microLean and macroLean

Anthony Wang, Rey Li, Zach Marinov, Dan Klishch

May 13, 2026

Abstract

The paper explores two approaches for building zero-knowledge proofs (ZKPs) that certify correctness of arbitrary mathematical proofs. We introduce a bespoke verification language μLean and implement a proof checker for μLean in a zk-SNARK-oriented language. Separately, we benchmark performance of generating ZKPs using an existing Lean proof checker running inside a zkVM. Our results demonstrate that both approaches are viable but each runs into different limitations of current ZKP tooling.

1 Introduction

Formal verification and zero-knowledge proofs (ZKPs) are two powerful approaches for establishing trust in computation. Formal verification uses proof assistants such as Lean to construct machine-checked mathematical proofs, while ZKPs allow one party to convince another that a computation was performed correctly without revealing the underlying witness.

Although both systems are fundamentally concerned with proving correctness, they operate on very different computational models. Proof assistants such as Lean verify proofs using type checking, whereas modern zero-knowledge systems are designed around arithmetic circuits or low-level virtual machine execution. Consequently, generating ZKPs for mathematical proofs requires either encoding a proof assistant's type checker into a circuit-friendly representation or executing the type checker inside a zkVM.

This paper investigates both approaches. The first approach, microLean (abbreviated μLean), implements a simplified Lean-inspired proof assistant directly in a zk-SNARK-oriented language. The second approach, macroLean, instead executes

an existing Lean type checker inside a zkVM. We evaluate both approaches and compare their expressiveness, implementation complexity, and performance.

The remainder of this paper is organized as follows. Section 2 introduces the necessary background on formal verification, Lean, and ZKPs, and states the problem we are solving. Section 3 describes μLean and its simplified type system and vernacular. Section 4 presents macroLean, a pipeline for executing the Lean type checker inside a zkVM. Sections 5 and 6 compare the performance and tradeoffs of both approaches, and Section 7 discusses directions for future work.

2 Background

2.1 Formal Verification and Lean

Formal verification is the process of checking mathematical proofs using proof assistants. The most common approach for implementing a proof assistant is based on the Curry-Howard correspondence [12], which states that every system of logic corresponds to a type system and vice versa. Under the correspondence, propositions in logic correspond to types in programming languages, while proofs correspond to programs inhabiting those types. Proving a theorem is therefore equivalent to constructing a function with a particular type.

Consequently, theorem proving in such systems is just type checking. Throughout this paper, we use the terms "proof checker" and "type checker" interchangeably.

Lean is an example of such a system [10]. It is a dependently-typed functional programming language and proof assistant widely used for formal verification. Its mathematical library, mathlib, contains formalizations of large portions of undergraduate mathematics. Lean has also been used in industry to formally verify security-critical software systems.

2.2 Zero-Knowledge Proof Systems

Zero-knowledge proof systems allow a prover to convince a verifier that a computation was executed correctly without revealing the private witness used during the computation. Modern systems typically produce succinct proof certificates that can be verified substantially faster than re-executing the original computation.

zk-SNARK systems represent computations as arithmetic circuits or constraint systems [7]. More recently, zkVMs have emerged as an alternative to writing ZKP circuits by hand. In them, low-level virtual machine instructions and executed inside

a zero-knowledge proving system. Compared to handwritten circuits, zkVMs significantly reduce implementation effort and allow existing software to be reused with minimal modification, although with a significantly increased proving overhead.

2.3 Problem Formulation

The overall goal of this work is to generate a ZKP for the statement that a theorem admits a valid proof under Lean's type system. More formally, we wish to prove the following statement:

$$\exists\ p,\ \mathsf{typecheck}(T, p) = \mathsf{true}.$$

Here, T denotes a theorem statement and p denotes a proof term inhabiting that theorem. The private witness corresponds to the proof term itself, while the computation being verified is execution of the proof assistant's type checker. The verifier learns only that a valid proof exists and type checks successfully, without learning the proof itself.

