What "Lean-verified" means, and what it does not

READING LEVEL KIDS · EVERYONE · EXPERTS

Seven Lean demonstrations, each checked by Lean and leanchecker, and by a second kernel checker, nanoda, on a rebuild with Lean 4.35.0; not yet refereed.

Lean-checked 4 · computed 3 · open 1

CHECK IT YOURSELF

The check guide for this book · How to check a result yourself

---

What you can rerun: All seven demonstrations are from-files; each needs Lean. Each result page gives its own rerun value and 5-minute check.

What a match shows: a match tells you that the published files really produce the published numbers. It doesn't tell you that the question was the right one to ask, or that a written argument around the numbers is complete. Each recipe says plainly what its steps show and what they don't. For a result whose evidence is logs (logs-only), a match checks the bookkeeping of the logs, not the search that wrote them.

Cite: Sonny (2026). What "Lean-verified" means, and what it does not (preliminary). Go check yourself. https://gocheckyourself.org/read/lean-verified/everyone/

1. Start here
2. What the words mean
3. What "Lean-verified" means
4. What "Lean-verified" does not mean
5. Five ways to get "2 + 2 = 5" past a build
6. Kernel bugs, and why a second checker helps
7. Statement
8. Theorems in this book

9. Check it yourself: what "Lean-verified" covers
10. Files and fingerprints
11. Trust and verification
12. Lean sources: how to check them
13. People and works
14. Files and rights

1 Start here

Sonny

What this book shows

- My result. Seven small Lean 4.34.1 files (Lean has a small judge inside it, the kernel, that checks every step of a proof against a fixed list of rules), with what Lean actually printed. Five of them make a clean-looking build say $2 + 2 = 5$, and a different check catches each kind of trick.
- What a clean build tells you: that Lean built the file without an error, given the definitions and axioms (starting assumptions that are not proved) in scope, and nothing more. It does not by itself show that the kernel checked every declaration: a switch can turn that check off and the build still passes. An added axiom or a sorry (a placeholder for a missing proof) shows in the axiom report (Lean's list of the assumptions a theorem rests on); a switch that turns kernel checking off is caught by leanchecker, which replays every declaration through the kernel; a true theorem about a one-number system or a redefined plus sign passes every automatic check and is caught by reading the statement, by no automatic check.
- Why it matters. More and more mathematics, and more claims made by computers, carry a "checked in Lean" label. Knowing which check catches which problem lets a reader ask the right question of a result.
- In Lean: all seven files. Four are proved in Lean, as written, and accepted by leanchecker and by a separately written kernel, nanoda (a second checker for Lean proofs, written by different people). Two

depend on an added or placeholder assumption, which the axiom report shows and nanoda refuses. One is rejected by the re-check.
- Every theorem in this book is listed in the theorem library.

The question

When a mathematical result is called "Lean-verified", what has actually been checked, and can a file that Lean accepts still say something false?

Lean is a program for writing mathematics so exactly that a small piece of software, the kernel, can check every step. A "Lean-verified" label is a strong claim and is easy to over-read. The people who build Lean ask the same question: the Lean Language Reference has a chapter, "Validating a Lean Proof", on how to judge a result you did not write. In July 2026 the question stopped being only about readers' care: a user reported a bug in the kernel itself, in an issue titled "Kernel accepts wrong-structure projections, allowing an axiom-free proof of False" (issue 14576 in the Lean repository, 28 July 2026), and it was fixed the same day.

This book answers the question for one narrow case: what a clean build does and does not tell you, shown on seven small Lean files.

The short answer is that a clean build tells you that Lean built the file without an error, given the definitions and axioms in scope, and nothing more; a separate replay with leanchecker is what shows the kernel accepted every declaration. Five of the files below make a clean-looking build say $2 + 2 = 5$, and a different check catches each kind of trick.

Why it matters

More and more mathematics, and more and more claims made by computers, are presented with a "checked in Lean" label. A reader who takes the label to mean "the English sentence I just read is proved" can be wrong in four ways: the Lean sentence may say something else, a definition may mean something else, part of the argument may sit outside Lean, or the checking program may

have a bug. Knowing which check catches which problem lets a reader ask the right question of a result.

What answering it would help with

- A reader of a "Lean-verified" claim may use the same short list of checks (the axiom report, a re-check through the kernel, a second kernel, reading the statement) to see how far the label can be trusted.
- Authors may be able to say which of these checks a result has passed, in the same words every time, so that labels become comparable.
- Teachers may use the seven files as small, runnable examples of what a formal check does and does not cover.

What our research shows

What this is

A SHORT BOOK ON WHAT A "LEAN-VERIFIED" LABEL COVERS, WITH SEVEN SMALL LEAN 4.34.1 FILES THAT SHOW HOW A CLEAN-LOOKING BUILD CAN STILL SAY "2 + 2 = 5" (SONNY, 2026).

Still open

WAYS A LEAN RESULT CAN MISLEAD BEYOND THE FOUR THIS BOOK COVERS. THE BOOK EXPLAINS THE LABEL; IT REVIEWS NO ONE'S PROOFS.

Checked by

EACH FILE WAS BUILT WITH LEAN 4.34.1 AND RE-CHECKED WITH LEANCHECKER; THE SAME FILES BUILT WITH LEAN 4.35.0 WERE ALSO GIVEN TO A SEPARATELY WRITTEN KERNEL, NANODA. SUPPORTING FILES ARE AVAILABLE ON REQUEST.

<!--BOX-->

What's in this book

Counts are computed from the book's statement file (its table of seven demonstrations), except the open item, which comes from chapter 4 and the check guide.

