that the differences in these field sizes would require adjustment to the hash parameters used in the Merkle tree. Specifically, since each BLS12-381 element is encoded in 48 bytes and BW6-761 96 bytes, we need to to change the parameters for hash gadgets so that `WINDOW_SIZE * NUM_WINDOWS` to match the bit lengths of these fields. The hash parameters themselves can also be considered hyperparameters that could be profiled to see whether it would affect runtime of our application.

Proving Systems

Recall the requirements in configuring a SNARK circuit involves a trusted setup process by a (thereby) trusted proving party — as mentioned in the paper's context. More specifically, the trusted process feeds the circuit public and private witnesses, where public witnesses can henceforth be fed by verifiers while the private witness is encoded into the verifier's key. The effects of how the constraint system is derived and the private witness allocated and encoded into the verification key is studied in the following proving systems.

Groth16

The first iteration of our application leverages the Groth16 proving system due to its established efficiency in verification procedures and relatively compact proof sizes. Groth16 is particularly suited for scenarios where proofs need to be verified quickly with minimal computational effort, which aligns with the project's objective to generate a witness-based proof of a student's enrollment in an institution, while allowing for verifiers to "cheaply" verify the enrollment status without incurring additional overhead and obfuscating the student's PII.

Marlin

Marlin proofs support a universal setup, which means that a single trusted constraint system can be used for various circuit logics as long as the constraint matrix is satisfied by the universal setup process. This overcomes the mentioned limitation of Groth16 proofs where any change to the underlying data structure invalidates past proofs and warrants regenerating new proofs and supporting cryptomata (e.g. prover and verifier keys).

Additionally, Marlin's universal constraint setup also enables future circuit logic applications on top of membership attestation — e.g. membership metadata evaluation such as age — to be verifiable by a single verification key. This overcomes the need to maintain distribution and mapping consistency between a verification key and the circuit application, which is necessary for Groth16 systems; and a solvent approach to **Objective #4**.

Framework

Arkworks

We will be using **Arkworks** to build our project[2]. Arkworks is an open-source ecosystem written in Rust, designed for developing zkSNARKs. It provides efficient implementations for all components required to implement zkSNARK applications, from generic finite fields to R1CS constraints for common functionalities. The ecosystem includes libraries for finite fields, elliptic curves, polynomial arithmetic, and SNARK proving systems like Groth16 and Marlin. It's widely used in blockchain and privacy-focused projects.

Registrar Data Structure

Recall that SNARK circuits are designed around some public and private witnesses. Though the registrar system need not manage members in a Merkle Tree data structure, it is prudent to build the aforementioned circuit witnesses by migrating members in a Merkle tree. This is because the understanding the association of the individual member hashes and the Merkle Tree's aggregate root hash is an NP-complete problem — a class of decision problems for which any proposed solution can be verified quickly, but finding a solution is believed to be hard. Per Merkle Trees, it is more trivial to determine whether a root is valid for a merkle tree after recursively computing the structure, than it is to identify all possible leaf hashes that may be encapsulated by the aggregate root hash.

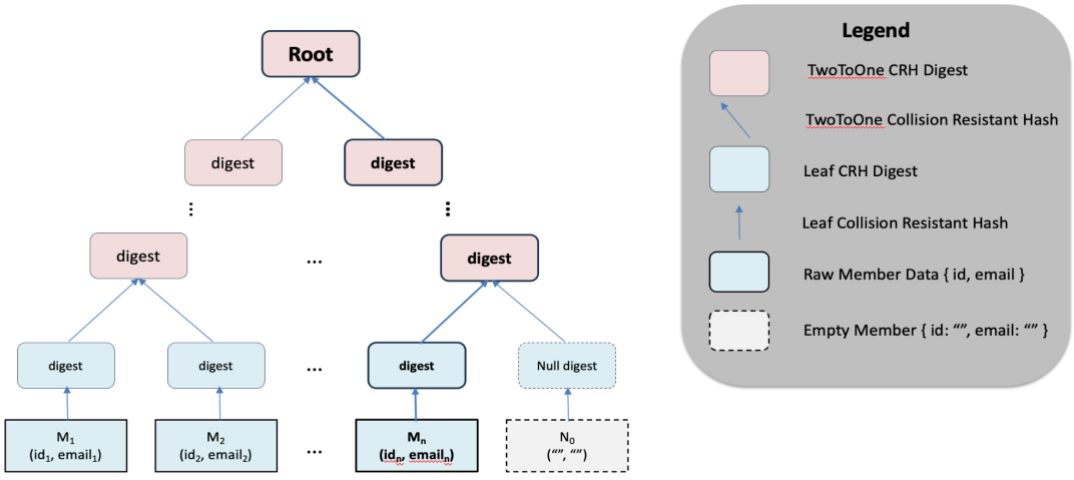


Figure 2. Merkle Tree data structure is best suitable for trivial membership attestation based on 2 public witnesses *(the tree's root and the member's leaf hash)* and a single private witness *(merkle path)*.