This task has several potential applications. One application is privacy-preserving verification of proprietary software. In many settings, the proof of correctness for a system may itself reveal sensitive implementation details, which we can hide with a ZKP. Another application is a new model for responsible security disclosure. A security researcher could publicly prove the existence of a vulnerability or exploit without revealing the exploit itself. This has been recently done by Google to announce a faster quantum algorithm for breaking elliptic curve cryptography [6].

2.4 Related Work

The zkPi project at Stanford explored a similar approach for generating ZKPs for Lean [8]. Their work demonstrated the feasibility of constructing zk-SNARKs for Lean proof checking using a custom circuit representation for a subset of Lean's type system. However, the implementation omits several important features of Lean, including quotient types and η-reduction, and relies on manually designed circuits for the proof checker.

3 μLean

Our first approach was to write a type checker ourselves in a zero-knowledge proof language. We chose Lurk [1], a high-level Lisp-like programming language for zk-SNARKs, instead of writing circuits manually.

3.1 Type System and Vernacular

Similar to zkPi, our type checker only supports a subset of Lean's type system, which we call μLean. Specifically, the μLean type system is based on the calculus of constructions [4], with cumulative type universes and a few hardcoded inductive types such as the natural numbers. This type system is equivalent to higher-order intuitionistic logic and can express many theorem statements such as Fermat's Last Theorem. Although μLean can be used as a programming language and we have successfully implemented the Fibonacci function in μLean, it is not Turing complete. This is because it forbids infinite recursion, which would otherwise allow for writing unsound induction proofs without a base case.

We determined it was unfeasible to automatically translate real Lean proofs to μLean because of μLean's lack of advanced type system features that are extensively used in real Lean proofs. Thus, we designed our own high-level syntax for writing mathematical proofs, called the μLean vernacular, that can be desugared into the AST that the type checker understands. The μLean AST is very basic and even lacks variable names, using de Bruijn indices instead [3], so the desugaring process handles conversion from the former to the latter. We reused Lean's parser and elaborator to parse the vernacular and wrote around 300 lines of Lean code to perform desugaring.

For example, the proof of $\forall A, \forall B, A \to B \to A \wedge B$ in the μLean vernacular is:

```
la ['α, 'β]
  (ab pmk ['α, const 'β]),
  (α◆U ⇨ β◆U ⇨ 'α ⇨ 'β ⇨ prod 'α (const 'β))
```

When desugared, it becomes the following, where the numbers followed by ns are finite field elements:

```
'((1n (1n (2n (2n (6n) (4n (3n 0) (4n (4n (0n 0) (3n 0))
↪   (4n (0n 1) (4n (2n (0n 1) (4n (0n 2) (3n 0)) (0n 0))
↪   (5n (0n 3) (0n 2)))))) (0n 1)) (4n (4n (0n 1) (3n 0))
↪   (4n (0n 2) (4n (2n (0n 1) (4n (0n 3) (3n 0)) (0n 0))
↪   (5n (0n 4) (0n 2))))) (1n (0n 1))))) . (4n (3n 0) (4n
↪   (3n 0) (4n (0n 1) (4n (0n 1) (5n (0n 3) (1n (0n
↪   3)))))))))
```

In comparison, the equivalent Lean proof of that theorem is:

```
example : ∀ A B : Prop, A → B → A ∧ B := by
  intro A B hA hB
  exact ⟨hA, hB⟩
```

Additionally, we formalized several classic proofs such as the irrationality of $\sqrt{2}$ and the infinitude of primes in the μLean vernacular. For instance, the proof of the commutativity of addition is:

```
la ['n, 'm]
  (ab nat_rec [
    la ['m] ('n + 'm =? 'm + 'n),
    □ add_zero_eq_zero_add ['n],
    la ['m, 'h]
      (□ rw [
        ℕ, suc ('m + 'n), suc 'm + 'n,
        la ['x] (suc ('n + 'm) =? 'x),
        □ succ_add ['m, 'n],
        □ rw [
          ℕ, 'n + 'm, 'm + 'n,
          la ['x] (suc ('n + 'm) =? suc 'x),
          'h, ab refl [ℕ, suc ('n + 'm)]
        ]
      ]),
    'm
  ]),
  n◆ℕ ⇨ m◆ℕ ⇨ 'n + 'm =? 'm + 'n
```

We first implemented the μLean type checker in around 100 lines of Lean as a reference implementation, and then ported it to Lurk. μLean satisfies the de Bruijn criterion for proof assistants, which is the property of having a small type checker and large syntactic sugar layer, which is useful since only the type checker needs to be trusted and ported. The Lean implementation is also partially formally verified using denotational semantics so that we can be more confident of its correctness, and in turn the correctness of the Lurk port.

