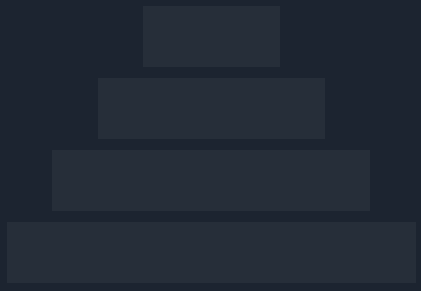
A HANDS-ON COURSE IN

Verifying Cryptography with Lean 4

From `1+1=2` to a machine-checked proof that real elliptic-curve code is correct.



A curriculum for the curious undergraduate —
no prior formal-verification or Lean experience assumed.

Companion to the `*-ed25519-verified` and `pasta-pallas-verified` proof projects.

Every code snippet in this book runs. Every claim it makes about a proof, a proof assistant has checked.

How to read this book

You are about to learn one of the most powerful ideas in computer science: how to make a computer *prove* that a program is correct — not test it on a few inputs and hope, but establish, with the certainty of mathematics, that it does the right thing on *every* input. We will aim that power at cryptography, where a single overlooked carry bit can quietly compromise every key a system ever generates.

This book assumes you can program a little and remember a little high-school algebra. It assumes **nothing** about formal methods, proof assistants, or Lean. We start from `1 + 1 = 2` and end at a real, published, machine-checked proof that the field arithmetic behind Ed25519 — the signature scheme in your SSH client, your phone, and half the internet — is correct.

- **The colored boxes each mean one thing.** A coral *big idea* box holds the load-bearing concept of a section. A grey *try it* box is an invitation to run something yourself. An amber *pitfall* box is a trap with its warning sign. A green *aha* box is an intuition meant to click. A framed *checkpoint* ends each chapter.
- **Do the exercises.** Reading a proof is like watching someone swim. You learn by getting in the water. Solutions are in the `solutions/` folder, but consult them only after a real attempt.
- **Everything runs.** The `exercises/` folder has Lean files you can open and check. When the book says "Lean accepts this," you can watch it happen.

⋆ Aha

The secret this book reveals: a proof is not a wall of Greek symbols meant to intimidate. A proof is a *program* — and a proof assistant is a very strict compiler for it. Once you see proofs as programs, the fear evaporates and the fun begins.

Contents

Chapter 1

Why Verify? The Bug That Testing Cannot Find

1.1 A story about one carry bit

In 2014, researchers examining widely deployed elliptic-curve code found arithmetic bugs of a very particular species: the code was correct on *almost every* input. Not most inputs — almost all of them, in a precise sense. One famous example, a carry-propagation flaw in an implementation of curve25519 arithmetic, produced a wrong answer with probability on the order of 2^{-64} per random input.

Pause on that number. If you tested this function a billion times per second, around the clock, you should expect to wait *centuries* before a random test happens to catch the bug. Every unit test passes. Every integration test passes. Fuzzers shrug. The code ships.

△ Pitfall

"It passed all the tests" means: it worked on the inputs we tried. For a 32-bit function there are four billion inputs and exhaustive testing is feasible. A field element in Ed25519 is 255 bits. The number of input *pairs* to a two-argument field operation is about 10^{153} — more than the square of the number of atoms in the observable universe. Testing samples a raindrop from that ocean.

Why does cryptographic code have bugs of exactly this shape? Because of how it must be written. To be fast and resistant to timing attacks, real implementations represent a 255-bit number in several machine-word *limbs* (we will spend happy hours with limbs in Chapter 9) and postpone expensive carry propagation as long as possible. The rare inputs where a deferred carry finally overflows are precisely the inputs no test generator stumbles on. The bug lives in the gap between "the arithmetic we meant" and "the arithmetic we wrote," and that gap is only visible on a set of inputs of measure nearly zero.

And in cryptography, "rare wrong answer" does not mean "rare small glitch." Wrong field arithmetic can leak private keys: several published attacks turn a single faulty group operation into full key recovery. The stakes are not a corrupted pixel; they are every signature your machine has ever made.

1.2 There is another way

What if, instead of sampling inputs, we could make a statement about *all* of them — and have a machine check that statement with the same rigor a compiler checks syntax?

∗ The big idea

A **formal proof of correctness** is a mathematical argument, written in a language precise enough for a computer to verify, that a program satisfies its specification on *every* input. Not sampled. Not probabilistic. Every input, forever, or the proof does not check.

The tool that checks such arguments is called a *proof assistant*. This book uses **Lean 4**, a modern proof assistant that is also a full-fledged programming language. Others you may have heard of: Rocq (formerly Coq), Isabelle/HOL, Agda. The ideas transfer; the syntax differs.

A proof assistant is built around a small, paranoid core called the *kernel*. Everything you will learn in this book — clever tactics, powerful automation, beautiful notation — is scaffolding whose only job is to produce a proof object the kernel accepts. The kernel is a few thousand lines of code that does one thing: check that each step of a proof follows from the previous ones by a fixed set of rules. If the kernel accepts, the theorem holds. If it does not, no amount of confidence, seniority, or good intentions makes the program correct.

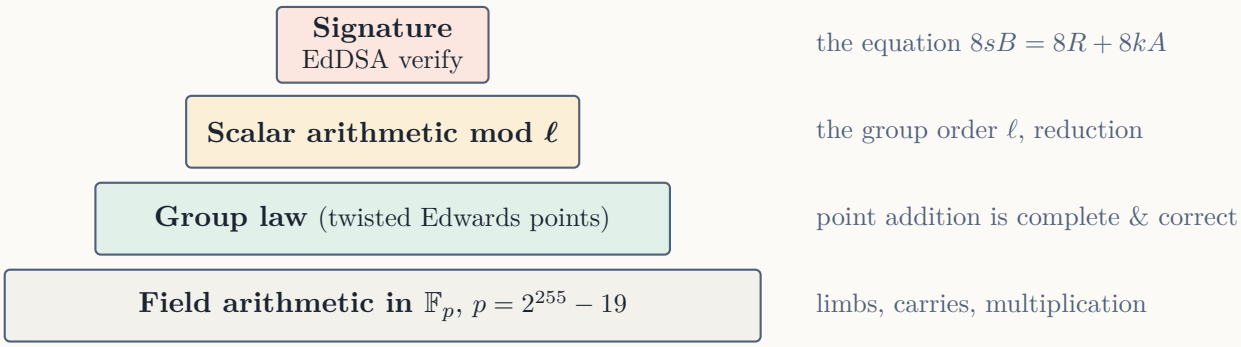
⋆ Aha

Here is the emotional core of formal verification, and it is worth internalizing early: **the proof assistant is not your examiner, it is your collaborator**. It never gets tired, never skips a case, never says "obviously." Every hour you spend arguing with it is an hour a bug did not survive. People who love proof assistants love them the way climbers love a good belayer.

1.3 What we will actually verify

This book is not a tour of toy examples. It is the curriculum companion to a set of real verification projects in which the arithmetic core of **Ed25519** — the elliptic-curve signature scheme used by SSH, Signal, TLS, and most cryptocurrency systems — was machine-checked in Lean 4, starting from the actual Rust source code of the `curve25519-dalek` library and several of its production forks.

The proofs are organized as a pyramid. Each layer states the correctness of one abstraction level and rests on the layer beneath it:



By the end of this book you will be able to read — and extend — the real proofs at every layer of this pyramid. The journey looks like this:

- **Chapters 2–5** teach Lean itself, from `#eval 1+1` to proofs by induction and the automation that dispatches arithmetic goals.
- **Chapters 6–7** build the mathematics: modular arithmetic, finite fields, and how to convince a paranoid kernel that a 77-digit number is prime.
- **Chapters 8–9** cross the bridge from Rust to Lean: how real code is translated into a form we can reason about, and the single most important idea in the whole enterprise — the *denotation function*.
- **Chapters 10–12** assemble the pyramid: field correctness, the ethics of axioms and honest boundaries, and the layers above.

1.4 Proofs versus tests: the honest comparison

Formal verification is not magic, and this book will never pretend otherwise. It is worth being precise, right now, about what a machine-checked proof does and does not give you.

Testing	Proving
Checks sampled inputs	Checks *all* inputs
Cheap to start, cheap to run	Expensive to write, cheap to re-check
Finds bugs	Establishes their absence (w.r.t. the spec)
Trusts nothing	Trusts the spec, the model, the kernel
Silent about *why* code is right	The proof *is* the why

That word *spec* in the right column is the fine print, and it matters enormously. A proof shows that code satisfies a specification. If the specification says the wrong thing — or says nothing, or is accidentally trivial — the proof is worthless no matter how green the checkmark. A recurring theme of this book (it gets its own chapter, Chapter 11) is how to read a verification claim skeptically: What exactly was proven? Against which model of the code? Resting on which axioms?

⋆ Aha

The most dangerous artifact in formal methods is not a wrong proof — the kernel prevents those. It is a *correct proof of the wrong statement*. Learning to smell those is as important as learning to write proofs at all.

1.5 Why Lean, and why now

Twenty years ago, verifying real cryptographic C or Rust code was a heroic, multi-year effort. Three things changed:

1. **Proof assistants matured.** Lean 4 is fast, pleasant, and comes with *Mathlib*, a library of over a million lines of formalized mathematics — finite fields and elliptic-curve ingredients included, so we do not start from bare axioms.
2. **Translation pipelines appeared.** Tools like *Charon* and *Aeneas* mechanically translate real Rust code into Lean definitions, so the thing we verify is derived from the code that ships, not a hand-transcribed approximation (Chapter 8).
3. **Automation got serious.** Decision procedures like `omega` (linear integer arithmetic) and `decide` discharge the boring 90% of goals, leaving humans the interesting 10%.

None of this made verification *easy*. It made verification *possible for a well-prepared person in finite time* — and preparing you is exactly what this book is for.

>_ Try it yourself

You do not need anything installed yet, but if you want to run code from Chapter 2 onward, install Lean now. One command:

```
curl https://elan.lean-lang.org/elan-init.sh -sSf | sh
```

Then open the `exercises/` folder of this repository in VS Code with the *Lean 4* extension. The orange progress bar you will see is the proof checker working through the file — your new collaborator saying hello.

Exercises

Exercise 1.1. A function takes two 255-bit inputs and is buggy on exactly one input pair. Assume you can test 10^9 random pairs per second. Estimate the expected time to find the bug by random testing, in multiples of the age of the universe ($\approx 4 \times 10^{17}$ seconds). You may approximate freely; the point is the order of magnitude.

Exercise 1.2. Give an example, from your own programming experience, of a bug that survived a test suite. What property would a specification have needed to state in order to exclude it?

Exercise 1.3. (Discussion) A colleague says: "Our crypto library is audited by three firms every year; formal verification is redundant." Name one class of defect audits are better at than proofs, and one class where proofs are strictly stronger.

⋈ Checkpoint

Before moving on, you should be able to explain to a friend: (1) why testing fundamentally cannot establish correctness of a 255-bit arithmetic function; (2) what a proof assistant's kernel is and why its small size matters; (3) what a proof of correctness actually promises — and the role the specification plays in that promise.

Chapter 2

Meet Lean: A Language Where Programs and Proofs Live Together

2.1 First contact

Lean 4 is two things wearing one syntax: a programming language (fast, functional, compiled) and a proof assistant. You will learn both faces, but we start with the friendlier one. Open a new file `Scratch.lean` and type:

```
#eval 1 + 1           -- 2
#eval 2 ^ 255 - 19    -- a 77-digit number, instantly
#eval "hello".length -- 5
```

`#eval` runs an expression and prints the result right in your editor. Notice the second line: Lean's natural numbers are *arbitrary precision* by default. The number $2^{255} - 19$ — which will follow us through the whole book — is a perfectly ordinary value here, not an overflow.