For our application, we leverage two Collision-Resistant Hashes (CRH) that are constrained to a specified finite field passed as a generic; a LeafCRH is responsible for producing a unique digest for each leaf in the tree, and the TwoToOneCRH produces a digest from 2 inputs. Note that when the number of members is not a power of 2, the tree structure requires the number of leaves to be padded with some "default" value to derive a valid root for a binary Merkle tree. This is addressed by creating *default members* with

[2] https://github.com/arkworks-rs

an empty ID and Email, and a joined timestamp that matches the UTC timestamp of when the member is instantiated.

Evaluation

The benchmark tooling for this project is publicly available on GitHub[3] and orchestrated on several runtime environments to determine the decision trade-offs between the aforementioned Finite Fields (i.e. **BLS12_381** and **BW6_761**) and SNARK Proving Systems (i.e. **Groth16** and **Marlin**).

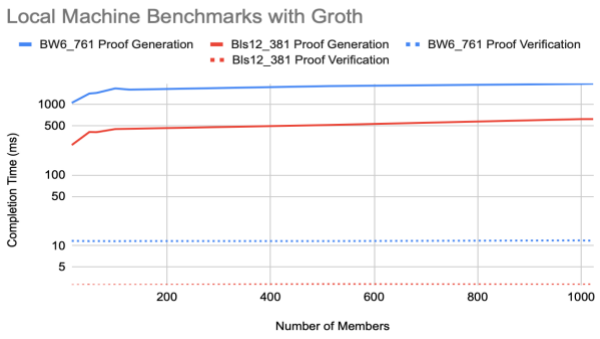
Setup

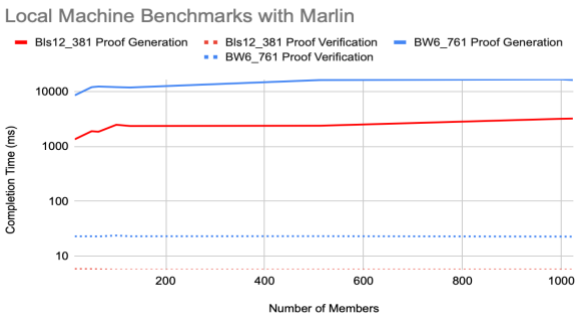
We evaluated the different systems on a few machines with a variety of capabilities. Locally, we tested on a 2016 MacBook Pro, equipped with 10 cores and 32 GB of memory with a clock speed of 3.2 GHz. Remotely, we tested on three machines through Google Cloud: c2-standard-8 with 4 cores and 32 GB memory, c2-standard-16 with 8 cores and 64 GB memory, and c2-standard 30 with 15 cores and 120 GB memory. All of these remote machines have a clock speed of 3.1 GHz and run Linux. The MacBook Pro runs on MacOS.

Finite Fields: BLS12_381 vs BW6_761

Figures 3 – 6 illustrate that computing circuits over the BLS12_381 field is significantly more efficient both in runtime and memory than BW6_761. This is expected most prominently due to the size of the field for BW6_761 being nearly double that of BLS12_381. Computation on larger prime fields incur more computation and memory overhead given the required buffer size of information, and the larger domain space of valid data. Proof verification on both Groth16 and Marlin was around 10 times faster with BLS12_381 than BW6_761. Proof generation and memory allocated was around 3 times smaller with BLS12_381. We were unable to measure the memory usage for BW6_761 on Marlin, but when we stopped execution for 16 members, there had been 8.5 GB allocated, so we expect a similar trend to Groth16 where BW6_761 would have required roughly 3 times the memory.

Figure 3 – 4. Benchmark metrics ran on "Local Machine" comparing Finite Fields and Proving Systems.