3.2 Achieving Cryptographic Guarantees with Lurk

Using the work from the preceding section we can write the μLean type checker, various theorem statements, and various proofs in the Lurk language. Here we discuss how we achieve cryptographic guarantees once in the Lurk environment.

There are two key primitives that make Lurk suitable for generating zk-SNARKs:

1. The proof and protocol API, which allows for the generation of SNARKs for specific statements. Provers can create SNARKs for statements of the form

`(expr, env) -> res`, meaning a public prover-defined expression evaluated over a public prover-defined environment leads to some public result. Verifiers can define protocols which require SNARKs of a certain form in order to be accepted.

2. Commitment primitives, which allow for zero-knowledge proofs. Users can hide values (with a random key nonce) and open values. One can only open a hash if the pre-image is known (previously computed in the Lurk REPL). Thus, a user can hide a private value and send the resulting hash over to create a cryptographic commitment; the hash can only be opened to the one specific value that the receiver cannot determine until the user doing the commitment reveals the private value.

The key idea is to provide a SNARK for a statement of the form:
`((check (open proof-hash) statement), env) -> t.`
The information that gets leaked through this SNARK is the hash of the proof, but not the actual proof itself. This will successfully run on the provers' computer because they know what the proof hash is supposed to open to and so the REPL is able to open the hash. However, verifiers will be unable to run the code directly because they are unable to open the hash; the SNARK is what provides evidence that the prover was able to do so successfully.

Using these primitives, the structure of our cryptographic exchange occurs in three steps:

1. A verifier releases a public protocol (using the Lurk protocol API) that only accepts hashes of proofs which the typechecker accepts against the protocol-specific theorem statement.

2. A prover which seeks to demonstrate knowledge of a proof for the theorem statement can run the proof API in Lurk, which leads to a proof file. This proof file can now be shared to demonstrate knowledge, but does not leak any information about the proof (only its hash).

3. The verifier can then verify that the public protocol accepts the resulting proof file. If accepted, the verifier can be confident that the prover is aware of a proof that proves the theorem statement.

The soundness of the system is based on the security of Lurk's commitment and SNARK primitives and the correctness of our type-checker implementation in Lurk. Any bugs in the type-checker could potentially be exploited to create a bogus

proof that will generate a valid SNARK. This kind of exploit has been done in practice; the code Google used to demonstrate the correctness of their quantum algorithm had bugs in it that Trail of Bits was able to exploit in order to create a valid zk-SNARK claiming knowledge of a quantum algorithm even more efficient than Google's, without actually coming up with such an algorithm [11].

4 macroLean

4.1 Motivation

Prior work to generate zk-SNARKs for Lean proofs ends with the aforementioned zkPi project [8], and does not include all features of the Lean programming language, leaving a clear path for improvement. In addition, the authors hand-wrote a circuit for their system, which is a lengthy and complex process. Therefore, we sought to build a system that could scale to generate zk-SNARKs for Lean proofs using all of Lean, without any manually written circuits.

4.2 Construction

The project is composed of a number of pre-existing primitives. We export proofs written in Lean to include all relevant dependencies using `lean4export` [9]. Our Lean type checker is `nanoda_lib` written in Rust [2]. To generate zk-SNARKs, we use a combination of the SP1 zero-knowledge virtual machine and the PlonK proof system [13, 5].

We will now walk through how the system works. The system is architected as an untrusted host program which runs a trusted guest program. The two communicate via a single input channel (whose contents the guest can independently reconstruct from the deterministic host code) and a single output channel (whose contents the SNARK signs). The host is the scaffolding we need, within which we execute the guest, including the type checker in the zkVM environment.

