Lecture 14: Lean: A Different Take on Verification

Lecturer: Cameron Freer **Date:** March 31, 2026
Scribe: Inimai Subramanian

1. Overview

This lecture introduces *Lean*, a modern proof assistant and programming language, through the lens of this course's central themes of verification and certificates. We situate Lean in the broader landscape of formal verification, motivate why mathematicians and computer scientists are increasingly formalizing their work, and examine how Lean's design—a *small trusted kernel* checking *untrusted proof terms*—makes it a natural fit for AI-assisted mathematical reasoning.

Lean is notable in that it is both:

- A **proof assistant**, for constructing and verifying formal proofs.
- A **programming language**, where code, specifications, and proofs coexist in a unified framework.

This dual nature allows developers and mathematicians to write code, specifications, and proofs within the same framework, eliminating the "translation layer" typically found between software design and formal verification.

The lecture proceeds as follows. We begin with the analogy between cryptographic certificates and Lean proof terms (Section 2). We then survey the reasons to formalize mathematics and software (Section 4). We describe Lean's type-theoretic foundations and its giant library Mathlib (Section 6). We walk through concrete code examples, including a one-time-pad round-trip theorem (Section 7). We survey the trusted computing base and a crypto-style threat model for proof checking (Section 8). We cover frontier formalizations and Lean for cryptography and CS (Section 9). Finally, we discuss AI-assisted proof workflows using `lean-lsp-mcp` and `lean4-skills` (Section 11).

2. The Crypto Analogy: Proof Terms as Certificates

A recurring theme in 6.S976/18.S996 is the idea of a *certificate*: a short, efficiently checkable object that witnesses some claim. In cryptography, many protocols produce certificates that a verifier checks. Lean fits squarely into this paradigm.

Core Analogy

In cryptography, a **prover** produces a *certificate* that a *verifier* checks. In Lean, a **proof search process** (human, tactic, or AI) produces a *proof term* that a small *kernel* checks. The design goal is identical: a **small trusted checker** facing potentially **untrusted search**.

Remark 2.1. The analogy is not merely aesthetic. It has direct implications for AI-assisted mathematics: because the checker is small and independent, one can allow the search process (an LLM or an agentic system) to be arbitrarily complex and unreliable, while retaining high confidence in the correctness of whatever it produces, as long as the kernel accepts the resulting proof term [3].

3. What Is Lean?

Lean is simultaneously a **programming language** and a **proof assistant**. The same syntax is used for code, specifications, and proofs; there is no translation layer between them. Proofs are first-class objects in the language.

Key structural features:

- **Tactics and automation** are programs that build proof terms. A one-line `simp` (simplification) call, for example, invokes an implicit search procedure; if it succeeds, the result is a fully valid proof term that the kernel accepts.
- **The kernel** is the only component that must be trusted. It checks that a given term has the claimed type. Everything else—tactics, automation, LLM-generated code—is "untrusted search."
- Lean is **implemented in Lean itself** and compiles to native code.
- **Mathlib**, the community mathematics library, contains over 266,000 theorems, 127,000 definitions, and contributions from 772 people (discussed further in Section 5). It is the primary substrate for formalization in Lean today [10].

4. Why Formalize?

Formalization offers several distinct benefits, each relevant to different audiences.

4.1. Confidence

Formal proofs provide an extremely high level of confidence in correctness. This is especially valuable for long, fragile proofs, or proofs that are difficult to referee carefully in the traditional way (e.g., proofs that rely on heavy case analyses or numerical computations).

4.2. Refactoring and Maintainability

Formalization allows proofs to be maintainable artifacts. For example, Terence Tao has described how the improvement of a constant in a bound (say, from 12 to 11), requires only changing that

constant to update the formal proof; the compiler then pinpoints exactly which downstream lemmas need to be re-examined [9].

4.3. Compositionality

These systems are a realization of the idea of "buying someone's part off the shelf" [4]. Once a result has been formalized and merged into Mathlib or the appropriate library, any subsequent user can invoke it with full confidence, even without fully understanding its proof, because Lean enforces that it is being applied correctly. This is compositionality at the mathematical level.

Remark 4.1. Compositionality has one important caveat: Lean can verify that a proof term correctly establishes a stated proposition, but it cannot verify that the stated proposition is the one you *meant* to prove. Specification risk—the risk that the formal statement diverges from mathematical intent—remains the responsibility of the human. More about specification risk can be found in Section 8.2.