`#check` asks a different question: not "what is the value?" but "what is the *type*?"

```
#check 1 + 1         -- 1 + 1 : Nat
#check "hello"       -- "hello" : String
#check (1 : Int) - 5 -- Int
```

∗ The big idea

In Lean, **every expression has a type**, and the type checker verifies every file before anything runs. This obsession with types is not bureaucracy — in Chapter 3 it will turn out to be the entire mechanism by which proofs work. Learn to read `e : T` as "e is of type T" — or, with a squint we will justify later, "e is *evidence* for T."

2.2 Definitions and functions

New names are introduced with `def`:

```
def p : Nat := 2 ^ 255 - 19

def double (n : Nat) : Nat := 2 * n

def isEven (n : Nat) : Bool := n % 2 == 0

#eval double 21      -- 42
#eval isEven p       -- false (p is odd, good: p is supposed to be prime!)
```

Three things to absorb from this snippet:

- Function application is written with a space: `double 21`, not `double(21)`. It looks strange for a week and then all other syntax looks noisy forever after.
- Every definition states its types: `double` takes a `Nat` and returns a `Nat`. Lean can often infer types, but in this book we write them — specifications are the whole game, and a type is a tiny specification.
- Definitions are *immutable equations*, not instructions. There is no "assignment." `p` *is* $2^{255} - 19$, forever.

Functions of several arguments simply take them in sequence, and functions are values you can pass around:

```
def addMul (a b c : Nat) : Nat := a + b * c

def twice (f : Nat -> Nat) (x : Nat) : Nat := f (f x)

#eval twice double 10   -- 40
```

The type of `twice` is worth staring at: `(Nat -> Nat) -> Nat -> Nat`. Arrows associate to the right, and a multi-argument function is really a chain of single-argument ones — this is called *currying*. Nothing about it needs memorizing now; it will become muscle memory.

2.3 Inductive types: building data from nothing

Where do types like `Nat` and `Bool` come from? They are not built-in magic. They are *inductive types* — data types defined by listing every way to construct a value. Here is `Bool`, exactly as the core library defines it:

```
inductive Bool where
  | false : Bool
  | true  : Bool
```

Read: "a `Bool` is either `false` or `true`, and there is no other way to make one." That closed-world clause is what makes case analysis — and later, proofs by cases — airtight.

Now the star of the show. The natural numbers, following Peano's 1889 idea:

```
inductive Nat where
  | zero : Nat               -- 0 exists
  | succ (n : Nat) : Nat     -- every number has a successor
```

Every natural number is `zero` wrapped in finitely many `succ`s: the number 3 *is* `succ (succ (succ zero))`. (Lean stores big numbers efficiently under the hood, but *reasons* about them through this two-case skeleton.) Functions on inductive types are defined by *pattern matching* — one equation per constructor:

```
def add : Nat -> Nat -> Nat
  | m, Nat.zero   => m
  | m, Nat.succ n => Nat.succ (add m n)
```

⋆ Aha

Look at what just happened. Addition — the operation at the bottom of every cryptosystem in this book — is not an axiom or a CPU instruction here. It is a *two-line recursive program*, and every arithmetic fact we will ever prove unwinds, ultimately, to these two equations. When Lean later claims $a + b = b + a$ for *all* numbers, it will be because the structure of this definition forces it, not because anyone tested it.

△ Pitfall

`Nat` subtraction *truncates*: `#eval (3 - 5 : Nat)` prints `0`, not -2, because natural numbers have nowhere to go below zero. This single fact causes a large fraction of all beginner proof failures — an identity like $a - b + b = a$ is simply *false* for `Nat`. When subtraction must mean subtraction, use `Int`, or carry a hypothesis $b \le a$. Real verification projects hit this constantly: machine arithmetic wraps, truncates, and overflows, and the proofs must say so honestly.

2.4 Structures: records with guarantees

The last data-building tool we need bundles several fields together:

```
structure Point where
  x : Int
  y : Int

def origin : Point := { x := 0, y := 0 }

#eval origin.x   -- 0
```

A preview of why this matters to us: the Rust type `struct FieldElement51(pub [u64; 5])` — five 64-bit limbs representing one element of $\mathbb{F}_p$ — will arrive in Lean (Chapter 8) as essentially a structure holding an array of five machine words. The verification question of this entire book is: *do operations on those five words faithfully implement arithmetic in $\mathbb{F}_p$?* Structures are how the data crosses the bridge.

2.5 Namespaces, Mathlib, and reading error messages

Real developments organize names in `namespace` blocks (`Nat.add`, `Point.x`) and import the mathematical library:

```
import Mathlib
open Nat

#check Nat.Prime          -- the primality predicate, ready-made
#check ZMod               -- integers mod n -- our Chapter 6 home
```

And a word of comfort about *error messages*. You will see many. Lean's are precise and honest, and the single most useful habit you can develop this week is: **read the expected/actual types in the message, slowly**. A message like

```
type mismatch: argument has type Int but is expected to have type Nat
```

is not the compiler being difficult; it is a specification violation caught at the cheapest possible moment. Verification is this same experience scaled up: the machine holds the line, and the line is exactly where you drew it.

>_ Try it yourself

Open `exercises/Ch02.lean`. It contains the definitions from this chapter with a few holes marked `sorry` (a placeholder Lean accepts with a loud warning). Replace each with a working definition and watch the warnings disappear. In particular: define `mul : Nat -> Nat -> Nat` by recursion, using `add` — the same bootstrapping order (add, then mul) the real field proofs follow.

Exercises

Exercise 2.1. Define `pow : Nat -> Nat -> Nat` (by recursion on the exponent) and check with `#eval` that `pow 2 10 = 1024`.

Exercise 2.2. Define `fib : Nat -> Nat`. Then evaluate `fib 32`. Notice the pause — naive recursion is exponential. (Lean's `#eval` is fast; your algorithm is slow. The distinction matters when we later care about *what* is being computed versus *how*.)

Exercise 2.3. Write a structure `Rational` with fields `num : Int` and `den : Nat`, and a function `Rational.add`. What property of `den` can your type *not* enforce yet? (Keep your answer; Chapter 3

gives you the tool to fix it.)

⋈ Checkpoint

You should now be able to: evaluate and type-check expressions with `#eval`/`#check`; define functions, including recursive ones over `Nat`; explain what an inductive type is and recite the two constructors of `Nat`; and state from memory what `Nat` subtraction does on $3 - 5$, and why that will matter.

Chapter 3

Propositions as Types: The Idea That Makes It All Work

3.1 A suspicious similarity

Here are two things that look unrelated. First, a function that converts a pair into something else:

```
def swap (pair : A x B) : B x A := (pair.2, pair.1)
```

Second, a fact of logic: *if A and B both hold, then B and A both hold.* To prove it, you would say: "suppose I have evidence for A and evidence for B; then I can produce evidence for B and evidence for A — just present the same two pieces in the other order."

That prose proof and that program are the *same object*. The function takes a pair of values and reorders it; the proof takes a pair of pieces of evidence and reorders it. This is not an analogy or a teaching trick. It is a theorem about logic and computation, discovered independently by logicians (Curry, Howard) and now the load-bearing wall of Lean:

∗ The big idea

The Curry–Howard correspondence. A proposition can be read as a type — the type of its proofs. A proof is then simply a *program* of that type. Checking a proof is type-checking a program. There is no separate "proof checker" bolted onto Lean: the type checker you met in Chapter 2 *is* the proof checker.

Logic	Programming	In Lean
proposition P	type	`P : Prop`
proof of P	value/program of that type	`h : P`
$P \to Q$ (implication)	function type	`P -> Q`
$P \wedge Q$ (and)	pair type	`P / Q`
$P \vee Q$ (or)	tagged union	`P Q`
$\neg P$ (not)	`P -> False`	`Not P`
"true"	type with one trivial value	`True`
"false"	*empty* type	`False`

Take a minute with the last row: `False` is a type with *no constructors* — no way to build a value. To prove a false statement you would have to produce an inhabitant of an empty type. That is why the system is sound: lies have no evidence, so lies do not type-check.

3.2 Proofs are programs: first proofs

Let us write actual proofs as actual programs. Implication is a function type, so proving "P implies P" means writing the identity function:

```
theorem p_implies_p (P : Prop) : P -> P :=
  fun h => h
```

Read `fun h => h` aloud as a proof: "assume P holds — call the evidence h; then P holds, by h." Every classical proof-writing phrase has a program shape:

You say in prose	You write in Lean
"Assume P; call it h"	`fun h => ...`
"By hypothesis h"	`h`
"Apply lemma f to fact h"	`f h`
"Both parts hold: ... and ..."	`And.intro pf1 pf2`
"From $h : P \wedge Q$, the first part"	`h.1`

The pair-swapping example, now as an official theorem:

```
theorem and_swap (P Q : Prop) : P /\ Q -> Q /\ P :=
  fun h => And.intro h.2 h.1
```

And transitivity of implication is exactly function composition:

```
theorem imp_trans (P Q R : Prop) : (P -> Q) -> (Q -> R) -> (P -> R) :=
  fun pq qr => fun p => qr (pq p)
```

⋆ Aha

If you have ever composed two functions, you have already done everything this proof does. The intimidating part of formal logic — "natural deduction," "inference rules" — turns out to be the part you knew from programming all along. What logicians call *modus ponens*, you call *calling a function*.

3.3 Equality and the proof that $1 + 1 = 2$

The proposition $a = b$ is also a type. Its only constructor is reflexivity — `rfl` — which proves `a = a`. How can that ever prove anything interesting? Because Lean *computes* before comparing:

```
theorem one_plus_one : 1 + 1 = 2 := rfl
```

Lean unfolds `1 + 1` using the two-line definition of addition from Chapter 2, arrives at `2`, and sees that both sides are *literally the same value*. The equation holds by computation. This mechanism — *definitional equality* — is the engine that lets proofs lean on programs, and later lets us prove facts about extracted Rust code by, in part, just running it symbolically.

△ Pitfall

`rfl` proves $2^{255} - 19$-sized computations happily, but it can only prove what computation alone can see. `n + 0 = n` is `rfl` (the definition's first equation matches), yet `0 + n = n` is *not* — recursion is on the *second* argument, and `n` is an opaque variable, so nothing unfolds. The statement is still true; it just needs a real proof (induction — next chapter). The asymmetry feels unfair for about a day. Then it becomes your sharpest mental model of what a computer can and cannot know for free.

3.4 Universals, existentials, and dependent types

Cryptographic specifications are universal statements: "*for all* inputs, the output is correct." In Lean, $\forall$ is a function type whose *result type mentions the argument*:

```
theorem add_self_even : forall n : Nat, isEven (n + n) = true := ...
```

A proof of `forall n, P n` is a function that eats any `n` and returns a proof of `P n` — one uniform recipe covering all the infinitely many cases at once. This is precisely the thing testing could not give us in Chapter 1: testing produces finitely many `P 3, P 17, P 42`; a proof produces the function.

Dually, `exists n, P n` is proved by handing over a concrete witness together with evidence: `Exists.intro 4 pf`. And remember the `Rational` exercise from last chapter — the denominator you could not keep nonzero? Dependent types fix it by letting data carry proofs:

```
structure Rational where
```

```
num   : Int
den   : Nat
den_ne_zero : den ≠ 0    -- a PROOF, stored inside the value
```

No value of this type with a zero denominator can ever be constructed, anywhere, by anyone. In the real Ed25519 development this exact pattern appears as a *bounds invariant*: a field element travels together with the proof that its five limbs are small enough not to overflow the next multiplication. The data structure makes the unsafe states unrepresentable.