[3] https://github.com/abipalli/zkmember

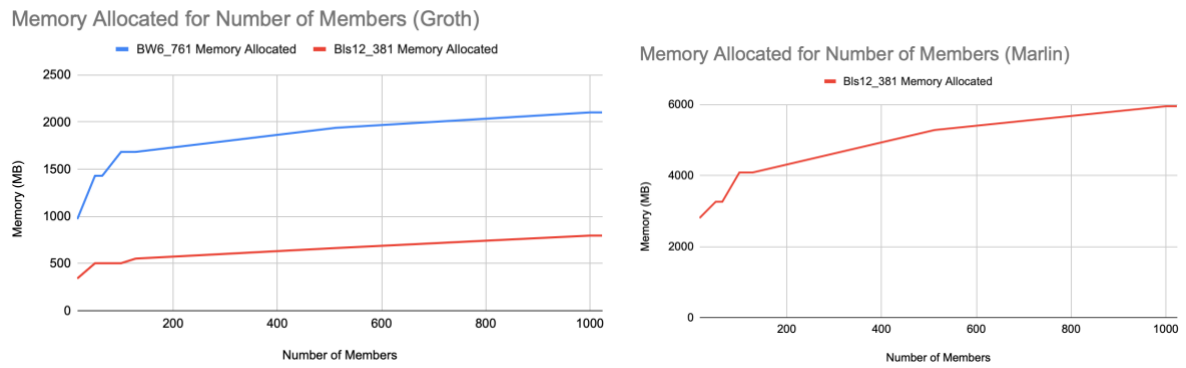


Figure 5 – 6. Memory allocation for Circuit Benchmarks. *Marlin BW6_761 testing did not finish.*

Runtime Environments: Local vs Remote

Figures 7 and 8 capture a baseline benchmark of Groth16 proofs computed over the BLS12_381 field, to better understand the computational difference between the local and remote runtime environments. From the graphs above, it's clear that the MacBook Pro outperformed all of the remote machines in both proof generation and proof verification. This is likely due to the fact that these remote machines are running on virtual CPUs and are competing with other virtual machines for the same resources, whereas the local MacBook Pro has access to all its physical cores. As expected, when the remote machine is vertically scaled (ie. provisioned more cores and memory) the performance on the benchmarks improved.

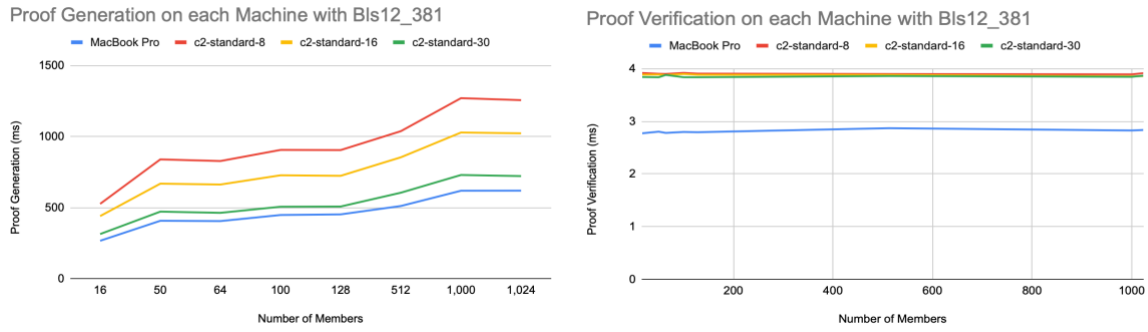


Figure 7 – 8. Proof Generation and Verification metrics of Groth16 proofs over BLS12_381 for varying registrar sizes.

The **proof generation time increases logarithmically as the number of members increases**. Moreover, we see jumps in completion time when the ceiling of $\log_2$(Number of Members) increases. This is understandable because the private witness for the circuit is the "authentication path" — the path from the root of the Merkle tree to the leaf node — which is always $\log_2$(Number of Members) long. The graphs visualize this by plateauing performance between 50 and 64, 100 and 128, and 1000 and 1024 members. This is because even though there are fewer members, the Merkle tree must still be padded with "empty" members to generate the final proof.

Proof verification time stays constant as the number of members grows. This is because the final proofs generated have the variable private witness encoded into the verification key and the public witnesses are

always of constant size regardless of registrar size — the Member LeafCRH digest and Root TwoToOneCRH digest. Similar to the proof generation, the remote machines all performed very similarly, and were outperformed by the local machine most likely due to the overhead of the Google Cloud Provider hypervisor scheduling tasks on shared cores.

Proving Systems: Groth16 vs Marlin

In **Figure 9**, we can see the memory allocation of Marlin is significantly more than Groth16. Regardless of the numbers of members, the memory allocated by Marlin is roughly 8 times more than the memory allocated by Groth16. The difference in Marlin vs Groth16 proofs are attributed to the universal constraint system that Marlin circuits support, thereby yielding additional variance and context encoded into the verifying key. When comparing the proof generation times for both, we clearly see that on every machine Groth16 outperforms Marlin as the number of members grows.