Before it receives any Lean proof, the system has already compiled `nanoda_lib` to RISC-V instructions. Given a Lean proof for which the user seeks a zk-SNARK, macroLean does the following. The host exports the Lean proof to an `ndjson` file using `lean4export`. The host then compiles this to a binary blob, paired with a SHA-256 hash of the blob, and passes this to the guest environment, as input for our compiled type checker to run in the RISC-V emulator. Within this environment, the type checker executes the program and SP1 generates a STARK certifying correct execution of the type checker on the input program. SP1 also has a STARK verifier

written as a PlonK circuit, and the final step of this process is running the PlonK prover on this circuit and the STARK to generate a zk-SNARK.

4.2.1 Modifications made to `nanoda_lib`

In order to make it feasible to run `nanoda_lib` within the zkVM, we had to make some modifications to the codebase, none of which compromise its ability to typecheck Lean proofs. Specifically, we removed the `pretty_printer` and `debug_printer` functionality, as the guest does not need to print anything. We removed the `main.rs` launcher as the host is responsible for orchestrating the system. We removed support for parallel type-checking (`tc::check_all_declars_par`) because the zkVM only supports single-threaded execution. We removed the JSON config reader `Config::TryFrom<&Path>` because we replace the JSON config file with a hardcoded config for security reasons. And we replace the NDJSON parser `parser::parse_export_file` and associated `serde_json` crate because we decided to pre-parse the NDJSON on the host side for performance.

4.3 Cryptographic Soundness

Each proving run of macroLean gives us three cryptographically-relevant objects:

- Verifying key (identifies the guest program run in the zkVM host)
- Public values (`input_hash`, `theorem_anchor`, `num_declars`)
- zk-SNARK proof

The verifying key is a hash of the guest program, so what the zk-SNARK informally states is "I know an execution trace such that the guest program with the given verifying key and the following hashed public values ran cleanly." To be precise, the zk-SNARK certifies correct execution of a STARK verifier, running on a STARK generated by SP1 running the compiled `nanoda_lib` RISC-V on a binary input whose hash is `input_hash`, whose structured theorem statement is represented by `theorem_anchor`, with `num_declars` declarations (kernel-level terms in Lean such as `Axiom`, `Theorem`, and `Definition`).

We now demonstrate the soundness of our construction, stating our assumptions clearly. We analyze against two adversaries - adversary A, which tries to learn information about the original Lean proof from the outputs, and adversary B, which tries to generate a proof certificate of a theorem for which it does not have a proof.

Adversary A is unable to determine any information about the original proof by the soundness guarantees of PlonK [5].

We conceive of a few possible attacks for adversary B. Adversary B could try to pass in a binary blob corresponding to a proof it knows about, but then attempt to claim that it has actually proven something else, for which it does not have a proof. We defend against this attack via the `input_hash` and `theorem_anchor`, i.e. we have commitments to the input binary passed into the guest in the zkVM and to the theorem statement itself, and we obtain soundness here due to the collision-resistance of SHA-256. Adversary B could also try to tamper with `nanoda_lib`, whether in Rust form or in RISC-V form, and run a backdoored version. We defend against this with our verifying key, through which we know specifically what program was executed as the guest in SP1. Another attack is to tamper with SP1 itself to generate a false proof. However, this does not matter because as long as the verifier's SP1 instance can be trusted, an adversary that tampers with their SP1 instance as part of proof verification would then generate a proof which would fail verification.

Therefore, our construction is secure against these two kinds of adversaries, given the assumptions stated.

5 Results

We compared runtime and proof size of μLean and macroLean on the following three theorems:

- Triple negation implies single negation: $\neg\neg\neg A \to \neg A$ (Fig. 1a)
- Commutativity of addition by zero: $\forall n : \mathbb{N}, n + 0 = 0 + n$ (Fig. 1b)
- Commutativity of addition: $\forall n\ m : \mathbb{N}, n + m = m + n$ (Fig. 1c)

The code for these experiments can be found in the accompanying GitHub repository at `https://github.com/i-love-lean/6.5610-project/`.

6 Discussion

We proceed to analyze the expressiveness, complexity, and performance of our implementations.

The μLean approach only works for a narrowly typed language, in contrast to macroLean working on all of Lean.