3.5 What about proof by contradiction?

One more resident of the logical zoo. Lean's core logic is *constructive*: a proof of existence builds a witness. Classical reasoning — "it's either true or false, and not false, hence true" — is available the moment you want it (Mathlib imports it as `Classical.choice`, and we will meet it again on the trust ledger in Chapter 11), but it is an *ingredient you can see*, not smuggled seasoning. When a verification result says "this proof uses only `propext`, `Classical.choice`, `Quot.sound`," that is a complete list of the logical beliefs you are being asked to hold. Three. You can audit them over coffee.

>_ Try it yourself

Open `exercises/Ch03.lean` and prove, as programs (no tactics yet!): `P -> Q -> P`; `(P / Q) -> (P  Q)`; and modus ponens `P -> (P -> Q) -> Q`. Each is a one-liner. Feel free to be delighted when the pieces click together like typed Lego.

Exercises

Exercise 3.1. Prove `and_assoc :  (P / Q) / R -> P / (Q / R)` as a term-mode program using `h.1`, `h.2`, and `And.intro`.

Exercise 3.2. Prove `or_swap :  P  Q -> Q  P`. You will need case analysis on which side holds: `match h with | Or.inl p => ...  | Or.inr q => ...`

Exercise 3.3. `Not P` is *defined* as `P -> False`. Using only that, prove `P -> Not (Not P)`. Write down in one sentence what program you just wrote.

Exercise 3.4. (Thought) Explain to a skeptical friend why a type with no constructors is the right representation of falsehood — and what would go wrong with the whole edifice if someone added a constructor to it.

⋈ Checkpoint

You should now be able to: translate each logical connective into its type (→, ∧, ∨, ¬, ∀, ∃); write small proofs as terms; explain why `rfl` proves `1 + 1 = 2` but not `0 + n = n`; and articulate the Curry–Howard slogan — *proofs are programs, propositions are types, checking is type-checking* — with a straight face and genuine conviction.

Chapter 4

Tactics: Proving as a Dialogue

4.1 From programs to conversations

Writing proofs as raw programs, as in Chapter 3, is honest work, but it scales badly: a real correctness proof for field multiplication would be a program the size of a small compiler. Nobody writes those by hand. Instead, Lean offers *tactic mode*: an interactive dialogue where you issue commands and Lean builds the proof program for you, step by step, showing you the remaining work after each move.

You enter the dialogue with the keyword `by`:

```
theorem and_swap (P Q : Prop) : P ∧ Q → Q ∧ P := by
  intro h
  constructor
  · exact h.2
  · exact h.1
```

Place your cursor after `by` in the editor and Lean shows the **goal state** — the exact logical situation at that point:

```
P Q : Prop
⊢ P ∧ Q → Q ∧ P
```

Everything above the turnstile ⊢ is what you *have* (the context); the line after it is what you *owe* (the goal). Every tactic transforms this picture. After `intro h`, the hypothesis moves above the line; after `constructor`, the goal splits in two. Proving becomes a game whose board you can always see.

∗ The big idea

A tactic proof is a **recorded conversation with the goal state**. The skill of proving is not memorizing tactic names — it is learning to *read the goal state* and recognize which of a handful of moves makes it simpler. Below is the core vocabulary; it covers the vast majority of every proof in the real Ed25519 development.

Tactic	When the goal looks like…	Effect
`intro h`	`P → Q`, `∀ x, P x`	assume it; name the evidence
`exact e`	anything	finish: `e` is a proof of the goal
`apply f`	`Q`, given `f : P → Q`	reduce the goal to `P`
`constructor`	`P ∧ Q`, `P ↔ Q`, …	split into pieces
`cases h`	have `h : P ∨ Q` (or `∧`, `∃`)	case analysis on `h`
`rw [eq]`	contains a rewritable subterm	replace using equation `eq`
`simp`	simplifiable clutter	rewrite with a lemma database
`induction n`	`∀ n : Nat, …`	base case + inductive step
`rfl`	`a = a` after computation	close by computation

4.2 Rewriting: equality as a tool

The workhorse tactic of equational reasoning is `rw` (rewrite). Given a proven equation, it replaces one side by the other inside your goal:

```
example (a b : Nat) (h : a = b) : a + a = b + b := by
  rw [h]      -- goal becomes: b + b = b + b, closed by rfl automatically
```

Chains of rewrites read like the two-column proofs of school geometry, except a machine checks every line. Here is commutativity-and-associativity shuffling, Mathlib lemmas by name:

```
example (a b c : Nat) : a + b + c = c + b + a := by
  rw [Nat.add_comm a b]        -- b + a + c = c + b + a
  rw [Nat.add_assoc]           -- b + (a + c) = c + b + a
  rw [Nat.add_comm a c]        -- b + (c + a) = c + b + a
  rw [← Nat.add_assoc]          -- b + c + a = c + b + a
  rw [Nat.add_comm b c]        -- done
```

The arrow ← rewrites right-to-left. Nobody enjoys writing five-line shuffles like this, which is exactly why Chapter 5 introduces `ring` — but you must *once* feel the manual version to understand what the automation is doing on your behalf.

4.3 Induction: the tactic that conquers infinity

Remember the embarrassment of Chapter 3: `0 + n = n` does not hold by computation. Now we can prove it — by induction, the proof technique that inductive types were born for:

```
theorem zero_add (n : Nat) : 0 + n = n := by
  induction n with
  | zero      => rfl                -- 0 + 0 = 0: computes
  | succ k ih => rw [Nat.add_succ, ih]
```

The `induction` tactic converts a statement about *all* naturals into two finite obligations: the statement

for `zero`, and the statement for `succ k` *assuming it for* `k` (the induction hypothesis `ih`). Because every natural number is built from those two constructors, the two cases cover infinity.

⋆ Aha

Induction is not a new axiom to swallow — it falls out of the inductive definition of `Nat` itself. "Every `Nat` is `zero` or a `succ`" *is* the license to do case analysis; recursion on the structure *is* the induction. Data and proof principle are two views of the same declaration. This is Curry–Howard paying rent again.

△ Pitfall

When a proof gets stuck, resist the urge to try random tactics — the formal-methods equivalent of mashing buttons. The goal state is telling you something. Three honest questions unstick most situations: (1) Is the statement actually true as written — check a small example with `#eval`! (2) Am I missing a hypothesis — is there an unstated bound or nonzero condition? (3) Is my induction on the right variable? In the real projects behind this book, "the proof is stuck" was, more often than not, the *statement* being subtly wrong — a truncated subtraction, a missing bound. The proof assistant was the messenger.

4.4 Structuring real proofs: `have` and `calc`

Big proofs are not flat lists of tactics; they are structured arguments with named intermediate results. The `have` tactic states and proves a stepping stone; `calc` lays out a chain of equalities or inequalities the way you would on a whiteboard:

```
example (a b : Nat) (h : a = 2 * b) : a + a = 4 * b := by
  have h2 : a + a = 2 * a := by rw [Nat.two_mul]
  calc a + a = 2 * a       := h2
       _     = 2 * (2 * b) := by rw [h]
       _     = 4 * b       := by rw [← Nat.mul_assoc]
```

This style is not cosmetic. In the verified field arithmetic you will read later, a single multiplication correctness proof is a `calc` chain tracking limb products through carries — dozens of steps, each trivial, whose *composition* is the theorem. `have` and `calc` are how proofs stay readable at that scale; they are also how they stay *maintainable*, because a broken step localizes the damage to one line.

There is one more structuring fact worth knowing early, because it saved the real project from a crash-course (literally — see Chapter 11): breaking a proof into small named `have` steps also controls the proof assistant's *memory appetite*. A monolithic "figure it all out at once" tactic call over a huge context can consume gigabytes; ten targeted steps, pennies each, prove the same thing. Structure is not just style — it is engineering.

>_ Try it yourself

Open `exercises/Ch04.lean`. It sets up each theorem with the goal state drawn in a comment, then asks you to: prove `and_swap` in tactic mode; prove `zero_add` *without* peeking above; and

repair a broken `calc` chain in which exactly one step is wrong. The third exercise is secretly the most realistic job training in this book.

Exercises

Exercise 4.1. Prove by induction: `∀ n : Nat, n + 0 = n` and `∀ n m : Nat, n + succ m = succ (n + m)`. (These are the mirror images of the definitional equations — the ones computation gives you for free — and together they yield commutativity.)

Exercise 4.2. Using the previous exercise, prove `∀ n m : Nat, n + m = m + n` by induction on `m`. Write out, in one prose sentence per case, what each branch of your proof says.

Exercise 4.3. Prove `∀ n : Nat, 2 * n = n + n` twice: once with `induction`, once with a single `rw` using a Mathlib lemma you find yourself (search hint: `exact?` asks Lean to search for you).

Exercise 4.4. (Reading) In the goal state `h : a < 2^51 ⊢ a * 19 < 2^56`, no induction is needed — this is pure arithmetic. Which tactic from the table would you *guess* handles it? (Answer next chapter; your guess is the point.)

⋈ Checkpoint

You should now be able to: read a goal state (context, turnstile, goal); drive the core tactics `intro`, `exact`, `apply`, `cases`, `rw`, `induction`; structure a multi-step argument with `have` and `calc`; and — most importantly — when stuck, interrogate the *statement* before blaming the proof.

Chapter 5

Numbers and Automation: Making the Machine Do the Boring Parts

5.1 The 90/10 rule of verification

Here is a trade secret: most of a real verification effort is not clever. Opening the correctness proof of Ed25519 field multiplication, you will find that the overwhelming majority of proof obligations are statements like

$$a < 2^{51} \;\wedge\; b < 2^{51} \implies a + b < 2^{52}, \qquad\qquad (a + b) \cdot c = a \cdot c + b \cdot c,$$

— bookkeeping a patient undergraduate could verify by hand in a minute each. There are *thousands* of them. The craft of modern verification is to hand exactly this 90% to decision procedures — tactics that implement a complete algorithm for a well-defined logical fragment — and save the human for the 10% that needs insight. This chapter is your tour of the arsenal.

5.2 `omega`: linear arithmetic, decided

The tactic you will use more than any other in this book's domain is `omega`. It completely decides *linear arithmetic* over integers and naturals: any goal built from variables, constants, $+$, $-$, multiplication *by constants*, $=$, $<$, $\leq$, $\neg$, $\wedge$, $\vee$, including the hypotheses in context.

```
example (a b : Nat) (h1 : a < 2^51) (h2 : b < 2^51) :
    a + b < 2^52 := by omega

example (a b : Nat) (h : a ≤ b) : a + (b - a) = b := by omega
  -- note: truncated Nat subtraction handled CORRECTLY -- omega knows
```

That second example deserves a salute: `omega` understands `Nat` truncation natively, defusing the Chapter 2 pitfall by algorithm rather than by vigilance. When the goal is false, `omega` *fails* — it is a decision procedure, so failure on a linear goal means the goal (with the hypotheses in view) is simply not true. That property turns `omega` into a *statement-debugging* tool: if it refuses your "obviously

true" bound, go find the counterexample; there is one.

△ Pitfall

`omega` does not touch multiplication of two *variables* ($a \cdot b$ is not linear), division in general, or bit-shifts by variables. For a goal mixing $a \cdot b$ with bounds, you often first name the product — `have hab : a * b ≤ 2^102 := ...` using a multiplication monotonicity lemma — and then let `omega` finish with `hab` as an opaque atom. This two-step, *bound the nonlinear part, then release the linear solver*, is the single most-used proof pattern in verified field arithmetic. You will write it dozens of times, and by Chapter 10 it will feel like breathing.