Status	Count
Proved and checked in Lean (the Lean statement as written, accepted by the kernel; the meaning of the statement is not claimed)	4 (the honest file, the one-number system, the different plus sign, the faulty definition)
Accepted by the build but dependent on an added or placeholder assumption, visible in the axiom report	2
Stored with kernel checking turned off, rejected by the re-check	1
Open	1 (a run with lean4lean, which could not read the compiled files here)

Item	Count
Chapters	4
Statement files	2 (a table and its machine-readable form)
Written check recipes	7, one per demonstration, plus an optional second-kernel recipe and the fingerprint check
Lean files	7
Fingerprint list	1 (SHA-256 of the seven Lean files and the build files)

<!--/BOX-->

Yesterday, today, tomorrow

Yesterday. Leonardo de Moura and the Lean community built Lean. Kim Morrison and Sebastian Ullrich wrote leanchecker (earlier lean4checker), which replays a compiled proof through the kernel to detect tampering. Mario Carneiro wrote lean4lean and ammkrn wrote nanoda, two checkers that re-implement the kernel separately. The Lean reference manual has a chapter, "Validating a Lean Proof", on how to judge a result you did not write. In July 2026 a bug in the kernel itself was reported and fixed.

Today. I ran seven small demonstrations on Lean 4.34.1, on one Windows PC, and pasted what Lean actually printed. An added axiom builds and is shown by the axiom report. A sorry builds with a warning and is shown by the axiom report. A true theorem about a one-number system, or about a redefined plus sign, passes every automatic check and is caught only by reading the statement. A switch that turns kernel checking off lets a false statement build with "no axioms", and leanchecker rejects it. The separate

kernel, nanoda, accepted the files that only a reader can catch, and did not accept the one with checking switched off.

Tomorrow. The same checks can be run by anyone on these files in a few minutes. lean4lean, a second separately written kernel, could not read the compiled files here, so only nanoda was used; running lean4lean too would be a natural next step. Other readers may know more ways a Lean result can mislead, and those can be added here with the same format: the file, the real output, and the check that catches it.

Check it yourself

In this book: the project file that builds the seven Lean files, a SHA-256 fingerprint (a 64-character code computed from a file; change one bit and the code changes) for each of those files, and a written recipe for each demonstration in the check guide. The seven Lean files are in the demos folder, and chapter 3 prints each one in full. A recipe gives the exact command, the exact output to expect, and the standard tool that produces it: Lean's build tool, the axiom report, leanchecker, and optionally nanoda. To try one, install Lean through elan (Lean's installer), go to the lean folder and run:

```
lake build Demo.Honest
lake env leanchecker Demo.Honest
```

The build prints that the theorem does not depend on any axioms, and leanchecker accepts silently.

How I direct this work, and my own tools, are not part of this book. Everything you need to check the results is public: the results, their certificates and fingerprints, the Lean proofs, and a written recipe for each check. In this book the results are the seven Lean files and what each check printed; there are no certificate files beyond the fingerprint list of those files.

About this research: https://gocheckyourself.org/about/#how

The chapters

- What "Lean-verified" means: the guarantee, step by step.
- What "Lean-verified" does not mean: four things the label leaves out.
- Five ways to get $2 + 2 = 5$ past a build: the seven files, with real output.
- Kernel bugs, and why a second checker helps: the case where the check itself is wrong.

The statement of the seven demonstrations, the check guide, the plain words page, the children's version and the credits page complete the book.

We stand on the shoulders of giants: Lean, its checkers and the people who found and fixed kernel bugs are other people's work, credited on the credits page.

Where to go next

- Which one of the seven files would you have missed if you only read the build log? See Five ways to get $2 + 2 = 5$ past a build.
- If the kernel itself can be wrong, how much does a second kernel help? See Kernel bugs, and why a second checker helps.

Go check yourself.

Have a question about this? Ask it

2 What the words mean

What the words mean

The question: what do the words in a "Lean-verified" claim mean, in plain words and for experts? The answer in brief: each is one link in a chain, from the kernel that judges a proof, to the axiom report, to a second look, to the meaning of the sentence itself.

"Lean-verified" is a short label for a long chain of steps. Each idea below has a plain version and a version for experts.

The kernel

In plain words. Lean has a small judge inside it. You hand it a statement and a proof, and it checks every step against a fixed list of rules. If it says yes, the proof is correct for the statement as written, using the definitions and starting assumptions that are in scope. It does not decide whether the statement is the one you meant.

For experts. Lean 4's kernel is a type checker for a dependent type theory with inductive types, universes, quotients and definitional proof irrelevance. A theorem is accepted when its value has the stated type in the current environment. The kernel's input is an expression produced by the elaborator, so everything before that point (parsing, notation, instance resolution, tactics) is outside the trusted base, and everything after it (the kernel implementation, the compiler for native evaluation, the toolchain) is inside it.

Axioms and the axiom report

In plain words. An axiom is a starting assumption that is not proved. Lean uses three standard ones. You can ask Lean to list the assumptions a theorem rests on. If the list is empty or holds only the three standard names, nothing extra was assumed. If it shows the special placeholder axiom, someone left a gap. If it shows a name nobody recognises, someone added an assumption, and the theorem is true only if that assumption is. The list reports the stored assumptions; a separate kernel replay is another check.

For experts. The axiom report collects the axioms reachable from a constant in the environment, including the placeholder axiom, axioms introduced by native evaluation (names ending in a native marker), and any user-declared axiom. It does not re-run the kernel, so a declaration added without checking (through the kernel-skip option, or the function that adds a declaration without checking it) is reported with whatever axioms its stored value mentions, which can be none.

A second look

In plain words. There are two kinds. One is Lean looking again at its own stored work, with the same judge, which catches work that was stored without being judged. The other is a different judge, written by different people in a different language, which is given the proof as plain text. If both say yes, a bug in one is unlikely to be in the other. It is still not a guarantee.

For experts. leanchecker loads the compiled module files and replays each declaration through the same kernel, with its fresh mode replaying imports as well. It does not defend against a kernel bug or a malformed compiled file. External checkers (nanoda in Rust, lean4lean in Lean) read the lean4export text format and use separate kernel implementations. The reference manual's strongest recommendation, comparator, builds in a sandbox and then checks with Lean's kernel and an external checker. A kernel bug affecting all checkers used remains an assumption.

A true sentence that means something else

In plain words. A proof is a proof of the exact sentence written in Lean, with the exact meanings of the symbols in scope. If someone redefines the plus sign, or uses a number system with a single number, or leaves a condition out of a definition, then Lean will happily confirm a sentence that looks like something else. The only defence is for a person to read what the sentence and its definitions really say.