Implementation	Generation	Verification	Proof size
Lean (no cryptography)	-	1.2ms	3KB
μLean (has ZK)	3.4s	930ms	3.1MB
macroLean (without ZK)	11.88s	57.8ms	2.66MB
macroLean (with ZK)	909.2s	291.92ms	4.05KB

(a) Results for $\neg\neg\neg A \to \neg A$

Implementation	Generation	Verification	Proof size
Lean (no cryptography)	-	2.3ms	58 KB
μLean (has ZK)	17.39s	602ms	3.3MB
macroLean (without ZK)	104.66s	151.06ms	4.17MB
macroLean (with ZK)	1144.06s	274.22ms	4.04KB

(b) Results for $\forall n : \mathbb{N}, n + 0 = 0 + n$

Implementation	Generation	Verification	Proof size
Lean (no cryptography)	-	2.6 ms	61 KB
μLean (has ZK)	66.91s	576ms	3.5MB
macroLean (without ZK)	120.34s	134.74ms	5.62MB
macroLean (with ZK)	957.10s	276.10ms	4.05KB

(c) Results for $\forall n\ m : \mathbb{N}, n + m = m + n$

Figure 1: Benchmark results comparing μLean and macroLean implementations.

What's more is the μLean approach required a substantial amount of work to write our own type checker and minimal proof assistant, and we expect that implementing a type checker for Lean's full type system would require an order-of-magnitude more code. In comparison, `nanoda_lib` ran with minimal modifications in a zkVM. Lurk also provided a difficult developer experience due to unclear errors. Ultimately, a more mature zk-SNARK language will be needed before practical use.

We did not encounter any obstacles towards using macroLean for larger theorems other than very slow performance. However, Lurk consistently crashed due to internal bugs when generating ZKPs for more complicated proofs. It also appears that Lurk stores the equivalent of the entire execution trace in memory which sometimes makes us hit out of memory errors even before reaching asserts. We tried to mitigate this by writing a bytecode-based type checker which was designed to run in linear time in μLean proof size. However, the constant factor was still too large to generate a zk-SNARK for the proof $\sqrt{2} \notin \mathbb{Q}$.

Currently, the speed of verification for both approaches is so slow that running the Lean type checker directly would be faster. The proof sizes for μLean and macroLean without the ZK component are also larger than the Lean source; however, the macroLean with the ZK does successfully have a smaller proof size compared to the Lean source; there does come a realistic point in which the succinctness of the proofs is relevant. Ultimately though, the primary advantage our system provides is the zero-knowledge component.

In addition, one of the benefits of macroLean is that the user can selectively switch the slow zero-knowledge feature on or off. If zero-knowledge is not important, the user can obtain a STARK in a much shorter time.

Overall, we can see that across all three theorems, macroLean with ZK (using PlonK) takes the longest to generate a proof, though proof verification is faster than Lurk's, and proof sizes are also smaller. Running these theorems directly in Lean is significantly faster (though proof sizes are not always the smallest), meaning the primary benefit our implementations provide is zero-knowledge.

7 Future work

7.1 μLean

A common problem when deploying zero-knowledge proofs in the real world is the difficulty of writing bug-free ZK statements. For instance, a bug in the Lurk implementation of μLean that enabled any statement to be proven using a malicious proof would completely undermine the soundness of our project. One possible solution is to leverage formal verification. The Lean implementation of μLean is partially verified so that we can be more confident in its correctness, but bugs could arise while porting the code from Lean to Lurk. To eliminate these potential bugs, we would also have to formally verify the Lurk implementation, which would be a very large and complex project. The high-level idea is to embed the semantics of Lurk into Lean and then prove that the Lean and Lurk implementations are semantically equivalent. However, we would still need to trust that the semantics we formalized accurately match the behavior of Lurk.

We also hope to add more features to the μLean type system such as general inductive types and quotient types to enable the auto-translation of actual Lean proofs into μLean.

The zk-SNARK protocol in Lurk could also use improvement; currently, the entire typechecker implementation is saved for every theorem proof as the environment variable, but it should be possible to only store a hash of the typechecker. This

would reduce the size of the resulting protocol file significantly, but not lead to much improvement in proof sizes (as that is mostly the zk-SNARK).