4.4. Verification Stronger Than Testing

For software and cryptographic protocols, formal verification offers guarantees strictly stronger than testing. Wrong assumptions surface as proof obligations that cannot be discharged, rather than as bugs found (or not found) at runtime. The Cedar authorization system at AWS exemplifies this: the Lean model and proofs found bugs that testing did not.

4.5. Exploration

Making every step explicit often reveals structure that is invisible on paper. Side conditions a mathematician would wave through informally become proof obligations, and discharging them can expose hidden assumptions or suggest stronger formulations. AI systems backed by formalization can assist humans in exploring a mathematical landscape more deeply and with more confidence.

4.6. Stronger Results

Formalization can force one to carry explicit bounds through arguments that would informally be stated as existence proofs. This can yield constructive versions of existence theorems and rigorous numerical bounds via validated numerics.

4.7. Collaboration at Scale

Precise formal interfaces allow many people—or AI agents—to work on different components of a large project simultaneously. Mathlib's 772 contributors and the blueprint-driven style of large projects (discussed in Section 9) exemplify this.

4.8. Teaching

Formalization enables executable textbooks and interactive explanations in which every claim is machine-checkable. An AI tutor backed by a formal library can teach with verified claims rather

than plausible ones.

4.9. Consolidation of Prior Work

Formalization is a natural vehicle for revisiting and organizing existing results — extracting modular lemmas, clarifying interfaces between theorems, and turning a backlog of published-but-scattered work into a coherent, research-enabling library.

4.10. Impact on Scientific Literature

Formal libraries enable semantic search over mathematical knowledge, "living papers" that update as dependencies evolve in Mathlib, and a shift in peer review from tedious line-checking to evaluating ideas.

4.11. Interaction with AI

Formal proof systems provide AI with an unambiguous success criterion: the kernel either accepts or rejects the proof term. This is qualitatively different from the vague feedback available in informal mathematical reasoning. The interaction between LLMs and Lean is discussed further in Section 11.

5. Mathlib as a Substrate

Mathematics has deep dependencies: even a "simple" result may require substantial infrastructure from analysis, algebra, or topology. Mathlib provides this infrastructure. Its current scale reflects years of coordinated community effort:

Fact 5.1. *As of the time of this lecture, Mathlib contains 266,437 theorems and 127,144 definitions, contributed by 772 people whose expertise spans mathematics, computer science, and software engineering [10].*

Mathlib's coverage is broad: analysis, algebra, topology, measure theory, number theory, combinatorics, and more. Despite its size, it remains incomplete enough that frontier formalization projects routinely find themselves building infrastructure before they can even state their target theorem. This incompleteness is a practical constraint every practitioner encounters.

A distinctive feature of the Lean ecosystem is that contributions **flow back**: when a project builds missing infrastructure, that infrastructure is pushed upstream into Mathlib, making it available to every future user. Each project thus leaves Mathlib in a better state than it found it.

Remark 5.2. A missing dependency can be more work to formalize than the target theorem itself, and that dependency may in turn have missing dependencies. Much of the work in frontier formalizations is **library engineering**—building the missing pieces of Mathlib—rather than directly formalizing the target result. Each such project pushes its prerequisites upstream, gradually expanding the library for future users.

6. How Lean Works: Types, Terms, and Dependent Types

6.1. Dependent Type Theory

Lean is characterized by a very strong type system, a version of **dependent type theory**. The key definition is:

Definition 6.1 (Propositions as Types, Proofs as Terms)**.** Every proposition is of type `Prop`. To prove a proposition P, one constructs a term t of type P (i.e. `t : P`). The kernel checks that t indeed has type P.

Remark 6.1. Throughout Lean source code, the notation `(x : A)` should be read as "x is a term of type A." This uniform syntax covers both ordinary programming values (e.g., `(n : Nat)`) and proof objects (e.g., `(h : P)`, meaning h is a proof of proposition P).

The primitive construct is the *dependent function type*: given a type A and a family of types $B(x)$ indexed by $x : A$, the type $(x : A) \to B(x)$ is the type of functions mapping each $a : A$ to a term of type $B(a)$.