For experts. Statement fidelity: elaboration of $2 + 2 = 5$ depends on the instances in scope for addition and numerals and on the expected type. The statement $(2 + 2 : \mathrm{Fin}\ 1) = 5$ is true by computation because the type with one element has a single value. A local addition instance on the natural numbers shadows the core instance within its scope and makes $2 + 2 = 5$ hold by reflexivity. A definition of "prime" that omits the condition $2 \leq$ n admits 1. In all three the kernel's verdict is correct and the axiom report is clean.

The switch that turns checking off

In plain words. Lean has a setting, meant for experts debugging Lean itself, that tells it to store a declaration without sending it to the judge. A file that sets it can store a false statement, and the build and the assumption list both look clean. Looking again with leanchecker sends the stored statement to the judge, which refuses it. The setting can be placed in the build file, so searching the build file matters as much as searching the Lean files.

For experts. The kernel-skip option makes the kernel's add-declaration entry point store the declaration without checking it. Set in the options section of the project's build file, it applies to every module of the library, and no Lean source file mentions it. The elaborator still checks ordinary terms, so a false theorem needs a direct call that adds a declaration, from meta-code, to be stored. Lean's reference manual classes such use as dishonest.

Where to go next

- Which of these words are shown at work, with real Lean output? See Five ways to get $2 + 2 = 5$ past a build.
- What does the label leave unsaid? See What "Lean-verified" does not mean.

Experts: the files and checks are here: check guide.

Yesterday, today, tomorrow

Yesterday: Lean was built so that a very small program, the kernel, is the judge of every proof, and so that a reader need not trust the person who wrote it. Its authors also wrote tools to look again at a finished proof, and two other groups wrote separate kernels that can re-check a proof exported from Lean.

Today: I ran small examples to show what the label covers. An extra axiom, an unfinished proof, a redefined plus sign, a number system with one number and a switch that turns checking off each give a Lean file that "proves" $2 + 2$

$= 5$ (or a related falsehood). In each case the book shows what Lean printed and which check catches it.

Tomorrow: The same questions can be asked of any Lean result. Readers who know other ways a Lean result can mislead are asked to say so.

3 What "Lean-verified" means

What "Lean-verified" means

The question

When a result is called "Lean-verified", what exactly has been checked?

Why it matters

Results are more and more often described as "Lean-verified". A reader needs to know what that label promises before relying on it.

What answering it would help with

A shared, exact meaning of the label lets readers and authors say what was checked and what was not.

What our research shows

The answer in one line: given the definitions and axioms in scope, Lean's kernel accepted this proof of this statement. Nothing more is promised.

The idea in one paragraph

Lean is a proof assistant. You write a mathematical statement and a proof in a precise language. A small program inside Lean, the kernel, then checks every step of the proof against Lean's rules, starting from the definitions and axioms (starting assumptions) that are in scope. If the kernel accepts, the statement follows from those definitions and axioms by the rules. That is the guarantee, and it is a strong one: it does not depend on trusting the person

who wrote the proof, or the tactics (helper programs) that helped build it, because the kernel's verdict is the one that counts at the end.

What is checked, step by step

1. The statement is read. Lean turns the text of a theorem into a precise internal form. For example $2 + 2 = 5$ is read as a statement about natural numbers (0, 1, 2, ...) unless something says otherwise. Which symbol means what is decided by the definitions in scope. The kernel does not choose these; it only checks what it is handed.
2. The proof is checked. The kernel checks that the proof really proves that statement. A proof that does not is rejected with an error.
3. The axioms can be listed. A command prints the axioms a theorem depends on, directly or through other theorems:

```
#print axioms theoremName
```

Lean's reference manual says a clean result uses only three standard axioms:

```
propext
Classical.choice
Quot.sound
```

Anything else tells you something. The special axiom sorryAx means a sorry (a placeholder for a missing proof) is somewhere underneath. A name ending in a native-computation marker means the proof used native computation. Any other name is an axiom someone added, and the theorem is valid only if that axiom is.

The axiom report reads what is stored. It does not check anything again.

Checking again

Two kinds of re-check exist.

- A replay with the same kernel. leanchecker ships with Lean. After a build it reads the compiled files and sends every stored declaration

through the kernel again. This catches a declaration that was stored without ever going through the kernel (chapter 3 shows one). Its own documentation says it is not an external verifier, only a tool for detecting tampering with the environment. It uses the same kernel code, so it cannot catch a mistake in the kernel itself.

- A different kernel. Lean can export a proof as plain text, and other programs can check that text with a kernel written separately: nanoda (written in Rust) and lean4lean (the kernel written again, in Lean itself). If two kernels written by different people agree, a bug in one is unlikely to be present in the other. Chapter 4 explains how far that helps.

The Lean reference manual lists these as an escalating sequence: the check marks in the editor, then the axiom report, then re-checking, then external checkers. Which step is enough depends on whether you are guarding against honest mistakes or against someone who is trying to mislead.

Yesterday, today, tomorrow

Yesterday. Leonardo de Moura and the Lean community built Lean around a small kernel, so that trust rests on a short piece of code and not on every proof author. Kim Morrison and Sebastian Ullrich wrote the replay tool that became leanchecker, Mario Carneiro wrote lean4lean, and ammkrn wrote nanoda. The Lean reference manual has a chapter, "Validating a Lean Proof", that sets out the checks in order.

Today. I read "Lean-verified" as a claim with a precise meaning: given these definitions and these axioms, the kernel accepted this proof of this statement. Chapter 3 shows, with real output, what that does and does not rule out.

Tomorrow. Each Lean result I publish can say which of the checks above it passed, in the same words every time. A result that passed only the checks in steps 1 and 2 is labelled that way.

Check it yourself

The checks named in this chapter (the build, the axiom report and leanchecker) are run on seven small files in the check guide, each with its command and expected output.

Where to go next

- What can a "Lean-verified" result still get wrong even when every step above passes? See What "Lean-verified" does not mean.
- What does it look like when a clean build says $2 + 2 = 5$? See Five ways to get $2 + 2 = 5$ past a build.