5.3 `decide`: when truth is a computation

Some propositions can be checked by running an algorithm to completion: "97 is prime," "these two sorted lists are equal," "$x^3 = x$ for all x in $\mathbb{Z}/6\mathbb{Z}$" (six cases — check them all). For any such *decidable* proposition, the `decide` tactic runs the decision algorithm inside Lean's kernel and turns the answer into a proof:

```
example : Nat.Prime 97 := by decide
example : ∀ x : ZMod 6, x^3 = x^3 := by decide   -- finite: try all six
```

The magic and the limitation are the same fact: the *kernel itself* re-executes the computation. That makes `decide` unimpeachable — and completely hopeless for our 77-digit prime $2^{255} - 19$, where trial division would outlast the universe. There is a variant, `native_decide`, that compiles the check to native code first — fast enough! — but it makes the compiler part of your trusted base, an IOU we will scrutinize hard in Chapters 7 and 11. For now, the rule of the house: **`decide` yes, `native_decide` never in a final certificate.**

5.4 `ring` and `norm_num`: algebra on tap

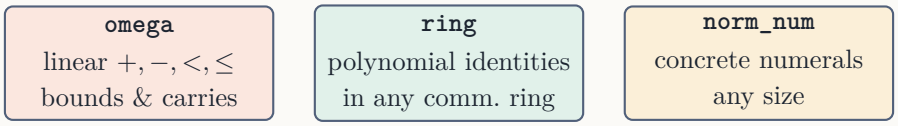
The five-line commutativity shuffle from Chapter 4? Here is the grown-up version:

```
example (a b c : Nat) : a + b + c = c + b + a := by ring

example (a b : ZMod p) : (a + b)^2 = a^2 + 2*a*b + b^2 := by ring

example : (2:Int)^255 - 19 > 2^254 := by norm_num
```

`ring` proves any identity that holds in every commutative ring — polynomial rearrangements, binomial expansions, distributivity avalanches — by normalizing both sides to a canonical polynomial form and comparing. `norm_num` evaluates concrete numeric facts, comfortable with numbers of any size. Between `omega`, `ring`, and `norm_num` you now hold the three keys that open most arithmetic doors:



the three keys of verified arithmetic — learn what each fragment *excludes*
and you will always know which door you are standing in front of

5.5 `simp`: the rewriting engine, and how to hold it

`simp` rewrites the goal to exhaustion using a curated database of thousands of "simplification" lemmas ($x + 0 \rightsquigarrow x$, `List.length (a :: l)` $\rightsquigarrow$ `l.length + 1`, ...). It is the most powerful tactic in Lean and the easiest to misuse. Used well, it clears brush so the real argument stands out. Used lazily — `simp [*]` with every hypothesis thrown in, in a context of sixty accumulated facts — it becomes a search over an enormous rewrite space: slow, fragile under library updates, and occasionally a memory monster.

This is not hypothetical. During the development this book accompanies, a single over-broad `simp`-style discharge in a fat context consumed twelve gigabytes of RAM and took down the machine. The postmortem produced house rules worth adopting from day one:

- Prefer `simp only [lemma1, lemma2]` — an explicit lemma list — in anything you intend to keep.
- Let `simp?` tell you the list: run it once interactively, then paste the `simp only [...]` it suggests into the file.
- Keep contexts lean (pun intended): a proof with sixty hypotheses in scope wants to be five `have`-steps with twelve each.

*** The big idea**

Automation is a *contract*, not a slot machine. Each tactic decides a known fragment: `omega` linear arithmetic, `ring` ring identities, `decide` finite computation, `simp only` a rewrite system you chose. The professional habit is to know *which* contract you are invoking — then failure is information ("this goal is not linear"; "this identity needs the modulus"), never mystery.

>_ Try it yourself

Open `exercises/Ch05.lean`: ten arithmetic goals, each solvable by exactly one of `omega` / `ring` / `norm_num` / `decide`. Your task is not just to close them but to close each with the *right* tool — the file rejects overkill by design. Goal number ten is the Chapter 4 cliffhanger: `a < 2^51 → a * 19 < 2^56`. (It is not linear — 19 is a constant, so it is! Think, then fire.)

Exercises

Exercise 5.1. For each, name the tactic and predict success before running: (a) `(a+b)*(a-b) = a*a - b*b` over `Int`; (b) `a < 100 → b < 100 → a*b < 10000` over `Nat`; (c) `Nat.Prime 65537`; (d) `2^51 + 2^51 = 2^52`.

Exercise 5.2. Goal (b) above is nonlinear, yet `omega` alone fails while the two-step pattern (bound the product with `Nat.mul_lt_mul` machinery, then `omega`) succeeds. Carry it out. Time yourself; the pattern should take under five minutes by the second attempt.

Exercise 5.3. Find a true statement about `Nat` that *no* tactic in this chapter proves in one shot, and sketch in prose how you would decompose it. (Anything genuinely inductive works — automation here decides arithmetic fragments, not all of mathematics.)

⋈ Checkpoint

You should now be able to: match a goal to its decision procedure by the shape of its operators; execute the bound-then-omega pattern for nonlinear bounds; explain why `decide` is trustworthy and where it hits its computational wall; and state the `simp` discipline — and the story of why this book is unusually sincere about it.

Chapter 6

Modular Arithmetic: The Number System Cryptography Lives In

6.1 Clock arithmetic, taken seriously

You already compute modulo twelve every day: four hours after ten o'clock is two o'clock. Wrap-around arithmetic — add, overflow the dial, keep the remainder — is the entire idea of *modular arithmetic*. Cryptography's only twist is the size of the clock: Ed25519's dial has

$$p = 2^{255} - 19$$

hours on it — a 77-digit prime. Everything else — the wrap-around, the remainder-taking, the algebra — is the twelve-hour clock you already know.

Formally, we write $a \equiv b \pmod{n}$ when n divides $a - b$, and we collect all integers with the same remainder into one object: the world $\mathbb{Z}/n\mathbb{Z}$ has exactly n elements, $\{0, 1, \ldots, n-1\}$, with addition and multiplication that wrap. In Lean/Mathlib this world is a first-class type, and computation in it is exactly what you would hope:

```
#eval (7 + 8 : ZMod 12)    -- 3     (fifteen o'clock is three o'clock)
#eval (5 * 9 : ZMod 12)    -- 9
#eval (3 - 7 : ZMod 12)    -- 8     (subtraction wraps -- no truncation!)

def p : Nat := 2^255 - 19
#eval (2^255 : ZMod p)     -- 19    (of course: 2^255 = p + 19)
```

Note the third line with relief: unlike `Nat`, subtraction in `ZMod n` is a total, well-behaved inverse of addition. Every element has a negative. In algebra terms, $\mathbb{Z}/n\mathbb{Z}$ is a *commutative ring* — and Lean knows it, so the `ring` tactic from Chapter 5 works there natively.

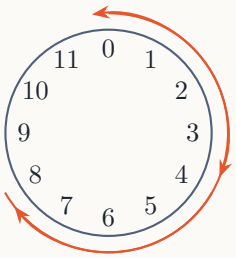
6.2 Division: where primes enter

Rings give us $+$, $-$, $\times$. Cryptography also needs $\div$ — and division is where the modulus stops being a size parameter and starts being a design decision. Dividing by a means multiplying by some a^{-1}

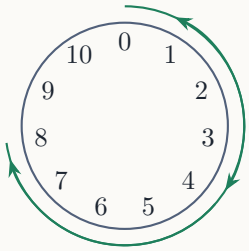
with $a \cdot a^{-1} = 1$. Does such an inverse exist? On the twelve-hour clock, try to invert 4: the multiples of 4 are $4, 8, 0, 4, 8, 0, \ldots$ — the value 1 never appears. 4 has no inverse mod 12, because 4 and 12 share the factor 4.

*** The big idea**

In $\mathbb{Z}/n\mathbb{Z}$, the element a is invertible exactly when $\gcd(a, n) = 1$. So if $n = p$ is **prime**, *every* nonzero element is invertible: you can divide freely, and $\mathbb{Z}/p\mathbb{Z}$ earns the title of **field**, written $\mathbb{F}_p$. Fields are the arithmetic paradise where linear algebra, polynomial factoring, and elliptic-curve geometry all work. This — and only this — is why cryptographic moduli are prime: primality is the entry ticket to division.



$\mathbb{Z}/12\mathbb{Z}$: stepping by 4 visits only $\{0, 4, 8\}$ — never 1



$\mathbb{Z}/11\mathbb{Z}$: stepping by 4 reaches 1 in three steps: $4^{-1} = 3$

How do you *find* a^{-1} mod p? Two classical answers, both of which appear in real Ed25519 code: the extended Euclidean algorithm, and Fermat's little theorem, which says $a^{p-1} \equiv 1 \pmod{p}$ for $a \not\equiv 0$ — hence a^{p-2} *is* the inverse. The dalek library computes inverses as a^{p-2} with a hand-crafted chain of 254 squarings and 11 multiplications; one of the proofs you will meet in Chapter 10 verifies precisely that this chain computes what Fermat promises.

6.3 Why $2^{255} - 19$? A prime chosen for machines

Any large prime makes a field. Why this one? Because arithmetic mod p is computed by *machines with 64-bit words*, and $p = 2^{255} - 19$ is shaped for them:

- **Reduction is a multiply-by-19.** Numbers at or beyond 2^{255} overflow the dial; but since $2^{255} \equiv 19 \pmod{p}$, any overflow bit re-enters the sum carrying a factor of just 19. Reducing mod p costs one small multiplication — no long division, ever.
- **255 bits split evenly into five limbs of 51.** A 64-bit word holding a 51-bit limb leaves 13 bits of headroom for carries to accumulate lazily — the delayed-carry style from Chapter 1, now by design rather than accident. The bound $a_i < 2^{51+\varepsilon}$ will follow us through Chapters 9 and 10.
- **It sits just below a power of two**, so 255-bit values fit in 32 bytes with one bit to spare — and Ed25519 spends that spare bit storing a point's sign. Engineering all the way down.

The Pasta curves verified in the companion projects (Pallas/Vesta, used in zero-knowledge proof systems) choose their $\approx 2^{254}$ primes by a different criterion — friendliness to fast Fourier transforms — and represent elements in *Montgomery form*, a representation trick we will touch in Chapter 9. Different shapes, same theory: pick a prime whose field your machine can love.

6.4 Modular arithmetic in Lean, hands on

Statements about $\mathbb{F}_p$ are ordinary Lean theorems, and the automation you already own applies:

```
-- ring identities hold verbatim in ZMod n:
example (x y : ZMod 12) : (x + y)^2 = x^2 + 2*x*y + y^2 := by ring

-- small finite worlds: check every case
example : ∀ x : ZMod 12, 4 * x ≠ 1 := by decide

-- inverses exist in prime fields (Mathlib knows ZMod 11 is a field):
example (a : ZMod 11) (h : a ≠ 0) : a * a⁻¹ = 1 :=
  ZMod.mul_inv_cancel_of_ne_zero h
```

What about the crown fact, $2^{255} \equiv 19$? For *concrete numeric* statements at 77-digit scale, tactic choice suddenly matters: `decide` would ask the kernel to grind case analysis it cannot afford, while `norm_num` and reflexivity-by-computation on efficient numerals handle it comfortably. Verifying real cryptography is partly a *performance engineering* discipline — Chapter 7 turns this observation into a principle when the question becomes primality itself.