This entails three special cases:

1. **Ordinary function type.** If B does not depend on x, then $(x : A) \to B$ is just the ordinary function type $A \to B$.
2. **Logical implication.** If A and B are both propositions (i.e., A : `Prop`, B : `Prop`), then $A \to B$ is a logical implication: a proof of $A \to B$ is a function taking proofs of A to proofs of B.
3. **Universal quantification.** If $B\ x$ (B applied to x) is a proposition for each x, then $\forall x : A,\ B\ x$ is a universal quantification. For all x in which A holds, B also holds.

Thus, there is a single unified syntax for programs, specifications, and proofs [6].

6.2. Dependent Types

Lean's **dependent types** allow one to encode *invariants* directly in the type of a value, turning runtime errors into compile-time type errors.

Some examples include:

Example 6.2 (Length-indexed vectors)**.** `Vector α n` is the type of lists of length exactly n with elements of type α. Appending a `Vector α m` to a `Vector α n` yields a `Vector α (n + m)`; the lengths are tracked automatically in the type.

Example 6.3 (Bounded natural numbers)**.** `Fin n` is the type of natural numbers strictly less than n. Attempting to use an index $k \geq n$ is a *type error* caught at compile time, not a runtime array-bounds crash.

Example 6.4 (Sorted lists with lower bounds)**.** One can define a type `SortedList lo` of sorted lists whose smallest element is at least `lo`. The sortedness condition is part of the type; constructing a `SortedList` requires providing a proof of the ordering condition at each step.

The relevant Lean definition is:

```
- A sorted list carries its lower bound in the type
inductive SortedList : Nat → Type where
| nil (lo : Nat) : SortedList lo
| cons (x : Nat) lo : Nat (h : x ≤ lo) (tail : SortedList lo)
: SortedList x

- [1, 3, 5] with proof that each element ≤ the next
example : SortedList 1 :=
.cons 1 (by omega) (.cons 3 (by omega) (.nil 5))
```

Remark

Side conditions that appear in prose (e.g. "the list must be sorted") become part of the type and are checked **automatically** by the kernel. In these examples, correctness conditions (e.g., bounds, ordering) are enforced statically by the **type** system rather than dynamically at runtime.

6.3. The Three Standard Axioms

Lean's dependent type theory is nearly constructive by default, but three additional axioms are adopted to support the full breadth of mathematical practice. Running `#print axioms thmName` on any theorem reports exactly which of these it depends on; a clean proof depends on only the three below and nothing else.

Definition 6.2 (The Standard Three Axioms)**.** Lean's foundational axioms are:

1. **Propositional extensionality (`propext`):** Two propositions P and Q that are logically equivalent, each implying the other, are declared equal as types: P = Q. Without this axiom, logically equivalent propositions would be provably equivalent but not substitutable in all contexts.

2. **Quotient soundness (`Quot.sound`):** A quotient type $A/\sim$ is the type obtained from a type A by declaring two elements `a, b : A` equal whenever `a ~ b` under some equivalence relation $\sim$. Intuitively, the quotient "collapses" all $\sim$-equivalent elements into a single representative class. As a concrete example, the integers $\mathbb{Z}$ can be constructed as the quotient of pairs of natural numbers `(a, b)` under `(a, b) ~ (c, d)` $\iff$ `a + d = b + c` (encoding `a - b = c - d`). The axiom `Quot.sound` asserts: if `a ~ b`, then a and b are equal in the quotient. This axiom is used pervasively in Mathlib to construct the integers, rationals, real numbers, and other standard algebraic objects.

3. **Choice (`Classical.choice`):** Given a type α together with a proof that α is nonempty, `Classical.choice` produces some element of α without specifying which (that is, every nonempty type has an element). This makes Lean's logic *classical*: one can prove existence without exhibiting a witness.

Remark 6.5. Classical logic (via `Classical.choice`) is used throughout Mathlib. Constructive subsets of Lean exist, but most formalized mathematics relies on classical logic.

Remark 6.6 (Incompleteness and search). Lean's logic is *incomplete* in the proof-search sense: the existence of a proof and the ability to *find* one are entirely separate questions. Lean can represent any problem expressible in dependent type theory — if a problem cannot be stated in Lean, it is likely not idiomatic for the system — but representability gives no guarantee that automation will find a proof.

7. Code Examples: One-Time Pads and Proof Styles

7.1. Defining XOR and Encryption

We define the XOR function on Booleans `def bxor` by pattern matching, then define a 1-bit encryption function. Lean checks that all cases are covered.