4 What "Lean-verified" does not mean

What "Lean-verified" does not mean

The question

If a proof passes every Lean check, what can still be wrong?

Why it matters

A result that passes every check can still be read as promising more than it does. The gaps are where a reader can be misled.

What answering it would help with

Knowing the gaps tells a reader what still has to be read by a person: the statement, the definitions, and anything outside Lean.

What our research shows

The answer in one line: the match between the Lean statement and the English sentence, the meaning of the definitions, anything computed outside Lean, and the tools themselves.

The label covers a proof of a Lean statement. The distance between that and the English sentence in a headline is where mistakes and misreadings live. Lean's reference manual draws the same line: it separates the question of whether a theorem has a valid proof from the question of what the theorem statement means.

1. It does not mean the Lean statement says what the English sentence says

A theorem is a statement plus its proof. The kernel checks the proof of the statement that is written. If the statement is a little off, the proof can be perfect and the headline still wrong. Examples of "a little off": a hypothesis left out, a condition dropped from a definition, a function defined to return 0 outside the range someone cared about, a type that allows a case the paper excludes. Chapter 3 has two short examples with real output: a definition of "prime" that lets 1 in, and $2 + 2 = 5$ read in a number system with one number.

2. It does not mean the definitions mean what readers assume

Lean lets anyone define new words, new notation and new meanings for existing symbols, and many of these are in scope only inside one file or one section. A reader who sees a plus sign assumes ordinary addition. Chapter 3 shows a file where the plus sign has been redefined for the natural numbers, and every automatic check is clean. The defence is to read the definitions the statement depends on, and not only the statement.

3. It does not cover anything outside Lean

A Lean proof can use the result of a computation only if that computation is done inside Lean's logic, or if Lean's native evaluation is used, which shows up as an extra axiom. A numerical experiment that was run outside Lean, in another language or on other hardware, is not covered, even if a Lean theorem later quotes its output. The same goes for the correctness of the machine the checker ran on. When a result mixes Lean with such computations, the label should say which part is which.

4. It does not mean the toolchain was not tampered with

The kernel is a program, and so are the compiler, the build tool and the checkers. A file can switch the kernel's checks off (the kernel-skip option, shown in chapter 3), and the option can sit in the build file and not in any Lean source. Lean's manual calls code that does this dishonest, as opposed to honest. Kernels can also contain bugs (chapter 4). Re-checking with the kernel again and with a second kernel narrows these risks (chapters 3 and 4). It does not remove them.

What this adds up to

"Lean-verified" is a good answer to one question: given these definitions and these axioms, is this a correct proof of this statement? It is silent on the other questions. Is the statement the one in the headline? Are the definitions the ones a reader assumes? Was everything checked? Is the checking program itself correct? A careful reader needs a separate look for each.

Yesterday, today, tomorrow

Yesterday. The Lean reference manual separates whether a theorem has a valid proof from what its statement means, and the documentation of leanchecker says it is not an external verifier. A 2026 paper by Bastounis, Circelli and Hansen states that a machine-checked Lean version of a written argument can differ from it, and gives examples in which an AI translation into Lean does not match the written proof it was meant to formalise. The AXLE paper by Xin, Schneidman, Cummins, Ram, Ganesh and Limperg (arXiv:2606.26442, June 2026) lists the same family of cases as chapter 3, ways for a file to compile without proving the intended statement (sorry, an axiom outside a short allowed list, a restated theorem, unsafe declarations), and compares the checkers lean4checker, Comparator and SafeVerify.

Today. I keep four claims apart in every Lean result: the proof of the Lean statement, the match between the Lean statement and the English one, the work done outside Lean, and the soundness of the tools. The label "Lean-verified" is used only for the proof of the Lean statement.

Tomorrow. Where a result depends on a match between a Lean statement and an English one, a short table can set the two side by side, line by line, for

a reader or an expert to check.

Check it yourself

Three of the seven demonstrations in the check guide pass every automatic check and are caught only by reading. Running them shows the gaps this chapter describes.

Where to go next

- How does a statement that is true in Lean say something other than the sentence you read? See Five ways to get $2 + 2 = 5$ past a build.
- If the kernel is the one thing everything rests on, what happens when it has a bug? See Kernel bugs, and why a second checker helps.

5 Five ways to get "2 + 2 = 5" past a build

Five ways to get "$2 + 2 = 5$" past a build

The question

Can a Lean file that ends in "Build completed successfully" still say something false, such as $2 + 2 = 5$, and what catches each way of doing it?

Why it matters

A build that ends in success is often taken as the whole check. These files show what that message does and does not establish.

What answering it would help with

Seeing each way side by side lets reviewers choose which checks to run and which things to read.

What our research shows

The answer in one line: yes, in five ways. Four of them are caught by reading something (the axiom report, a warning, the statement, the definitions) and

one is caught only by replaying the stored proof through the kernel again.

Each section below shows a short Lean file, the real output Lean printed for it, and the check that catches it. Everything was run, and run again after edits to the comments of six files and to one declaration name, with Lean 4.34.1 (commit 5045d005) on one Windows PC. The output shown is from the second run. The seven files are in the book's demos folder, and each section below prints its file in full. Three tools are used throughout:

- The build command runs Lean on the file. If it ends with "Build completed successfully", Lean accepted the file.
- The axiom report is a line inside each file. During the build, Lean prints the assumptions that theorem rests on.
- leanchecker replays every stored declaration through the kernel again. Its fresh mode also replays everything the file imports.

The commands, run from the lean folder of this book, are:

```
lake build Demo.Honest
lake env leanchecker Demo.Honest
lake env leanchecker --fresh Demo.Honest
```

leanchecker prints nothing when it accepts a file and stops with a message when it rejects one. The output shown below is what Lean printed, without timings.