△ Pitfall

In `ZMod n`, the numeral `57896044618658...` and the numeral `19` can be *the same element*. Equality of elements is not equality of the numerals you typed. When a surprising `rfl` succeeds — or two "different" constants turn out equal — remember you are on the clock face, not the number line. This is a feature: the entire verification story of Chapter 9 consists of choosing, deliberately, when to view a pile of bytes as an integer and when as a clock position.

>_ Try it yourself

Open `exercises/Ch06.lean`. You will: compute your first inverses with `#eval`; expand $(x+y)^3$ in `ZMod 7` with one tactic; show 4 has no inverse in `ZMod 12` by `decide`; and prove the reduction identity $2^{255} = 19$ in `ZMod p`.

Exercises

Exercise 6.1. Compute by hand (yes, hand): 3^{-1} in $\mathbb{Z}/7\mathbb{Z}$, and check that stepping by 3 on a 7-hour clock visits every position. Then confirm both with `#eval`.

Exercise 6.2. Prove in Lean: `∀ x : ZMod 12, 4 * x ≠ 1`, then change 12 to 13 and watch the same tactic refuse — find the witness it is implicitly pointing at.

Exercise 6.3. Fermat's little theorem is `ZMod.pow_card_sub_one_eq_one` in Mathlib. Use it to prove that in `ZMod 11`, `a^9 * a = 1` whenever `a ≠ 0`.

Exercise 6.4. (Paper) The Ed25519 code reduces a 256-bit value x by writing $x = x_{\text{low}} + 2^{255}\, x_{\text{high}}$ and returning $x_{\text{low}} + 19\, x_{\text{high}}$. Prove informally that this preserves the value mod p, and bound how

much smaller the result is. You have just re-derived the reduction step you will verify formally in Chapter 10.

⋈ Checkpoint

You should now be able to: compute in $\mathbb{Z}/n\mathbb{Z}$ and explain the notation; state exactly when division works and why primality guarantees it; give two independent reasons the constant 19 appears throughout curve25519 codebases; and prove small modular facts in Lean with `decide`, `ring`, and a Mathlib lemma found by name.

Chapter 7

Convincing a Paranoid Kernel That a 77-Digit Number Is Prime

7.1 The problem nobody warns you about

Chapter 6 ended with a quiet dependency: everything — division, field structure, the whole elliptic curve — rests on $p = 2^{255} - 19$ *being prime*. In Lean, that is a proposition like any other, and it must be *proved*:

```
theorem p_prime : Nat.Prime (2^255 - 19) := ?
```

Your Chapter 5 instincts say `decide`: primality is decidable — just try dividing. But trial division tests divisors up to $\sqrt{p} \approx 2^{127}$. At a billion billion divisions per second, that is about 10^{12} ages of the universe. The kernel, which happily *re-executes* every computation you feed it, cannot afford this one. And mathematicians clearly believe this number is prime — so how does *anyone* know?

7.2 Certificates: the deep idea hiding here

The answer reorganizes how you think about computation. *Finding* a fact and *checking* a fact can have wildly different costs. What we need is a **certificate**: a piece of data, possibly expensive to discover, that makes the fact *cheap to verify*.

You have met certificates before without the name. A composite number's certificate is a factor: finding a factor of a 77-digit number may be hard, but checking $n = a \cdot b$ is one multiplication. The beautiful surprise — Pratt's theorem, 1975 — is that *primality* has certificates too:

⁎ The big idea

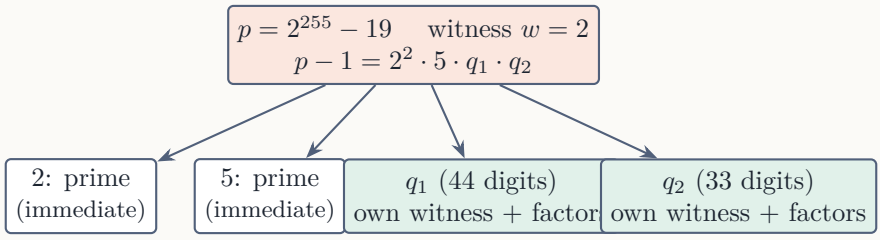
Pratt certificate. To certify that p is prime, exhibit a *witness* w such that

$$w^{p-1} \equiv 1 \pmod{p} \qquad \text{and} \qquad w^{(p-1)/q} \not\equiv 1 \pmod{p} \ \text{ for every prime factor } q \text{ of } p-1.$$

Such a w generates all $p-1$ nonzero residues, which forces $\mathbb{Z}/p\mathbb{Z}$ to have $p-1$ invertible elements

— something only a prime modulus allows. Checking the certificate costs a handful of modular exponentiations (milliseconds, by fast squaring), *plus recursively certifying the prime factors* q — each much smaller, so the recursion collapses fast.

Concretely for our hero: $p - 1 = 2^{255} - 20 = 2^2 \cdot 5 \cdot q_1 \cdot q_2$ where q_1 (44 digits) and q_2 (33 digits) are themselves prime, with their own small certificates. The full certificate for p is a small tree of witnesses and factorizations — a few hundred bytes of data standing behind a 77-digit claim:



each node: milliseconds to check; the whole tree: a proof

⋆ Aha

This find/check asymmetry is one of the great ideas of computer science — it is the P versus NP distinction wearing work clothes, and it is the engine of zero-knowledge proof systems (the very technology the Pasta curves serve). Proof assistants run on it too: Lean's whole architecture — clever tactics *finding*, dumb kernel *checking* — is the same asymmetry. A proof *is* a certificate.

7.3 The tempting shortcut, and why the house declines it

Lean offers a faster `decide`: the variant `native_decide` compiles the decision procedure to native machine code, runs it at full speed, and asserts the result. With a good primality test behind it, it can dispatch `Nat.Prime p` in seconds. Case closed?

Look at what you would be trusting. Ordinary `decide` produces a computation the *kernel* replays — the few-thousand-line paranoid core remains the only thing you trust. `native_decide` instead makes the theorem's truth depend on the Lean *compiler*, the C toolchain behind it, and the runtime — hundreds of thousands of lines promoted into your trusted base, in exchange for convenience on one theorem. Every proof downstream of the field — group law, scalars, signatures — would inherit that enlarged trust, visible forever in its axiom report (Chapter 11 shows you how to read those).

△ Pitfall

`native_decide` is not "cheating," and for exploratory work it is a fine tool. The trap is *silent trust inflation*: its use is invisible at the theorem statement — the cost appears only when someone audits the axioms, which is exactly what most readers never do. House rule, adopted from the projects this book accompanies: exploratory scaffolding may use it; **no shipped certificate depends on it**. The final Pallas-modulus primality proof in `pasta-pallas-verified` is a kernel-checked Lucas/Pratt certificate for precisely this reason.

7.4 Certificates in practice: Mathlib's toolbox

You will not hand-roll witness trees. Mathlib provides the machinery (`Nat.Prime` decision lemmas, `lucas_lehmer`-style infrastructure, and the `norm_num` extension `Nat.Prime` plugin) that constructs and checks Pratt-style certificates behind a single tactic call — while keeping every step kernel-checked. The shape in real code:

```
theorem p_prime : Nat.Prime (2^255 - 19) := by
  norm_num  -- certificate-backed primality, kernel-checked, ~seconds
```

When the built-in route struggles (very large or awkward moduli), the fallback is explicit: state the witness data as definitions, prove the two Pratt conditions with `norm_num`-driven modular exponentiation, and assemble. That is exactly the structure of the Pallas certificate in the companion repository — worth reading now with fresh eyes: `pasta-pallas-verified/verification/Proofs/Primality.lean`.

>_ Try it yourself

Open `exercises/Ch07.lean`. Ladder: certify 97, then 65537 (a Fermat prime beloved of RSA), then the ten-digit Mersenne prime $2^{31} - 1$, watching what each tool costs as the numbers grow. (Amusingly, the find/check asymmetry bites the *tactic* too: `norm_num` must *find* the witness tree before the kernel checks it, and at $2^{61} - 1$ the finding already takes minutes.) Finale: implement square-and-multiply modular exponentiation yourself and check the top witness condition for $p = 2^{255} - 19$ with `#eval` — your own hands on the certificate, at 77 digits, in milliseconds.

Exercises

Exercise 7.1. Verify by hand that $w = 2$ is a Pratt witness for $p = 13$: compute 2^{12} mod 13 and $2^{12/q}$ mod 13 for each prime $q \mid 12$. Write the full certificate tree for 13, recursing into the factors of 12.

Exercise 7.2. Why does the witness condition force primality? Sketch the argument: if w has order exactly $p - 1$ in $\mathbb{Z}/p\mathbb{Z}$, then the multiplicative structure has $p - 1$ elements, which fails if $p = ab$ with $1 < a, b < p$. (Full rigor optional; the shape is the point.)

Exercise 7.3. Estimate, in modular multiplications, the cost of checking the certificate for $2^{255} - 19$: count squarings for one exponentiation at 255 bits, times the number of conditions in the tree above. Compare with the 2^{127} divisions of trial division. Write both numbers down next to each other. Smile.

Exercise 7.4. (Discussion) Bitcoin miners *find* block hashes; nodes *check* them. GPS receivers *check* satellite signals they could never *find*. Name two more systems built on the find/check asymmetry, and one system that would collapse without it.

⋈ Checkpoint

You should now be able to: explain why `decide` cannot prove $2^{255} - 19$ prime while a certificate can; reproduce the two Pratt witness conditions and check them on a small prime; articulate exactly what additional trust `native_decide` would introduce and why shipped certificates decline it; and recognize the find/check asymmetry as the common engine of certificates, proof assistants, and the P-vs-NP question.

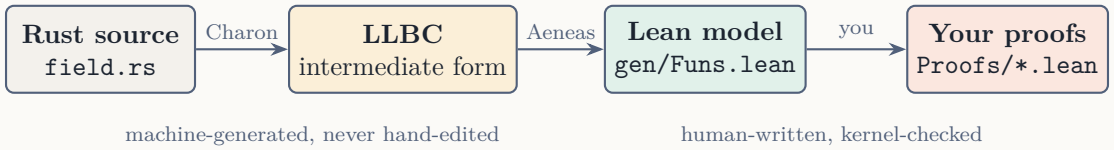
Chapter 8

From Rust to Lean: Verifying the Code That Actually Ships

8.1 The transcription problem

Everything so far proved facts about *Lean* programs. But the Ed25519 that guards your SSH connection is written in *Rust* (in our case, `curve25519-dalek` and its forks). An obvious plan: read the Rust, rewrite it in Lean by hand, verify the rewrite. The plan has a hole you could drive a key-recovery attack through: **what if you transcribe it wrong?** A hand-copy that silently fixes a bug — or introduces one — makes the proof a beautiful statement about code nobody runs.

The projects behind this book close the hole with a mechanical translation pipeline:



Charon compiles the Rust crate into LLBC ("low-level borrow calculus"), a simplified intermediate representation. **Aeneas** translates LLBC into pure Lean functions. The generated Lean — the *model* — lands in a `gen/` directory with a strict house rule: *never edit it.* Regenerate it from source, or don't touch it. Your proofs import the model and state theorems about it.

⁎ The big idea

The object of verification is the **extracted model**, produced from the shipping source by a deterministic tool — not a human transcription. The trust question shifts from "did we copy the code right?" (unauditable squinting) to "does the translator preserve meaning?" (one tool, studied once, shared by every project that uses it). You will hear this called *shrinking the trusted base*: swap many ad-hoc trusts for one well-examined trust.