Definition 7.1 (Boolean XOR and a sample encryption function).

```
- XOR on booleans, by pattern-matching.  same → false, different → true.
def bxor : Bool → Bool → Bool
| true, true => false
| false, false => false
| _,     _     => true

- Encrypt a 1-bit message with a 1-bit key.
def enc (m k : Bool) : Bool := bxor m k
```

These are *executable definitions*, not axioms. Lean can compute with them and checks that all cases are covered in the pattern match; omitting a case would be a compile error.

7.2. Propositions vs. Computations

The functions `bxor` and `enc` return a `Bool` and can be evaluated. Lean can also express *propositions*. For example:

```
def odd (n : Nat) : Prop := ∃ k, n = 2 * k + 1
```

The type of `odd 3` is `Prop`. This is a statement that can be true or false; it cannot be *evaluated* (reduced to `true` or `false`) the way `bxor true false` can. But it can be *proved* by providing a witness:

```
example : odd 3 := <1, by decide>
```

The witness $k = 1$ and the proof that $3 = 2 \cdot 1 + 1$ together form a term of type `odd 3`.

7.3. The OTP Round-Trip Theorem

We can view the process of proving a theorem in Lean as a game: you are given *goals* and *hypotheses*, and subsequently apply moves (*tactics*). Each tactic transforms the goal until nothing remains.

We look at one such example here:

Theorem 7.1 (One-Time-Pad Round-Trip)**.** *For all*

```
(m k : Bool) : enc (enc m k) k = m
```

That is, encrypting a message twice with the same key recovers the original message.

Proof. We prove this theorem in Lean, using the tactic `cases`:

```
theorem otp_roundtrip (m k : Bool) : enc (enc m k) k = m := by
cases m <;> cases k <;> rfl
```

The tactic `cases m` splits the goal into the cases `m = true` and `m = false`. The combinator `<;>` applies the next tactic to all remaining goals simultaneously. After splitting on both `m` and `k`, all four resulting goals reduce by computation (`rfl`), since both sides evaluate to the same value. The kernel checks the resulting proof term. □

Running `#print axioms otp_roundtrip` reveals that the theorem depends only on `propext` (propositional extensionality), one of the three standard axioms. No `sorry` or `Classical.choice` is present, and the trust base is fully visible.

7.4. Three Proof Styles

Lean supports three surface styles for writing proofs; all produce the same proof term in the kernel. Lean does not distinguish between them once elaboration is done.

Example 7.2. For `variable {p q : Prop}`:

1. **Term style.** The proof object is constructed directly:

   ```
   theorem and_swap_term (h : p ∧ q) : q ∧ p :=
   And.intro h.right h.left
   ```

2. **Tactic style.** Lean is told how to build the proof interactively:

   ```
   theorem and_swap_tac (h : p ∧ q) : q ∧ p := by
   exact ⟨h.right, h.left⟩
   ```

3. **Declarative style.** The structure of the proof is made explicit:

   ```
   theorem and_swap_decl (h : p ∧ q) : q ∧ p := by
   constructor
   · exact h.right
   · exact h.left
   ```

All three methods produce the same proof term.

Procedural style (matching the structure of a recursive program) is also available (see Section 7.6 for an example), and is often the most natural for inductive proofs. Tactic style is typically fastest to write. Declarative and term styles can be easier to read back later.

7.5. Dependent Types and n-bit OTPs

The 1-bit example extends naturally to n-bit one-time pads using dependent types:

```
def OTP (n : Nat) := Fin n → Bool

def encVec {n : Nat} (m k : OTP n) : OTP n :=
fun i => bxor (m i) (k i)

example {n : Nat} (m k : OTP n) (i : Fin n) :
encVec (encVec m k) k i = m i := by
simp only [encVec]
cases m i <;> cases k i <;> rfl
```

Remark 7.3. Here `n` appears **inside** the type. The type `OTP n` represents an n-bit pad as a function from `Fin n → Bool`. The length is part of the type: the checker prevents mixing pads of incompatible lengths **before any proof begins**.

7.6. Deny-Overrides

To make the connection between Lean's abstractions and real software concrete, consider a minimal authorization policy evaluator. Policies return either `allow` or `deny`; the deny-overrides semantics says that a single `deny` anywhere in the policy list overrides all `allow` decisions.