An honest file

```
-- A true statement with an ordinary proof. Every check should accept this
theorem Demo.honest : 2 + 2 = 4 := rfl

#print axioms Demo.honest
```

```
ℹ Built Demo.Honest
info: ../demos/Demo/Honest.lean:4:0: 'Demo.honest' does not depend on any a
Build completed successfully (2 jobs).
```

Both leanchecker runs accept. The theorem rests on no axioms, the build passes and both re-checks accept. This is what a clean result looks like.

An added axiom

```
-- Added axiom: an extra axiom. Lean accepts any axiom you declare; that is
-- The build passes. `#print axioms` lists the extra axiom, so the claim is
axiom Demo.bad : 2 + 2 = 5

theorem Demo.two_plus_two_is_five : 2 + 2 = 5 := Demo.bad

#print axioms Demo.two_plus_two_is_five
```

```
ℹ Built Demo.AddedAxiom
info: ../demos/Demo/AddedAxiom.lean:7:0: 'Demo.two_plus_two_is_five' depend
Build completed successfully (2 jobs).
```

Both leanchecker runs accept.

What happened. Lean accepted the file. An axiom is a starting assumption, and Lean lets anyone declare one. The build passes, and leanchecker accepts too, because adding an axiom is not an error: nothing was bypassed.

How it is caught. The axiom report says the theorem depends on the added axiom. Anything other than the three standard axioms is a reason to stop and read. The nanoda check described in chapter 4 was set to permit only the three standard axioms, and it refused this file on that rule (output below).

So this "certificate" for $2 + 2 = 5$ is valid in exactly one sense: if you grant the added axiom, the theorem follows. The axiom report says so in plain view.

An unfinished proof (sorry)

```
-- Unfinished proof: `sorry` stands in for a missing proof.
-- The build passes with a warning. `#print axioms` lists `sorryAx`.
theorem Demo.two_plus_two_is_five_sorry : 2 + 2 = 5 := by sorry

#print axioms Demo.two_plus_two_is_five_sorry
```

```
⚠ Built Demo.UsesSorry
warning: ../demos/Demo/UsesSorry.lean:3:8: declaration uses `sorry`
info: ../demos/Demo/UsesSorry.lean:5:0: 'Demo.two_plus_two_is_five_sorry' d
Build completed successfully (2 jobs).
```

Both leanchecker runs accept.

What happened. The word sorry tells Lean "pretend this step is proved". The build passes, with a warning, and leanchecker accepts, because Lean stored the placeholder as a special axiom.

How it is caught. By the warning, and by the axiom report, which lists the special axiom. A build log that is searched only for the word "error" will miss it, so read warnings and the axiom report, not only error lines. Nanoda refused this too, under the same three-axiom rule.

A true theorem about a different meaning

Here nothing is wrong with the proof. The kernel, the axiom report and both re-checks are all clean. What is different is what the sentence means. There are three variants.

A number system with one number

```
-- One-number system: a true theorem about a different meaning.
-- `Fin 1` is a number system with exactly one number, 0. Every number writ
-- so 2 + 2 and 5 both mean 0. The proof is by `decide`, in core Lean, with
-- Build, axiom report and re-check are all clean. Only reading the stateme
theorem Demo.small_number_system : (2 + 2 : Fin 1) = 5 := by decide

#print axioms Demo.small_number_system
#check (5 : Fin 1)
#eval (5 : Fin 1)
#eval (4 : Fin 1)
```

```
ℹ Built Demo.SmallerNumberSystem
info: ../demos/Demo/SmallerNumberSystem.lean:7:0: 'Demo.small_number_syster
info: ../demos/Demo/SmallerNumberSystem.lean:8:0: 5 : Fin 1
info: ../demos/Demo/SmallerNumberSystem.lean:9:0: 0
info: ../demos/Demo/SmallerNumberSystem.lean:10:0: 0
Build completed successfully (2 jobs).
```

Both leanchecker runs accept.

What happened. The numbers below 1 are only the number 0. Writing 5 in that system means 5 modulo 1, which is 0, and the output confirms it: evaluating 5 prints 0, and so does evaluating 4. Both sides of $2 + 2 = 5$ are 0, so the proof by computation succeeds. The axiom report shows one of the three standard axioms.

How it is caught. Only by reading the statement: the number-system annotation in it is the whole difference between this and ordinary arithmetic. Neither the kernel nor the axiom report is able to say "this is not the sum you meant". Nanoda accepted the exported proof too, which is correct: the proof is valid.

The same symbols, a different plus sign

```
-- Different plus sign: the same symbols, a different meaning.
-- A local instance changes what `+` does on natural numbers inside this na
-- it adds and then adds one more. Under that reading `2 + 2 = 5` is true b
-- The build and the axiom report are clean. Only reading the definitions i
-- `#synth Add Nat` asks Lean which addition it would use here: the local o
namespace Demo

local instance : Add Nat := ⟨fun a b => Nat.succ (Nat.add a b)⟩

theorem redefined_plus : 2 + 2 = 5 := rfl

#print axioms redefined_plus
#eval 2 + 2
#synth Add Nat

end Demo

-- Outside the namespace the core addition is back, and the arithmetic is u
#synth Add Nat
theorem Demo.ordinary_plus : Nat.add 2 2 = 4 := rfl
```

```
ℹ Built Demo.RedefinedPlus
info: ../demos/Demo/RedefinedPlus.lean:12:0: 'Demo.redefined_plus' does not
info: ../demos/Demo/RedefinedPlus.lean:13:0: 5
info: ../demos/Demo/RedefinedPlus.lean:14:0: instAddNat_demo
info: ../demos/Demo/RedefinedPlus.lean:19:0: instAddNat
Build completed successfully (2 jobs).
```

Both leanchecker runs accept.

What happened. Inside the namespace, a local definition changes what the plus sign means on natural numbers: add, then add one more. Under that meaning $2 + 2 = 5$ is true by direct computation, and evaluating $2 + 2$ prints 5. The axiom list is empty. Outside the namespace, the core addition is back.

How it is caught. By reading the definitions that are in scope. The two lookups of which addition Lean would use show it: inside the namespace Lean would use the local one, and outside it uses the core one. This is why a careful

review covers notation, local definitions and options, not only the theorem line.