8.2 What extracted code looks like

Here is real input and real output, lightly abridged. The Rust (from `curve25519-dalek`, radix-51 field addition):

```
impl Add for FieldElement51 {
    fn add(self, rhs: &FieldElement51) -> FieldElement51 {
        let mut output = *self;
        for i in 0..5 {
            output.0[i] += rhs.0[i];
        }
        output
    }
}
```

And the Lean model Aeneas produces for it (shape, not verbatim):

```
def fieldElement51_add (self rhs : Array U64 5) :
    Result (Array U64 5) := do
  let a0 <- Array.index_usize self 0
  let b0 <- Array.index_usize rhs 0
  let s0 <- a0 + b0            -- U64 addition: can FAIL on overflow
  let out <- Array.update self 0 s0
  ...                          -- and so on for limbs 1..4
```

Three features deserve your full attention, because every proof in the next two chapters engages them:

- **Machine integers are honest.** `U64` is not $\mathbb{N}$; it is 64-bit words. Aeneas models arithmetic on them precisely, overflow included.
- **Everything returns `Result`.** The `do`/`<-` notation threads a computation that can *fail* — returning an error value instead of a result — exactly where the Rust could panic or overflow. There is no pretending partial functions are total.
- **It is ugly.** Five limbs times load-add-store, all sequenced. Machine-generated code has no taste. The *proofs* restore the elegance; the model's job is fidelity.

8.3 Overflow is a proof obligation, not a footnote

In Rust, `a + b` on `u64` panics in debug mode and wraps in release mode when it overflows. In the extracted model, `a + b` returns a `Result` that is an error unless the mathematical sum fits. So the innocent theorem "add returns the right field element" *cannot even be stated* without first proving *add returns at all*:

```
theorem add_spec (a b : Array U64 5)
    (ha : LimbsBounded a) (hb : LimbsBounded b) :
    ∃ c, fieldElement51_add a b = .ok c ∧ LimbsBounded c ∧ ...
```

That hypothesis `LimbsBounded` — each limb below 2^{54}, say — is the bounds invariant promised in Chapters 3 and 6: 51-bit payload plus headroom, so limb additions cannot reach 2^{64}. The specification exposes what the Rust comments only whisper: this code is correct *under a discipline of bounded inputs*, the discipline must be maintained by every caller, and now there is a machine checking that it is.

⋆ Aha

Notice what just happened to "ugly generated code with Results everywhere": it forced us to discover, state, and prove the *implicit operating envelope* of the optimized implementation. The dalek authors knew this envelope; it lived in comments and code-review lore. Now it is a theorem. Extraction does not merely enable verification — it *interrogates* the code.

8.4 Practicalities: extraction as surgery

Running Charon on a whole real-world crate drags in everything the crate touches — iterators, byte serialization, trait machinery, SIMD backends — much of it irrelevant to the arithmetic core and some of it beyond what the translator supports. The working method, learned the honest way in the companion projects:

- **Extract functions, not crates.** Charon accepts specific roots (individual functions and impls); the extraction scripts in each companion repo (`extract.sh`, `extract-scalar.sh`) name exactly the arithmetic functions and get a small, clean model — 28 definitions instead of a thousand.
- **Some code will not translate.** The dalek scalar-multiplication backends use CPU-specific SIMD intrinsics no translator models. The boundary is then *documented*: those functions enter the trusted base, stated as assumptions, visible in every audit. Honest boundaries beat heroic fictions (Chapter 11 dwells on this).
- **Pin your tools.** The pipeline records exact versions of Charon, Aeneas, and Lean. A model regenerated with a different translator version is a *different model*; reproducibility of the proofs starts with reproducibility of the artifact under proof.

△ Pitfall

When an extraction fails or a model looks bizarre, the temptation is to "fix" the generated Lean by hand. Resist absolutely. A hand-edited model is a hand-transcription with extra steps — the exact hole this pipeline exists to close. The fixes live in extraction scope (choose different roots), in the source (rarely), or in documented assumptions (openly).

>_ Try it yourself

Open the companion repo `dalek-ed25519-verified`: read `extract.sh` (the roots), skim `verification/gen/Funs.lean` for `add`/`sub` (recognize the load-add-store pattern above), then read the first twenty lines of `verification/Proofs/FieldSpec.lean` and identify: the bounds invariant, the `Result` handling, and the statement of the theorem. You now recognize every structural element. The mathematics inside is Chapters 9 and 10.

Exercises

Exercise 8.1. The Rust expression `output.0[i] += rhs.0[i]` hides four distinct failure/effect points that the Lean model makes explicit. Name them. (Hint: two indexings, one arithmetic operation, one write.)

Exercise 8.2. Suppose limbs are bounded by 2^{54}. What is the largest value `a0 + b0` can take, and how many such additions can chain before a `U64` overflow becomes possible? Show the margin calculation — this is precisely why the invariant is 2^{54} and not 2^{63}.

Exercise 8.3. (Design) Your colleague proposes verifying a hand-written Lean "reference implementation" instead of the extracted model, because it is prettier. List two failure modes this reintroduces, and one legitimate use a reference implementation still has (hint: Chapter 9 uses one as the *specification* side).

⋈ Checkpoint

You should now be able to: draw the Rust → LLBC → Lean pipeline and say what each stage preserves; explain why generated models are never hand-edited; read a `Result`-typed extracted function and point to where overflow lives; and state why "the proof needs a bounds hypothesis" is a discovery about the *code*, not a weakness of the method.

Chapter 9

The Denotation Bridge: What Do Five Numbers *Mean*?

9.1 Two worlds, one bridge

We now hold two very different objects. On one side, mathematics: the field $\mathbb{F}_p$, where elements are abstract clock positions and $+$ means ideal modular addition. On the other side, the extracted model: arrays of five `U64` words, shuffled by loads, adds, and stores. The entire question of verified cryptography is how to say — precisely — that the second *implements* the first.

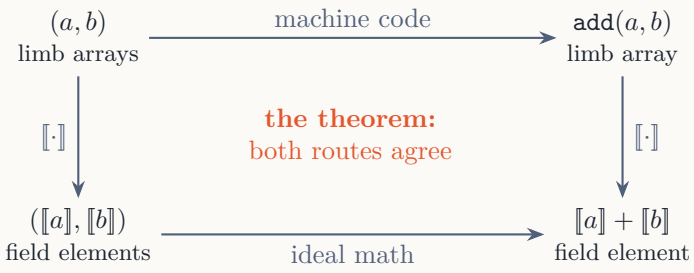
The answer is a single function, small enough to write on one line and important enough to carry this whole book. Given limbs $a = (a_0, a_1, a_2, a_3, a_4)$, define the **denotation**:

$$[\![a]\!] \;=\; a_0 + 2^{51}a_1 + 2^{102}a_2 + 2^{153}a_3 + 2^{204}a_4 \;\in\; \mathbb{F}_p.$$

Read $[\![a]\!]$ as "the field element these limbs *mean*." It is positional notation, nothing more — base 2^{51} instead of base 10, with the result interpreted on the clock face of $\mathbb{F}_p$. In Lean:

```
def denote (a : Array U64 5) : ZMod p :=
  a[0].val + 2^51 * a[1].val + 2^102 * a[2].val
           + 2^153 * a[3].val + 2^204 * a[4].val
```

With the bridge in hand, correctness of an operation becomes a *commuting square* — one picture you should internalize until you see it in your sleep:



⁎ The big idea

The correctness of an implementation is the statement that denotation commutes with every operation:

$$[\![\texttt{add}(a,b)]\!] = [\![a]\!] + [\![b]\!], \qquad [\![\texttt{mul}(a,b)]\!] = [\![a]\!] \cdot [\![b]\!], \qquad \dots$$

The left-hand side lives in the machine world (with its bounds hypotheses and `Results`); the right-hand side is pure mathematics. One equation per operation, and the ugly optimized code is pinned, forever, to the textbook meaning. Every verified-crypto project you will ever read is this diagram, instantiated.

9.2 Why redundancy is freedom (and where bugs hide)

A subtlety with consequences: denotation is **many-to-one**. The limb arrays $(19, 0, 0, 0, 0)$ and $(p + 19 \bmod 2^{\cdots}, \dots)$ — or more mundanely, unreduced sums whose limbs exceed 2^{51} — can denote the *same* field element. The representation has slack, and the implementation *exploits* it: the fast `add` from Chapter 8 just adds limbs pairwise, letting values drift above 2^{51}, and nobody reduces until a cheaper moment. The commuting square still closes because $[\![\cdot]\!]$ doesn't care how bloated the limbs are — positional value is positional value.

This is also exactly where the carry bugs of Chapter 1 live: code that is correct only while the drift stays within headroom, and wrong on the rare inputs where it spills. In the verified development that danger becomes a visible, machine-checked pair of clauses attached to every operation:

```
theorem add_spec (ha : Bnd54 a) (hb : Bnd54 b) :
    ∃ c, add a b = .ok c
      ∧ Bnd55 c                          -- (1) bounds: the envelope holds
      ∧ denote c = denote a + denote b -- (2) value: the meaning is right
```

Clause (2) is the commuting square. Clause (1) feeds the *next* operation's hypothesis — correctness composes only because every theorem hands the following one the envelope it requires. A chain of such specs is the formal skeleton of "this sequence of optimized operations computes the formula we claim."

⋆ Aha

The denotation idea is vastly older and bigger than cryptography. Compilers prove "optimized code means the same as naive code"; databases prove "this query plan means the same query"; hardware verifies "this pipelined circuit means this instruction set." The pattern — map both sides into a mathematical meaning-space and prove the square commutes — is called *denotational semantics*, and you have now used it for real. It is the single most transferable idea in this book.

9.3 Multiplication: where the bridge earns its keep

Addition's square closes in an afternoon. Multiplication is the boss fight, and seeing *why* teaches you what verified arithmetic is really like. Schoolbook multiplication of two 5-limb numbers produces nine columns of partial products $\sum_{i+j=k} a_i b_j$; each column then owes a *carry* to the next; and columns $k \geq 5$ — weights 2^{255} and up — must be folded back using the Chapter 6 identity $2^{255} \equiv 19$. The implementation interleaves all three concerns for speed. The proof must un-interleave them:

$$[\![\texttt{mul}(a,b)]\!] \stackrel{?}{=} \left(\sum_{k=0}^{8} 2^{51k} \sum_{i+j=k} a_i b_j\right) \bmod p \stackrel{?}{=} [\![a]\!] \cdot [\![b]\!].$$

The right equality is algebra — `ring` territory. The left is a walk through the extracted code: every intermediate `U128` product bounded (no overflow — the 2^{54} headroom at work), every carry accounted, every $\times 19$ fold placed. In the companion projects this is a long `calc`-and-`have` museum: dozens of small steps, each dispatched by `omega` or a bound lemma, composed into one commuting square.

One more representational dialect, because you will meet it in the Pasta repos: **Montgomery form** stores x as $x \cdot R \bmod p$ (with $R = 2^{256}$) because it makes reduction after multiplication cheap. The bridge absorbs the twist without complaint — define $[\![a]\!]_{\mathrm{M}} =$ (positional value of a) $\cdot R^{-1}$ and the same commuting squares govern everything. Denotation is a *policy about meaning*, and it bends to fit the representation, not the other way around.

△ Pitfall

When a denotation proof refuses to close, the failure is information — read it like a detective, in order: (1) Is the *bound* hypothesis strong enough for the intermediate products? (Count bits, on paper.) (2) Is the *denotation* right for this representation — radix, limb count, Montgomery factor? (3) Only then suspect the code. In the companion projects this checklist ran hundreds of times; its order reflects the actual base rates of what was wrong.