Definition 7.2 (Deny-overrides)**.** Formally we define this as:

```
inductive Decision where
  | allow | deny

def authorize : List Decision → Decision
  | []           => .deny
  | .deny  :: _  => .deny
  | .allow :: rs => authorize rs
```

The function is defined by structural recursion on the list. We can now state and prove a key safety property: inserting a `deny` anywhere in a policy list always produces a `deny` outcome, regardless of what precedes or follows it.

Theorem 7.4. *For any policy lists `pre` and `suf`, `authorize (pre ++ [deny] ++ suf) = deny`.*

```
theorem once_denied_always_denied
    (pre suf : List Decision) :
    authorize (pre ++ (.deny :: suf)) = .deny := by
  induction pre with
  | nil => rfl
  | cons d ds ih =>
      cases d <;> simp [authorize, ih]
```

Proof. By induction on `pre`. The base case reduces immediately by computation: the list begins with `deny`, so `authorize` returns `deny` at once. In the inductive step, the head `d` of `pre` is split by `cases d`: if `d = deny`, `authorize` returns `deny` immediately; if `d = allow`, `authorize` recurses on the tail and the inductive hypothesis applies. `simp` discharges both subcases. □

Remark 7.5. This is toy code, but the structure — inductive types for policy decisions, recursive evaluation, inductive proofs over list structure — is precisely the structure used in Cedar's production Lean model at AWS. The same technique scales: define the evaluator as a recursive function over an inductive type, state safety properties as propositions, and discharge them with the kernel. The four bugs Cedar's Lean model found that testing missed were found by exactly this kind of proof obligation failing to discharge.

In addition, we can use the three proof methods mentioned in Section 7.4 to prove the authorization theorem:

Example 7.6. Authorization theorem proofs:

1. **Procedural style.** Looks like a recursive program:

```
theorem once_denied_proc :
∀ pre suf :  List Decision,
      authorize (pre ++ (.deny :: suf)) = .deny
  | [], _ => rfl
  | .deny :: _, _ => rfl
  | .allow :: ds, suf => by
      simpa [authorize] using once_denied_proc ds suf
```

2. **Tactic style.** Delegates work to induction and case splitting:

```
theorem once_denied_tac
    (pre suf : List Decision) :
    authorize (pre ++ (.deny :: suf)) = .deny := by
  induction pre with
  | nil => rfl
  | cons d ds ih =>
      cases d <;> simp [authorize, ih]
```

3. **Declarative style.** The structure of the proof is made explicit:

```
theorem once_denied_decl
    (pre suf : List Decision) :
    authorize (pre ++ (.deny :: suf)) = .deny := by
  induction pre with
  | nil => rfl
  | cons d ds ih =>
      cases d with
      | deny => rfl
```

```
  | allow =>
      calc
        authorize ((.allow :: ds) ++ (.deny :: suf))
            = authorize (ds ++ (.deny :: suf)) := by rfl
        _ = .deny := ih
```

Al three styles elaborate to proof terms in the same dependent type theory.

8. What Is Trusted? A Crypto-Style Threat Model

8.1. The Trusted Computing Base

Lean's trusted computing base (TCB) is intentionally minimal. The following list complementary ways to check a Lean proof:

- **Reference kernel (C++):** approximately 8,000 lines. This is the implementation Lean itself uses to check proof terms.
- **Nanoda (Rust):** an independently implemented external checker. Running Nanoda on a proof export provides independent confirmation.
- **Lean4Lean:** a complete Lean 4 type-checker written in Lean itself. Having the checker in the same language as the proofs allows it to be verified by Lean's own reasoning.
- `leanchecker`: replays built declarations through the kernel in a fresh process, guarding against `.olean` state tricks.
- `comparator`: uses a trusted challenge file with a sandboxed build and external validation. Effective against certain kinds of kernel hacking, including LLM-generated code that is optimized merely to compile rather than to be correct.

This is analogous to having multiple independent implementations of a cryptographic protocol: independent agreement provides much stronger assurance than any single implementation.