A definition that lets 1 be prime

```
-- Faulty definition: a definition that does not mean what its name suggest
-- `MyPrime` forgets the condition 2 ≤ n, so 1 counts as "prime".
-- Build, axioms and re-check all pass. Only a reader of the definition cat
def MyPrime (n : Nat) : Prop := ∀ d : Nat, d ∣ n → d = 1 ∨ d = n

theorem Demo.one_is_prime : MyPrime 1 := by
  intro d h
  exact Or.inl (Nat.dvd_one.mp h)

#print axioms Demo.one_is_prime
```

```
ℹ Built Demo.WrongDefinition
info: ../demos/Demo/WrongDefinition.lean:10:0: 'Demo.one_is_prime' depends
Build completed successfully (2 jobs).
```

Both leanchecker runs accept.

What happened. The definition of "prime" leaves out the usual condition that a prime is at least 2, so 1 qualifies. The proof is correct for the definition as written. Build, axiom report (one standard axiom) and both re-checks are clean.

How it is caught. Only by someone reading the definition and noticing the missing condition.

Switching the kernel's check off

```
-- Kernel check off: a false "theorem" that skips the kernel check, for tea
-- It adds `False`, "proved" by `True.intro`, while the option `debug.skipK
-- tells Lean not to send the declaration to the kernel type checker.
-- `lake build` accepts this file and `#print axioms` reports no axioms.
-- `leanchecker` replays the declaration through the kernel and rejects it.
import Lean
open Lean Elab Command

set_option debug.skipKernelTC true in
run_cmd liftCoreM <| addDecl (Declaration.thmDecl
  { name := `Demo.unchecked, levelParams := [], type := mkConst ``False, va

#print axioms Demo.unchecked

-- Anything follows from `False`, so this "proves" a plainly wrong equation
theorem Demo.wrong : 2 + 2 = 5 := Demo.unchecked.elim

#print axioms Demo.wrong
```

```
ℹ Built Demo.SkipKernelTC
info: ../demos/Demo/SkipKernelTC.lean:13:0: 'Demo.unchecked' does not deper
info: ../demos/Demo/SkipKernelTC.lean:18:0: 'Demo.wrong' does not depend or
Build completed successfully (2 jobs).
```

```
$ lake env leanchecker Demo.SkipKernelTC
leanchecker found a problem in Demo.SkipKernelTC
uncaught exception: while replaying declaration 'Demo.unchecked':
(kernel) declaration type mismatch, 'Demo.unchecked' has type
  True
but it is expected to have type
  False
```

The fresh run stops with the same message, without its opening line.

What happened. The kernel-skip option tells Lean to store a declaration without sending it to the kernel. The file uses it to store a "theorem" whose statement is False and whose proof is a proof of True. Anything follows from False, so the next line proves $2 + 2 = 5$. The build passes, and the axiom report says both theorems "do not depend on any axioms".

How it is caught. leanchecker replays the stored declaration through the kernel and stops with a type mismatch: it has type True but should have type False. Both the plain run and the fresh run reject the file. The kernel was not fooled; it was never asked at build time, and the re-check asked it.

With the option left out, the kernel refuses the same declaration at build time, as it should. The option can also be set in the build file of a project, in which case no Lean file mentions it, so a search of the Lean files alone finds nothing; search the build files too. Lean's reference manual describes use of this option as dishonest, not as a mistake. An ordinary false proof is not let through by this option: Lean's front end still checks terms as they are written, and the file needs a direct call that adds a declaration from meta-code (Lean code that writes Lean code) to get the false declaration stored.

What a second kernel said

Chapter 4 describes the setup. For each file, the theorem was exported from the Lean 4.35.0 build of the same file and checked with nanoda, a kernel written separately in Rust, set to permit only the three standard axioms.

```
nanoda_bin 0.4.19
Honest: export Demo.honest, check with nanoda
Checked 38 declarations with no errors
AddedAxiom: export Demo.two_plus_two_is_five, check with nanoda
Error: export file declares unpermitted axiom "Demo.bad"
UsesSorry: export Demo.two_plus_two_is_five_sorry, check with nanoda
Error: export file declares unpermitted axiom "sorryAx"
SmallerNumberSystem: export Demo.small_number_system, check with nanoda
Checked 294 declarations with no errors
RedefinedPlus: export Demo.redefined_plus, check with nanoda
Checked 38 declarations with no errors
WrongDefinition: export Demo.one_is_prime, check with nanoda
Checked 198 declarations with no errors
SkipKernelTC: export Demo.wrong, check with nanoda
thread 'main' panicked at src\tc.rs:955:71:
assertion failed: self.def_eq(u, v)
```

- The honest file and the three different-meaning files: accepted. The proofs are valid, so a second kernel agrees. It cannot say the statements

mean something else.

- The added axiom and the unfinished proof: refused because the export contains an axiom outside the permitted three. That is the axiom report doing its job in a second tool.
- Kernel check switched off: nanoda did not accept. It stopped with an internal assertion failure about two terms being equal (exit code 101) and printed no success line. This is a program stopping, not a polished verdict; the book reads it as "not accepted" and no more. It is consistent with the type mismatch leanchecker reports.

All together

Demo	**Build**	**Axiom report**	**leanchecker, fresh**	**nanoda**	**Caught by**
Honest file	passes	none	accepts	accepts	nothing needed
Added axiom	passes	the added axiom	accepts	refuses (axiom rule)	the axiom report
Unfinished proof	passes, warning	the special placeholder axiom	accepts	refuses (axiom rule)	warning and axiom report
One-number system	passes	one standard axiom	accepts	accepts	reading the statement
Different plus sign	passes	none	accepts	accepts	reading the definitions
Faulty definition	passes	one standard axiom	accepts	accepts	reading the definition
Kernel check off	passes	none	rejects	does not accept	the re-check through the kernel

Only the last row is caught by an automatic re-check of the proof. Three rows are caught only by a person.

Yesterday, today, tomorrow

Yesterday. The Lean reference manual already lists axioms, sorry and the kernel-skip option, and says what each check protects against. Three demonstrations in this chapter, the honest file, the kernel-skip file and the

faulty-definition file, come from an earlier draft by Sonny, run earlier; all seven were run again for this book, and once more after small edits.