>_ Try it yourself

Open `exercises/Ch09.lean`. It builds a miniature of the whole story you can hold in your head: a *2-limb, radix-4* representation of $\mathbb{Z}/15\mathbb{Z}$ (limbs are values 0–3, denotation $a_0 + 4a_1$, and $16 \equiv 1$ makes the fold trivial). You will write `denote`, prove the commuting square for the provided `add` with carry, then for `mul` with its fold — every conceptual ingredient of the dalek proof, at a scale where `decide` can double-check your work.

Exercises

Exercise 9.1. Compute by hand the denotation of the limb arrays $(19, 0, 0, 0, 0)$ and $(0, 0, 0, 0, 2^{51})$ in the radix-51 system, reducing mod $p = 2^{255} - 19$. Conclude that $[\![\cdot]\!]$ is not injective by exhibiting the collision.

Exercise 9.2. In the mini-system of the Try It box, find two distinct limb pairs denoting the same element of $\mathbb{Z}/15\mathbb{Z}$, and check that the provided `add` treats them interchangeably *as far as denotation goes* — compute both sides.

Exercise 9.3. Sketch the multiplication column sums $\sum_{i+j=k} a_i b_j$ for the 2-limb system and carry out the fold $16 \equiv 1$ by hand for $a = (3, 2)$, $b = (1, 3)$. Check against direct computation in $\mathbb{Z}/15\mathbb{Z}$.

Exercise 9.4. (Paper, challenge) For the radix-51 system with limbs bounded by 2^{54}: bound one column $\sum_{i+j=4} a_i b_j$ of partial products and confirm it fits a `U128`. How much headroom remains? This number — not elegance — is why the invariant chose 2^{54}.

⋈ Checkpoint

You should now be able to: write the radix-51 denotation from memory; draw the commuting square and label which side owns bounds and `Results`; explain why many-to-one representation is both the performance trick and the bug habitat; and recognize the two-clause shape (bounds propagation + value equation) as the universal skeleton of implementation-correctness theorems.

Chapter 10

Verifying a Field: The Full Campaign

10.1 The summit statement

Every thread so far — specs as types, tactics, automation, $\mathbb{F}_p$, primality, extraction, denotation — was preparation for one theorem. In the companion repositories it is called the *field implementation certificate*, and (lightly paraphrased) it says:

```
theorem fieldImplementation :
    -- p is prime, so ZMod p is genuinely a field
    Nat.Prime p
    -- and for all bounded limb arrays, every operation's
    -- commuting square closes:
  ∧ (∀ a b, Bnd a → Bnd b → AddSquare a b)
  ∧ (∀ a b, Bnd a → Bnd b → SubSquare a b)
  ∧ (∀ a b, Bnd a → Bnd b → MulSquare a b)
  ∧ (∀ a,   Bnd a →         SquareSquare a)
  ∧ (∀ a,   Bnd a →         InvertSquare a)   -- Fermat chain: a^(p-2)
  ∧ ...                                       -- reduce, negate, encode
```

One theorem, kernel-checked, quantified over *every* input the representation admits: the extracted dalek field arithmetic implements $\mathbb{F}_p$. This chapter is the story of the campaign that proves it — told honestly, including the two places where the terrain fought back, because the failures teach more than the victories.

10.2 Order of battle

You do not prove such a conjunction by heroism; you prove it by sequencing. The campaign order in the real projects, and the reason for each position:

1. **Bounds lemmas first** — pure `omega` facts about limb sizes, no denotation at all. Cheap, and everything depends on them.
2. **Primality** (Chapter 7) — independent of the code entirely; it dignifies `ZMod p` into a field.

3. **add, sub, negate** — linear operations; the commuting squares close with `omega` plus the denotation unfolds. Confidence builders.

4. **reduce** — the ×19 fold in isolation. Proving it alone, before mul uses it, halves the hardest proof's size.

5. **mul, square** — the boss fight of Chapter 9: column sums, carries, folds. Squaring is mul with algebraic shortcuts — a separate code path in dalek, hence a separate theorem. No shortcuts in the proof: the *code's* shortcut is precisely what needs checking.

6. **invert** — the 254-squaring Fermat chain, verified as a `calc` of exponent bookkeeping on top of `mul`/`square` specs, ending at a^{p-2}; Fermat's little theorem (Mathlib's) closes the square.

Notice the shape: *each layer consumes only the specs of the layer below*, never reaching into implementations. By step 6 you are doing exponent arithmetic, blissfully ignorant of carries. That is the compositionality that Chapter 9's two-clause specs (bounds + value) were designed to buy.

10.3 Dispatches from the terrain

The wall that was really there. Partway up, one proof style hit a genuine limit of the tool: correctness certificates for `mul`-scale goals, when handed to a general decision procedure in one monolithic call, generate internal certificates with coefficients on the order of 2^{256} — and checking them can exhaust the proof checker's memory. One such call, during the development of the Pasta field proofs, consumed twelve gigabytes and crashed the machine (Chapter 5 told you this story from the tactic side). The cure was never cleverness — it was *decomposition*: isolate each carry step as its own small lemma with a tiny context, prove the value identity with `linear_combination` (a tactic that checks a *stated* linear certificate rather than searching for one), and let the big theorem be an assembly of small checked parts. The same discipline, plus hard memory caps on the checker process, became house infrastructure.

⋆ Aha

There is a deep symmetry in that fix worth savoring: the *proof* was restructured exactly the way the *code* was — into small steps with controlled intermediate size. Delayed carries in the implementation; small lemmas in the verification. Bounded limbs; bounded contexts. Good proofs and good fast code turn out to obey the same engineering aesthetics. This is not a coincidence; both are fighting combinatorial growth with structure.

Four forks, one method, real divergence. The companion projects verify not just upstream `curve25519-dalek` but three production forks (Solana's, RISC Zero's, Betrusted's) — each against *its own* extraction. Worth it? The audit found the forks implement the same constant-time conditional selection three different ways (a `subtle` crate trait, a volatile-read `black_box`, a hand-rolled arithmetic mask), and one fork reorders instructions in point doubling. All correct — *provably*, now — but the divergence is exactly the kind of thing that silently breaks when someone "harmonizes" code during a rebase. Per-fork verification is not pedantry; it is how you notice that "the same library" isn't.

△ Pitfall

A verified fork is verified *at a commit*. Change one line of arithmetic and the certificate is stale — that is a feature (the proof *should* break when the code changes), but it means verification is a *process wired into maintenance*, not a trophy. The companion repos ship `check.sh` scripts that re-extract and re-verify from scratch; treat those as the project's pulse, not as CI decoration.

10.4 Reading a certificate like a professional

Suppose a stranger hands you a repository claiming "formally verified field arithmetic." Chapter 11 gives you the full audit protocol, but the field-layer questions you can already ask are these:

- **What is denoted?** Find the denotation function. Does it map the *extracted* representation (good) or a hand-written lookalike (Chapter 8's hole)?
- **Is the square complete?** Value equation *and* bounds propagation *and* the `.ok` clause — a spec that assumes success proves nothing about overflow.
- **Is the quantifier honest?** ∀ `a b` with bounds hypotheses, or a handful of `examples` on constants dressed up as coverage?
- **Does anything say `sorry`?** One `sorry` anywhere in the dependency chain and the certificate is decorative. The checker will tell you; ask it.

>_ Try it yourself

Do the audit for real: open `dalek-ed25519-verified`, find the denotation, pick the `sub` spec, and check the three clauses against the list above. Then run the repo's `check.sh` and watch the whole pyramid rebuild. Total time: one coffee. Skill acquired: permanent.

Exercises

Exercise 10.1. Field subtraction in dalek computes $a - b$ as $a + (16p - b)$ limb-wise (adding a multiple of p keeps limbs positive — recall `Nat` truncation!). State the commuting square for `sub` including the exact multiple-of-p fact you would need as a lemma, and explain why $16p$ rather than p.

Exercise 10.2. The Fermat inversion chain computes a^{p-2} via 254 squarings and 11 multiplications. Verify the exponent bookkeeping for the first three steps of dalek's actual chain: a^2, $a^9 = (a^2)^{2\cdot 2} \cdot a$, $a^{11} = a^9 \cdot a^2$. (The full chain is exercise-by-induction: each step's exponent is a sum of previous ones — addition-chain arithmetic.)

Exercise 10.3. (Discussion) Step 5 refuses to trust the squaring shortcut and verifies `square` separately from `mul`. A colleague argues "square is just `mul a a`, reuse the theorem." What, concretely, would that argument miss about the code under verification?

⋈ Checkpoint

You should now be able to: state the field implementation certificate and every quantifier in it; recite the campaign order and justify why bounds and `reduce` come early; tell the memory-wall story and its decomposition moral; and audit a stranger's field-layer claim with four pointed questions.

Chapter 11

Honesty, Axioms, and the Art of Trusting Proofs

11.1 "Formally verified" is a claim, not a spell

The phrase *formally verified* has marketing gravity, and marketing gravity attracts abuse. This chapter arms you with the auditor's toolkit: what a Lean certificate actually rests on, how to interrogate it in one command, and the specific ways a verification claim can be hollow while every file compiles. Nothing here is hypothetical — each failure mode appears in the wild, and the discipline below is the one the companion projects hold themselves to.

11.2 The trust ledger

When the kernel accepts `fieldImplementation`, what exactly are you being asked to believe? Lean can tell you — precisely:

```
#print axioms fieldImplementation
-- 'fieldImplementation' depends on axioms:
-- [propext, Classical.choice, Quot.sound]
```

`#print axioms` walks the *entire* dependency tree of a theorem — every lemma, every lemma's lemmas, down to bedrock — and reports every assumption found there. The three names above are Lean's standard trio (propositional extensionality, classical choice, quotient soundness): ordinary classical mathematics, accepted by working mathematicians, scrutinized by logicians for a century. A certificate reporting exactly these three is called **axiom-clean**. Anything *else* in that list is a custom assumption someone added — and the whole audit consists of reading that list and asking whether you believe each entry.

⁎ The big idea

A machine-checked theorem is a receipt with a complete list of its own assumptions — something prose mathematics has never had. But the receipt only protects people who read it. **The**

one-command audit: run `#print axioms` on the headline theorem; anything beyond `[propext, Classical.choice, Quot.sound]` is where the bodies are buried. Make this reflex, and no verification claim can hide from you.

11.3 A field guide to hollow certificates

Compiling proofs can still be worthless. The four classic ways, in increasing order of subtlety:

1. The `sorry`. Lean's placeholder accepts any goal with a warning. Fine for work in progress; fatal in a certificate, because downstream theorems inherit the hole silently. Detection: the compiler warns, and `#print axioms` shows `sorryAx`. Trivial to catch — if you look.

2. The smuggled axiom. Stuck on a lemma? `axiom` makes it true. Sometimes legitimate (see the trusted-base section below); rotten when undisclosed — a proof "of" the group law that axiomatizes the hard half of the group law is theater. Detection: `#print axioms`, always.

3. The trivial specification. The most instructive one. Consider:

```
theorem mul_correct : ∀ a b, ∃ c, mul a b = c := by
  intro a b; exact ⟨_, rfl⟩   -- checks! and says NOTHING
```

Kernel-approved, axiom-clean, and utterly empty: "mul returns whatever it returns." No tool catches this, because nothing is wrong *formally* — the defect is that the statement doesn't say what the reader assumes it says. The only detector is a human reading the *statement* (never mind the proof) and asking: *if the code were wrong, would this theorem fail?* For the trivial spec, the answer is no — a buggy `mul` satisfies it identically.