Due to the aforementioned flaws in Lurk, it might be worthwhile to port the μLean type checker to a different zero-knowledge proof language.

7.2 macroLean

As can be seen from the data, our zkVM implementation's PlonK proving step for generating the zk-SNARK takes a very long time. This is because of the fixed-size 27.6M-constraint circuit which SP1 wrote for their STARK-verifier which is used to generate the zk-SNARK. This will not scale with proof size, and can only be sped up through more efficient computation such as using GPUs or tapping into the cloud-based Succinct Prover Network.

On the other hand, when proofs increase in size, the STARK generation component of this will take increasingly longer, and eventually become the main source of latency. In that situation, some steps we can take to reduce prover runtime include pruning `nanoda_lib` further, removing support for certain parts of Lean proofs, such as support for quotient types which zkPi also did not include [8]. Another small optimization we could implement is to use SP1's more efficient SHA-256 precompile, as this has been specifically optimized for hashing in as few cycles as possible.

8 Individual contributions

- Anthony implemented the μLean type checker in Lean, designed the μLean vernacular, and formalized several math proofs in μLean.
- Dan helped with improving and optimizing the μLean implementation, wrote the proof of consistency of one of the earlier versions of μLean, implemented a bytecode-based type checker to try to make the $\sqrt{2} \notin \mathbb{Q}$ proof run in Lurk.
- Rey designed and implemented the protocol and proof system in Lurk to achieve cryptographic guarantees for μLean.
- Zach implemented and tested the zkVM, and built the add-ons to support cryptographic soundness.

References

[1] Nada Amin et al. *LURK: Lambda, the Ultimate Recursive Knowledge*. Cryptology ePrint Archive, Paper 2023/369. 2023. DOI: 10.1145/3607839. URL: https://eprint.iacr.org/2023/369.

[2] ammkrn. *nanoda_lib: Library implementing type inference/checking functionality based on the Lean theorem prover*. Version 0.3.2. Sept. 17, 2025. URL: https://github.com/ammkrn/nanoda_lib (visited on 05/11/2026).

[3] Nicolaas Govert de Bruijn. "Lambda Calculus Notation with Nameless Dummies". In: *Indagationes Mathematicae* 34 (1972), pp. 381–392.

[4] Thierry Coquand and Gérard Huet. "The Calculus of Constructions". In: *Information and Computation* 76.2-3 (1988), pp. 95–120.

[5] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Paper 2019/953. 2019. URL: https://eprint.iacr.org/2019/953.

[6] Google Quantum AI et al. *Securing Elliptic Curve Cryptocurrencies against Quantum Threats*. Accessed: 2026-05-12. Mar. 2026. URL: https://quantumai.google/static/site-assets/downloads/cryptocurrency-whitepaper.pdf.

[7] Jens Groth. "On the Size of Pairing-based Non-interactive Arguments". In: *EUROCRYPT 2016*. Springer, 2016.

[8] Evan Laufer, Alex Ozdemir, and Dan Boneh. *zkPi: Proving Lean Theorems in Zero-Knowledge*. Cryptology ePrint Archive, Paper 2024/267. 2024. DOI: 10.1145/3658644.3670322. URL: https://eprint.iacr.org/2024/267.

[9] Lean FRO. *lean4export: Plain-text declaration export for Lean 4*. 2025. URL: https://github.com/leanprover/lean4export (visited on 05/11/2026).

[10] Leonardo de Moura et al. "The Lean Theorem Prover (System Description)". In: *Automated Deduction – CADE-25*. Springer, 2015.

[11] Keegan Ryan. *We beat Google's Zero-knowledge proof of quantum cryptanalysis*. The Trail of Bits Blog. Accessed: 2026-05-12. Apr. 2026. URL: https://blog.trailofbits.com/2026/04/17/we-beat-googles-zero-knowledge-proof-of-quantum-cryptanalysis/.

[12] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006.

[13] Succinct Labs. *SP1: A zero-knowledge virtual machine that proves the correct execution of programs compiled for the RISC-V architecture*. Version 6.0.2. Feb. 26, 2026. URL: `https://github.com/succinctlabs/sp1` (visited on 05/11/2026).