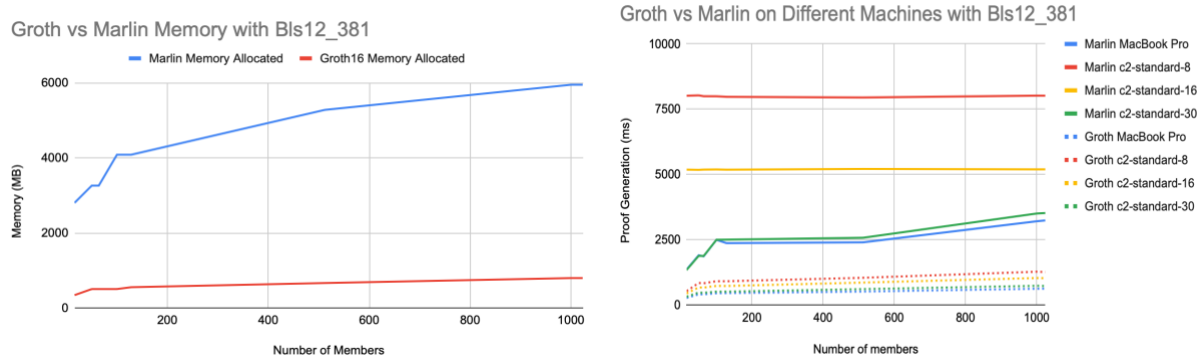
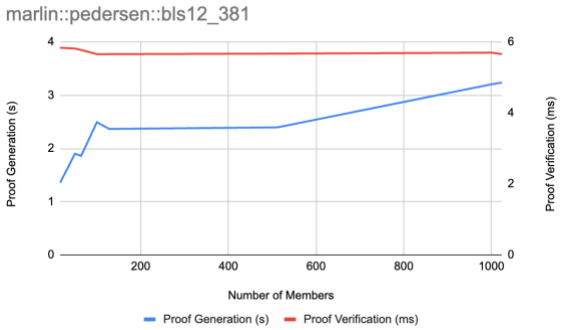


Figure 9 – 10 (above). Memory Allocation and Proof Generation metrics for Groth16 and (Circuit-specific) Marlin over BLS12_381

Figure 11 (right). Proof Generation and Verification metrics for Marlin circuits with Universal constraints. *Note the consistent computational load given a shared constraint system between registrar sizes. The constraint system is refined for the largest registrar size (1024) given that it has the most number of constraints (i.e. longest Merkle Path).*



Challenges

Measuring Memory Usage

In our intermediate report, we mentioned the challenge of measuring memory allocated in our systems. We tried a variety of ways to measure the memory locally, but they all proved to be unsuccessful. Fortunately, we were able to use valgrind on the remote linux machines to measure memory allocation, but this came with its own limitations. The overhead of valgrind made all of our tests take significantly longer. For example, when measuring the memory allocation for BW6_761 on Groth16, it took the

machine over an hour to complete a test with 128 members. When trying to measure memory allocation for BW6_761 on Marlin, the machine didn't complete execution in three hours even with just 16 members, which meant we weren't able to measure memory allocation at all for BW6_761 on Marlin.

Conclusions

Per Objective

This is trivially addressed given that registrar attestation is handled via LeafCRH Digests — a hash of the member's metadata — thereby not requiring any explicit / specific information of the member to be disclosed. This means that members may yield data to verifiers at their discretion and allow for verifiers to reconstruct the attestation as the verifier has the necessary information needed to reconstruct the LeafCRH hash that is captured by the Merkle Root. The zkSNARK circuit simply allows for verifiers to confirm inclusion of the attested member data in the Merkle tree without learning anything beyond the fact of membership, including the Merkle Root, the Merkle Path, or any auxiliary information.

"Which year a student is enrolled in" is achievable based on when and how the registrar proof is constructed. If the registrar proof is constructed via **Groth16 proving SNARK**, then it is important that verifiers access the specific verifying key produced when the Merkle Tree proof was first constructed. Otherwise, if the proof is constructed via a **Marlin proving SNARK**, then the registrar system has the ability to distribute proofs tethered to a single verifying key without needing to worry about versioning or data consistency for the verification keys and its distribution to verifiers.

Per Proving System

If the registrar system prioritizes compute efficiency (e.g. "low-spec" devices), then the registrar system should consider groth16 circuits. If the registrar system instead wants to ensure multiple proofs or varying constraint-bound circuits to be verifiable by single key, or possibly preserve constraint-bound backwards-compatible proofs then the registrar system should prioritize marlin circuits in its proof construction.

Future Work

It is worth evaluating the effects of tinkering with the Finite Field hashing configurations — specifically the WINDOW SIZE and NUM_WINDOWS — to determine how this additional independent variable affects the overall proof generation performance; this would not affect the proof verification performance unless the adjustments change the CRH digest size.

Additionally, this experiment only measures Pedersen commitments. Pedersen is generally preferred over Blake2s and Poseidon commitments for benchmark papers because they are not as trivial as Poseidon commitments and are not too complex for circuit computation as Blake2s commitments. Compared to the other two, Pedersen commitments provide enough computational complexity to afford enough complexity to the implemented finite fields without completely bottleneck performance to the circuit framework. Nevertheless, real-world implementations may opt to lean heavily towards optimizing for performance or complexity, and therefore should be considered in a holistic survey for registrar applications.