4. The wrong model. The proof is real, the spec is strong — but the thing verified isn't the thing that ships: a hand-transcription (Chapter 8), a stale extraction, a simplified semantics. Detection: trace the chain from artifact to source — extraction scripts, pinned tool versions, regeneration instructions. If the chain can't be replayed, the claim is about an orphan.

△ Pitfall

Rank these by danger and notice the inversion: the crude failures (`sorry`, smuggled axioms) are machine-detectable in seconds, while the subtle ones (trivial specs, wrong models) defeat every automated check and yield only to a thoughtful reader. Verification does not eliminate the need for human judgment; it *concentrates* all of it into two small, well-lit places — the statement and the model. That concentration is the gift; squandering it by not reading the statement is the sin.

11.4 Honest boundaries: the trusted base

Real projects meet real limits: SHA-512's compression function, SIMD backends no translator models, foreign function calls. The honest move is not to pretend, but to *declare*: state the unverified piece as an explicit assumption, document it in a ledger (the companion repos call theirs `TRUSTED-BASE.md`),

and let `#print axioms` carry the disclosure to every downstream theorem automatically.

```
-- Declared, documented, and visible in every audit forever:
axiom sha512_spec : ∀ msg, Sha512.hash msg = SHA512_ideal msg
```

A signature-layer certificate honestly reads: *EdDSA verification is correct, GIVEN the hash behaves ideally and GIVEN the documented backend assumptions* — with both *given*s machine-visible. Compare the two postures: "everything verified!" (and hope nobody checks) versus "these two assumptions, this ledger, audit me" — the second is both humbler and *stronger*, because its claim survives the audit.

The companion projects add one more layer of candor worth copying: a *failure map*. Their control repository documents the dead ends — tactic patterns that exhaust memory, extraction scopes that drag in the world, proof styles that don't scale — each with the tell that identifies it early. Knowledge of where the cliffs are is part of the method, and pretending the cliffs don't exist is how the next person walks off one.

⋆ Aha

Notice the running theme: at every level, the methodology converts *invisible* trust into *visible* trust. Extraction made the code-to-model step visible; two-clause specs made operating envelopes visible; `#print axioms` makes logical debts visible; the trusted-base ledger makes engineering limits visible. Formal verification's deepest product is not certainty — it is **legibility of exactly what remains uncertain**.

>_ Try it yourself

Audit the real thing. In `dalek-ed25519-verified`, run `#print axioms` on `fieldImplementation` and `edwardsImplementation` — confirm the clean trio. Then read `TRUSTED-BASE.md` and match each entry to where it would surface in an audit. Finally, write a deliberately trivial spec for `add`, prove it in one line, and observe that every automated check passes. Keep that file open for one full minute. That minute is the chapter.

Exercises

Exercise 11.1. For each hollow-certificate species, name its detector: (a) `sorry`; (b) smuggled axiom; (c) trivial spec; (d) wrong model. Which two can a CI pipeline catch mechanically, and what CI check would you write for each?

Exercise 11.2. Strengthen this spec until a buggy implementation would fail it: `theorem sub_ok : ∀ a b, ∃ c, sub a b = .ok c`. (List what's missing: bounds hypotheses? bounds propagation? the value equation? Compare with Chapter 9's two-clause shape.)

Exercise 11.3. A vendor's whitepaper says: "Our signature library is formally verified in Lean." Draft the five questions you would send them, in priority order, and the answer you would require for each before relying on the claim. (You now know all five.)

Exercise 11.4. (Discussion) The trusted-base `axiom` for SHA-512 and the smuggled `axiom` for a hard lemma are the *same language feature*. Articulate the difference in one sentence — it is not technical.

⋈ Checkpoint

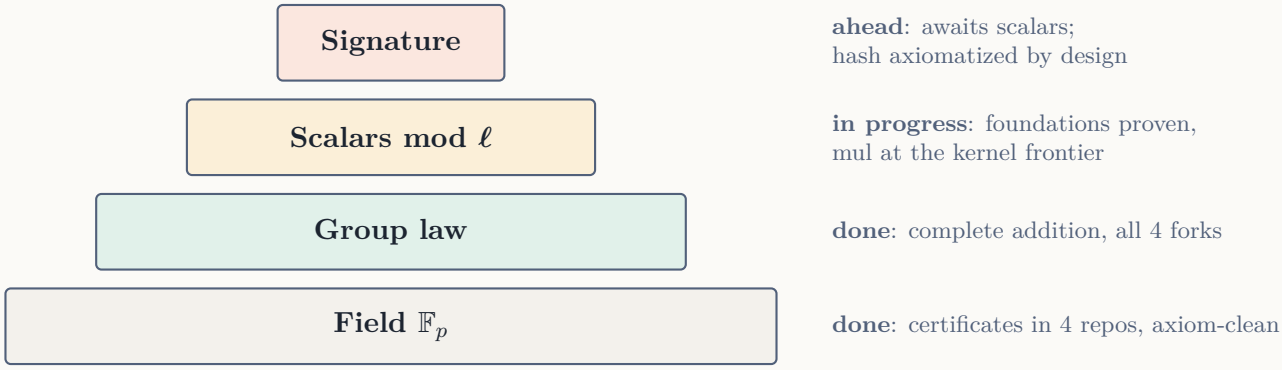
You should now be able to: run and interpret the one-command audit; name the standard three axioms and greet anything else with suspicion; explain why trivial specs and wrong models defeat automation and what defeats *them*; and argue — with conviction — why a declared trusted base is stronger, not weaker, than a claim of totality.

Chapter 12

The Pyramid: From Field to Signature, and Where You Come In

12.1 The view from the field layer

Chapter 10 left us holding a verified field. A signature scheme is still three stories up. This closing chapter walks the remaining layers — what each one *states*, what makes each one *hard*, and where the campaign stands as this book goes to press — then hands you the map and the keys.



12.2 The group law: geometry becomes algebra

An elliptic curve is a set of points (x, y) satisfying an equation; for Ed25519 it is the *twisted Edwards* curve $-x^2 + y^2 = 1 + d\,x^2y^2$ over $\mathbb{F}_p$. The miracle: these points form a *group* under the addition law

$$(x_1, y_1) + (x_2, y_2) \;=\; \left(\frac{x_1y_2 + x_2y_1}{1 + d\,x_1x_2y_1y_2}, \; \frac{y_1y_2 + x_1x_2}{1 - d\,x_1x_2y_1y_2} \right).$$

Two facts make this law a verifier's dream, and both carry Edwards-curve signatures for exactly this reason. First, it is **complete**: for the Ed25519 parameters those denominators are *never zero* — no special cases for doubling, no branch for the identity, hence constant-time-friendly code with no rarely-taken paths for bugs to hide in. (The proof, due to Bernstein and Lange, is a jewel of quiet

algebra: if a denominator vanished, d would have to be a square in $\mathbb{F}_p$ — and it is not, which is a `decide`-scale fact away from primality.) Second, the implementation represents points *projectively* (extended coordinates $(X : Y : Z : T)$, avoiding division entirely) — so the layer has its own denotation, $(X : Y : Z : T) \mapsto (X/Z, Y/Z)$, and its own commuting squares built on the field layer's specs. Same movie, one floor up: the verified group law in the companion repos is precisely the statement that projective point addition implements the rational formula above, all bounds included, for each fork's own extraction.

12.3 Scalars: a second field, and a frontier

The group of curve points has order 8ℓ with $\ell = 2^{252} + 27742\ldots$ prime. Signature arithmetic happens in exponents — multiples of points — so it is arithmetic mod ℓ: a *second* finite field, with its own Rust implementation (radix-52 limbs, Montgomery multiplication) and its own denotation bridge. Nothing conceptually new — which is itself the lesson: the method *scales sideways* without new ideas.

The engineering, however, has a frontier, and this book has told you enough truth to locate it precisely. Scalar Montgomery multiplication mixes 2^{256}-scale coefficients into single certificate steps; this is the kernel-capacity wall of Chapter 10, and it marks the current working edge of the campaign: additions and the foundational constants are certified (including the pleasing theorem that the code's constant `L` *is* ℓ); the multiplication path is a construction site with scaffolding — decomposed lemmas, isolated carry steps — mid-assembly, honestly labeled in-repo.

12.4 The apex: what "verified signature" will say

EdDSA verification accepts (R, s) on message m under key A iff

$$8sB \;=\; 8R + 8\,H(R, A, m)\,A$$

in the curve group (B the base point, H = SHA-512, the 8s absorbing the cofactor). The apex certificate will state: *the extracted verification routine returns true exactly when this equation holds* — given the two declared trusted-base entries you can already predict: SHA-512 as an ideal hash (axiomatized by design — hash function correctness is a different mathematical universe), and the SIMD point-multiplication backends (untranslatable, documented). Everything between those declared boundaries and the field bedrock: kernel-checked, axiom-clean, per fork.

Read that sentence again with Chapter 11 eyes: it is a *smaller* claim than "Ed25519 is verified!" — and that is exactly why you can believe it.

12.5 What you now know, and where to take it

Take inventory. You can read a goal state and drive a proof; you know which decision procedure owns which arithmetic fragment; you can build a denotation bridge and state a two-clause spec; you can certify a prime with a witness tree; you can audit anyone's certificate in one command and four questions. That skill set is not Ed25519-specific — it is the working method of machine-checked mathematics applied to systems, and elliptic curves were merely your first campaign.

Where to go from here, in increasing order of ambition:

- **Read a real proof end-to-end.** `FieldSpec.lean` in `dalek-ed25519-verified`, top to bottom, with this book as the decoder ring. Budget an afternoon; expect the odd hour of humility.
- **Extend the pyramid.** The scalar layer's open lemmas are decomposed, labeled, and waiting; the repos' `CONTRIBUTING` notes state exactly what a finished brick looks like (spec shape, axiom audit, check-script entry). Frontier work, undergraduate-accessible.
- **Verify something of yours.** Pick a 200-line pure function you actually use — a parser, a checksum, a data structure — write its denotation (what does it *mean*?), state the square, prove it. The first solo bridge is the moment this stops being a course.
- **Go deeper into the theory.** *Theorem Proving in Lean 4* (the official text), *Mathematics in Lean* (Mathlib's course), and the Lean Zulip — an unusually welcoming expert community — are the standard next doors.

⋆ Aha

One last reframe, the one this book was secretly about. "Formal verification" sounds like bureaucracy — forms, stamps, compliance. What you actually practiced is closer to *engineering's version of the scientific method*: make the claim precise enough to be falsifiable, then let an incorruptible referee try to falsify it, then publish the referee's report with the assumptions itemized. Cryptography needed that discipline first because its failures are silent and adversarial. It will not need it last.

>_ Try it yourself

The graduation exercise. In the mini-system from `exercises/Ch09.lean`, the file `exercises/Ch12.lean` plants a *deliberate off-by-one carry bug* in a variant `add'` — of exactly the species from Chapter 1: correct on all limb pairs except a thin boundary slice. Your final tasks: (1) write the spec — watch it *refuse to prove*; (2) extract the counterexample from the stuck goal state; (3) confirm by `#eval`; (4) fix the code and finish the proof. That arc — spec, refusal, counterexample, fix, certificate — is the entire profession in miniature. Welcome to it.

⋈ Checkpoint

The book's ending is a beginning, so the final checkpoint is prospective: you should be able to (1) state what each pyramid layer claims and which denotation it rides on; (2) explain to a security engineer why completeness of the Edwards law matters to *code*; (3) locate the current frontier and say precisely why it is hard; and (4) name the next proof *you* intend to write. The authors of the companion repositories left the scaffolding up on purpose.