8.2. Specification Risk

All of the checks above address the question: does the proof term match the stated theorem? There is a complementary question that no kernel can answer: does the stated theorem match what you actually meant (i.e. the **specification risk**)? Lean verifies proof terms against theorem statements. It has no access to mathematical intent. The following failure modes all live entirely outside the kernel's reach:

- Wrong object. Did I define the right mathematical structure? A subtly incorrect definition may admit a valid proof of the stated theorem while being meaningless or misleading for the intended application.
- Wrong statement. Did I state the theorem I intended? A statement can be formally provable yet subtly different from the desired result.

- Forgotten hypothesis. Did I accidentally omit a side condition? The theorem may be true only in a special case, but the formal statement is more general than intended.
- Totalized APIs hiding boundary cases. Lean's standard library often totalizes partial functions — defining them to return a default value (typically 0) where they would otherwise be undefined. A theorem stated using a totalized API may appear to have no hypotheses while silently sweeping boundary cases under the rug.

Example 8.1 (Totalized vs. explicit derivative)**.** We define the lemmas:

```
- totalized:  deriv = 0 when undefined
#check Real.deriv_rpow_const
- (x p :  ℝ) :  deriv (· ^ p) x = p * x ^ (p-1)

- explicit hypotheses
#check Real.hasDerivAt_rpow_const
- x p :  ℝ (h :  x ≠ 0 ∨ 1 ≤ p) :
- HasDerivAt (· ^ p) (p * x ^ (p-1)) x
```

The Mathlib lemma `Real.deriv_rpow_const` states $\partial_x(x^p) = p \cdot x^{p-1}$ without hypotheses, because Lean's `deriv` is totalized (defined to be 0 when the derivative does not exist). The corresponding `HasDerivAt` lemma includes the hypothesis $x \neq 0 \vee 1 \leq p$. Using the totalized version when the hypotheses are not satisfied is formally valid but mathematically misleading.

To bridge this gap, some mitigation strategies include: (1) use explicit predicates (`HasDerivAt`, `Continuous`, `Measurable`) over totalized APIs, (2) review definitions and statements separately from proofs, (3) triangulate with sanity-check lemmas, and (4) have someone other than the author read the formal statement before investing in a proof.

8.3. Threat Model

Different threat levels call for different checks, analogous to escalating from honest to malicious adversary models in this course.:

- **`.olean` or core state bugs:** Forgotten hypotheses, `.olean` state tricks, or core state bugs. Mitigated by `leanchecker --fresh` and re-running the build.
- **Hidden `sorry`:** Use `#print axioms thmName`. The output should list only the three standard axioms (`propext`, `Quot.sound`, `Classical.choice`). Any `sorry` will appear explicitly.
- **Custom or exotic axioms:** Also visible via `#print axioms`.
- **Proofs by native evaluation:** `Lean.trustCompiler` (which appears when using `native_decide`) enlarges the trusted computing base (TCB) and should be noted in the axiom list.
- **Actively malicious proof submissions:** Use `comparator` with external validation (e.g. trusted challenge file, sandboxed build).

Remark 8.2. AI-generated proofs present a subtly different threat from adversarial proofs: an LLM may not be explicitly trying to deceive, but will find *workarounds* to produce compiling Lean code even when those workarounds are unsound (e.g., inserting `sorry`, or exploiting native evaluation). The same checks apply regardless.

8.4. Proofs by Native Evaluation: `decide` vs. `native_decide`

Lean provides two decision procedures for decidable propositions:

- `decide`: runs a decision procedure entirely within Lean's kernel. Slower (often much slower for large instances), but entirely within the trusted kernel.
- `native_decide`: uses Lean's native code compiler to run the decision procedure, which can be 10,000× faster. However, it introduces `Lean.trustCompiler` as an axiom, extending the TCB to include the compiler.

For most purposes `decide` is preferred. `native_decide` is appropriate when the computation is too large to run inside the kernel and the performance benefit is necessary [6].

9. Frontier Formalizations

9.1. Blueprint-Driven Development

Large formalization projects have converged on a *blueprint* methodology: before writing any Lean code, the team produces a detailed semi-formal document that makes every lemma explicit, traces dependencies, and divides the work into manageable contributions. The blueprint serves as a coordination layer for many contributors working in parallel.

9.2. Exchangeability and de Finetti's Theorem

A formalization of the de Finetti–Ryll-Nardzewski theorem (every exchangeable sequence on a standard Borel space is conditionally i.i.d.) was completed with three distinct proof routes: the reverse martingale argument, L^2 contractability, and the Koopman/mean ergodic approach. The project required over 43,000 lines of Lean across 100+ files, because the necessary measure-theoretic infrastructure was missing from Mathlib. The majority of the effort went into building and upstreaming that infrastructure.