Today. Sonny re-ran all seven files, and again after edits to their comments and one declaration name, and pasted what Lean printed. Four different checks catch the five ways: the axiom report, the build warning, a person reading, and a re-check through the kernel.

Tomorrow. Other ways to mislead can be added in the same form: a short file, the real output, and the check that catches it. A run with lean4lean, which could not be used here, would add a second separately written kernel.

Check it yourself

Every command and its expected output is in the check guide, one recipe per demonstration. The seven files are in the demos folder, and their fingerprints are in the certificates folder.

Where to go next

- If the kernel itself has a bug, which of these checks still help? See Kernel bugs, and why a second checker helps.
- What does a "Lean-verified" label leave unsaid even when every check passes? See What "Lean-verified" does not mean.

6 Kernel bugs, and why a second checker helps

Kernel bugs, and why a second checker helps

The question

What if the Lean kernel itself has a bug, so that a false statement gets through?

Why it matters

Every Lean check in the end trusts the kernel. A kernel bug would let a false statement through every check that relies on it.

What answering it would help with

Knowing what a second, separately written kernel adds lets a reader decide how much checking an important result needs.

What our research shows

The answer in one line: it has happened (a 2026 report, fixed the same day), and a second checker written separately is evidence against such a bug, not a guarantee.

Everything in chapter 3 is something a careful reader can catch with the right check. This chapter is about the case where the check itself is wrong.

The kernel is a program

The kernel is small, but it is still software, and software has bugs. Lean's reference manual says so itself: its list of assumptions that remain even after its strongest check includes that no implementation bug affects all of the checkers used at the same time.

A 2026 example

This section is taken from the public sources named in CREDITS.md, which were read for this book. I did not reproduce the bug, and no exploit file is shipped in this book.

- On 28 July 2026 a user of GitHub with the name kiranandcode opened issue 14576 in the Lean repository, titled "Kernel accepts wrong-structure projections, allowing an axiom-free proof of False". The report says the example uses the ordinary, checked path for adding declarations, and does not use sorry, the kernel-skip option or a modified compiled file. It also says the report arose while reviewing a published proof of a famous open conjecture.

- The same day, Leonardo de Moura's pull request 14577, which adds a missing check in the kernel's handling of inductive declarations, was merged, and the issue was closed. Its description says that a nested inductive datatype whose parametric arguments were ill typed could be accepted, because those arguments escaped type checking.

- A post to the oss-security mailing list by Jonathan Brossard on 2 August 2026 describes the bug as a kernel soundness bug, lists the affected versions as all Lean 4 versions up to and including 4.31.0 and nightly builds through 27 July 2026, and says that leanchecker also accepts the $0 = 1$ proof it demonstrates. The post says a different forged proof, a disproof of the Collatz conjecture, was accepted by the Lean kernel and by the independent checker nanoda.
- The Lean 4.32.2 release notes (28 July 2026) say that the issue was discovered by Ramana Kumar and reported by Kiran Gopinathan, that "the bug can be exploited even when using comparator", and that nanoda "does not suffer from the same bug" but could be fooled by proof terms that also exploit a separate nanoda bug, since fixed. Leonardo de Moura's postmortem of 1 August 2026 adds that lean4lean is affected by the kernel bug, since its handling of inductives is a port of the reference implementation.

- Using the Lean release notes, the release tags and GitHub's lists of commits, I checked that the fix is in Lean 4.32.2 (released 28 July 2026) and in v4.33.0, v4.34.0, v4.34.1 (the version used in this book) and v4.35.0; v4.32.0 and v4.32.1 do not contain it. That is a statement about the source history, not a test of the bug.

What this shows

- A kernel bug is not the same kind of failure as the ones in chapter 3. There, a reader or a second tool catches a problem the kernel was never asked about. Here the kernel was asked, and answered wrongly.
- leanchecker could not help, as its documentation says: it uses the same kernel. The oss-security post reports that leanchecker accepts the $0 = 1$ proof, and that a separately written kernel, nanoda, accepted a different

forged proof (the Collatz disproof), so a second implementation did not catch that one either. The sources do not say why, and this book does not guess.

- A second kernel still helps. Two programs written separately by different people are unlikely to make the same mistake, and when one of them is wrong, a disagreement is a signal worth chasing. This example shows that agreement is evidence, not a guarantee.

- Version matters. A result checked with a toolchain from before a fix has a weaker claim than the same result checked after. Keep the toolchain version with the result, and read the release notes.

What this book does about it

Two re-checks were run on the demonstrations in chapter 3.

- leanchecker, which uses the same kernel as the build, on the Lean 4.34.1 builds.
- nanoda, a kernel written separately in Rust, version 0.4.19. Lean's own release toolchain for version 4.35.0 contains it, together with leanexport, the program that writes a theorem and everything it depends on out as plain text. Those tools are not in the 4.34.1 toolchain, so the same source files were built with Lean 4.35.0, exported, and given to nanoda. It was set to permit only the three standard axioms. Chapter 3 has the results: it accepted the files whose proofs are valid, refused the two with extra axioms under its axiom rule, and did not accept the file with checking switched off (it stopped with an internal assertion failure).

Two limits. One: the nanoda results are for the Lean 4.35.0 builds of the same files, not for the 4.34.1 builds. The other: lean4lean, the other separately written kernel, is also in the 4.35.0 toolchain folder on this PC, but it could not read the compiled files ("incompatible header"), so no lean4lean result is claimed.

Both Lean versions used here contain the fix for the 2026 bug, so none of these runs says anything about that bug.

Yesterday, today, tomorrow

Yesterday. Mario Carneiro wrote lean4lean and ammkrn wrote nanoda partly so that the Lean kernel would not be the only judge of Lean proofs. In July 2026 a kernel bug was reported by kiranandcode and fixed by Leonardo de Moura within hours, and Jonathan Brossard wrote it up for the security community.