Definition 9.1. Exchangeability refers to the invariance of a random sequence under finite permutations of coordinates.

9.3. Sphere Packing in Dimension 8

Viazovska (Fields Medal, 2022) proved that the E_8 lattice achieves the densest sphere packing in dimension 8. A Lean formalization of this result was carried out using the blueprint approach. Notably, all of Cameron's contributions were AI-assisted (using Claude, Codex, `lean4-skills`, and `lean-lsp-mcp`). One AI-assisted contribution found and corrected a missing E_4 factor in

the blueprint statements. A sorry-free AI-generated pull request of over 50,000 lines was merged in February 2026.

9.4. The Noperthedron

The Noperthedron (Steininger & Yurkevich, 2025) [8] is a convex polyhedron that disproves the conjecture that all convex polyhedra have *Rupert's property* (the ability to pass a larger copy of itself through a hole cut in the original). The proof partitions a 5-dimensional parameter space into approximately 18.7 million regions and checks each locally.

Remark 9.1. The Lean formalization of the Noperthedron has fully formalized the continuous-mathematics component. The computational step (the certificate for the 18.7 million region partition) is the remaining `sorry`. This illustrates the distinction between the mathematical and computational parts of a hybrid proof.

10. Lean for CS, Cryptography, and Security

10.1. Complexity Theory

Lean currently lags behind Rocq and Isabelle for CS verification and complexity theory. A notable recent contribution [7] verified coNP, Σ_2^P, PP, and PSPACE completeness results in approximately 24,000 lines of Lean. The **CSLib** project aims to build a comprehensive CS substrate for Lean analogous to Mathlib for mathematics.

10.2. Cryptography

Several connections between Lean and cryptographic proofs have emerged:

- **SNARK verification:** Groth16 soundness was verified in Lean [2]. Cairo/STARK algebraic verification has also been formalized.
- **ZK proofs from Lean theorems:** The zkPi system [5] compiles Lean theorems to zero-knowledge proofs. The Ix project (pre-alpha) is pursuing an even more ambitious goal: fitting all of Mathlib into a proof of approximately 1 KB, checkable in under 100ms.
- **Cedar (AWS):** The Cedar authorization language [1] has an executable Lean model (1,700 LOC) that runs alongside the production Rust implementation (15,700 LOC), with 5,700 LOC of Lean proofs and 100 million nightly differential tests. The Lean model found 4 bugs that testing missed. No new Cedar version ships unless the Lean model, proofs, and tests all pass.
- **Aeneas/SymCrypt (Microsoft):** The Aeneas framework translates Rust programs into Lean functional models for verified cryptographic implementations.

Remark 10.1 (Convergence of Formal Proofs and Cryptographic Certificates). Formal proof objects and cryptographic certificates are converging. A Lean proof term certifies mathematical correctness; a zero-knowledge proof certifies it without revealing the proof itself. Systems like zkPi bridge the two.

11. AI-Assisted Lean: `lean-lsp-mcp` and `lean4-skills`

11.1. `lean-lsp-mcp`: Giving Agents the Same Tools as Users

The `lean-lsp-mcp` package exposes Lean's Language Server Protocol (LSP) to AI agents via the Model Context Protocol (MCP). This gives an agent access to the same interactive tools a human user has: InfoView (goal state), hover documentation, code completion, and Mathlib search.

The available tools fall into four categories:

1. **Diagnostics and goals:** `diagnostic_messages`, `goal`, `term_goal`.
2. **Navigation and information:** `hover_info`, `completions`, `declaration_file`, `file_outline`.
3. **Code execution and verification:** `build`, `run_code`, `verify`, `multi_attempt`.
4. **Search:** `local_search`, `leansearch`, `loogle`, `leanfinder`, `state_search`, `hammer_premise`.

11.2. `lean4-skills`: Reliable Agentic Workflows

The `lean4-skills` package provides structured workflows for agentic Lean coding, organized around commands like `/prove`, `/formalize`, `/learn`, and `/checkpoint`. The goals are:

- **Efficiency:** Reliable workflows prevent agents from repeating failed approaches.
- **Guardrails:** Known antipatterns (e.g., gratuitous use of `sorry`, redundant axioms) are flagged explicitly.
- **Success criteria:** The agent has a clear notion of completion: the kernel accepts, no `sorry` is present, and the axiom list is clean.
- **Continual learning:** A community of users contributes patterns and antipatterns; insights from large projects (such as the exchangeability and sphere packing formalizations) are upstreamed into the skill.

11.3. `lean4-skills /learn`: Socratic Education

The `/learn` mode allows an AI agent to adapt to the user's learning profile. In Socratic mode, the agent asks questions calibrated to the user's level, uses `lean-lsp-mcp` search tools to discover relevant Mathlib definitions and lemmas, and guides the user toward a rigorous understanding of the concept.

Example 11.1 (Socratic learning of the `measurability` tactic)**.** A user asks to learn the `measurability` tactic. The agent first searches Mathlib for the tactic's definition (it is `aesop` with the `Measurable` rule set in `Mathlib.Tactic.Measurability`), then asks the user to state what measurability means before showing the definition. When the user over-

specializes to Borel sets, the agent corrects by pointing out that Lean's definition has no topological space, only a `MeasurableSpace`, nudging the user toward the general statement.

12. Summary

Key Takeaways

1. **Proof terms are certificates.** Lean fits the certificate/verifier paradigm central to this course. The design goal—small trusted checker facing untrusted search—is identical in Lean and in cryptographic proof systems.
2. **Search can be done by anyone.** Human mathematicians, tactics, and AI agents are all valid sources of proof search. The kernel's acceptance is the only standard that matters.
3. **Dependent types enforce invariants at compile time.** Length, sortedness, boundedness, and other side conditions can be incorporated directly into types, turning runtime errors into type errors.
4. **The trusted kernel is small and auditable.** Multiple independent implementations, external checkers, and `#print axioms` together provide a layered defense analogous to escalating adversary models in cryptography.
5. **Specification risk remains.** Lean verifies that a proof matches a statement; it does not verify that the statement matches intent. Human review of definitions and statements is essential.
6. **Mathlib is the critical substrate.** Much of frontier formalization is library engineering. Missing dependencies are often the binding constraint.
7. **AI + Lean is a natural fit.** The kernel's binary accept/reject provides unambiguous feedback for AI training and runtime verification.

References

[1] Amazon Web Services. Cedar: Verified authorization with Lean. https://lean-lang.org/use-cases/cedar/, 2026.

[2] Bolton Bailey and Andrew Miller. Formalizing soundness proofs of linear PCP SNARKs. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1489–1506, Philadelphia, PA, 2024. USENIX Association. ISBN 978-1-939133-44-1. https://www.usenix.org/conference/usenixsecurity24/presentation/bailey.

[3] Cameron Freer. Lean: a different take on verification. Guest lecture slides, MIT 6.S976/18.S996, https://cameronfreer.github.io/slides/202603-mit-lean/, March 2026.

[4] Alex Kontorovich. In math, rigor is vital. but are digitized proofs taking it too far? *Quanta Magazine*, March 2026.

[5] Evan Laufer, Alex Ozdemir, and Dan Boneh. zkPi: Proving Lean theorems in zero-knowledge.

In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*, pages 1–14. ACM, 2024. doi: 10.1145/3658644.3670322. Also available as IACR ePrint 2024/267. https://eprint.iacr.org/2024/267.

[6] Lean FRO. Lean 4 reference manual: Validating proofs. https://lean-lang.org/doc/reference/latest/ValidatingProofs/, 2026.

[7] Tristan Simas. Computational complexity of physical counting. arXiv preprint arXiv:2601.15571, January 2026. All results machine-checked in Lean 4 with no `sorry` placeholders. https://arxiv.org/abs/2601.15571.

[8] Jakob Steininger and Sergey Yurkevich. A convex polyhedron without Rupert's property. arXiv preprint arXiv:2508.18475, August 2025. v2: January 2026. https://arxiv.org/abs/2508.18475.

[9] Terence Tao. Machine-assisted proof. *Notices of the American Mathematical Society*, 72(1): 6–24, January 2025. doi: 10.1090/noti3041.

[10] The Mathlib Community. Mathlib: the Lean 4 mathematical library. https://leanprover-community.github.io/mathlib_stats.html, 2026. 266,437 theorems, 127,144 definitions, 772 contributors as of March 2026.