Today. Sonny notes the Lean version with each result and treats an agreement of two separately written kernels as stronger evidence than one, not as proof. In this book the demonstrations were re-checked with leanchecker and with nanoda, and the book says which Lean version each was run on.

Tomorrow. Getting lean4lean to run on these files, and running a separate kernel on every future Lean result that matters, is the next step. If a kernel bug is found later, a result checked with an older toolchain can be re-checked with the fixed one.

Check it yourself

The nanoda step for the seven demonstrations is the optional second-kernel recipe in the check guide. The 2026 bug is described from the public sources named in the credits; it was not reproduced here.

Where to go next

- How were the demonstrations that these checks were run on built, and what did each check say? See Five ways to get $2 + 2 = 5$ past a build.
- What does a successful check still leave unsaid? See What "Lean-verified" does not mean.

7 Statement

Statement

The question: for seven small Lean files that each look fine, which checks accept them and which catch what is wrong? The answer: the table below gives, for each file, what the Lean statement says and what each check reported; five files are caught by reading the axiom report, a warning or the text, one is caught only by replaying the stored proof through the kernel, and the honest file needs nothing.

Setup: Lean 4.34.1 (x86_64 Windows build, commit 5045d005) with core Lean only; no Mathlib. Each demonstration is one short file in the demos folder of this book. The second-kernel column uses the same files built with Lean 4.35.0 (commit eb6cafb7), because the tools for it (leanexport and nanoda_bin 0.4.19) ship in that toolchain's folder.

What each file says, and what each check reported

Demo	What the Lean statement says	Build	Axiom report	leanchecker, fresh	nanoda (Lean 4.35.0 build)	What catches it
Honest file	$2 + 2 = 4$ on the natural numbers, by reflexivity	passes	none	accepts	accepts	nothing needed
Added axiom	$2 + 2 = 5$ on the natural numbers, from an axiom declared in the file	passes	the added axiom	accepts	refuses: axiom not permitted	the axiom report
Unfinished proof	$2 + 2 = 5$ on the natural numbers, proof is the placeholder sorry	passes, with a warning	the placeholder axiom	accepts	refuses: axiom not permitted	the warning and the axiom report
One-number system	$2 + 2 = 5$ in the numbers below 1, so only 0	passes	one standard axiom	accepts	accepts	reading the statement
Different plus sign	$2 + 2 = 5$ where a local definition makes plus on natural numbers add one extra	passes	none	accepts	accepts	reading the definitions in scope
Faulty definition	1 is prime, where the definition of prime forgets the condition of being at least 2	passes	one standard axiom	accepts	accepts	reading the definition
Kernel check off	False, stored with kernel checking turned off, then $2 + 2 = 5$ from it	passes	none	rejects	does not accept: internal assertion failure	the re-check through the kernel

Reading the table: for the added axiom and the unfinished proof the extra assumption is visible in the axiom report. The one-number system, the different plus sign and the faulty definition are fully accepted by every automatic check, including a separately written kernel, and are caught only by a person reading what the statement and its definitions say. The kernel-check-off file is the only one a kernel rejects, and it does so because the declaration was stored without ever being checked.

The same switch can be set in a project's build file instead of a Lean file; then no Lean file mentions it.

The machine-readable version of the table is the demonstrations file in this folder.

Where to go next

- What does each of these checks do, and what is each blind to? See What "Lean-verified" does not mean.
- What do the real outputs look like? See Five ways to get $2 + 2 = 5$ past a build.

8 Theorems in this book

Theorems in this book

Every theorem and lemma in this book, with its exact statement, status and credit. Maths is written in LaTeX notation, between dollar signs.

The entries follow the book's own order (the order of the table in the statement page). The Lean names are internal labels, not terms of the problem. Each Lean statement is quoted word for word in a code block under its entry.

Demo 1: the honest file, $2 + 2 = 4$

Credit: Sonny (2026), built with Lean 4.34.1 (Leonardo de Moura, Sebastian Ullrich and the Lean community).

$2 + 2 = 4$ on the natural numbers, by reflexivity.

Status: Proved and checked in Lean (the Lean statement as written, accepted by the kernel; the meaning of the statement is not claimed). Build: passes. Axiom report: none. leanchecker, fresh: accepts. nanoda (Lean 4.35.0 build): accepts. What catches it: nothing needed.

```
theorem Demo.honest : 2 + 2 = 4 := rfl
```

Proof: chapters/03-two-plus-two-is-five.md (https://gocheckyourself.org/read/lean-verified/experts/#ch-chapters-03-two-plus-two-is-five), file demos/Demo/Honest.lean; Lean theorem named in the code block above.

Demo 2: the added axiom, $2 + 2 = 5$ **from a declared axiom**

Credit: Sonny (2026).

$2 + 2 = 5$ on the natural numbers, from an axiom declared in the file.

Status: Accepted by the build but dependent on an added or placeholder assumption, visible in the axiom report. Build: passes. Axiom report: the added axiom. leanchecker, fresh: accepts. nanoda (Lean 4.35.0 build): refuses, axiom not permitted. What catches it: the axiom report.

```
axiom Demo.bad : 2 + 2 = 5

theorem Demo.two_plus_two_is_five : 2 + 2 = 5 := Demo.bad
```

Proof: chapters/03-two-plus-two-is-five.md (https://gocheckyourself.org/read/lean-verified/experts/#ch-chapters-03-two-plus-two-is-five), file demos/Demo/AddedAxiom.lean; Lean theorem and axiom named in the code block above.

Demo 3: the unfinished proof, $2 + 2 = 5$ **by a placeholder**

Credit: Sonny (2026).

$2 + 2 = 5$ on the natural numbers, where the proof is the placeholder “sorry”.

Status: Accepted by the build but dependent on an added or placeholder assumption, visible in the axiom report. Build: passes, with a warning. Axiom report: the placeholder axiom. leanchecker, fresh: accepts. nanoda (Lean

4.35.0 build): refuses, axiom not permitted. What catches it: the warning and the axiom report.

```
theorem Demo.two_plus_two_is_five_sorry : 2 + 2 = 5 := by sorry
```

Proof: chapters/03-two-plus-two-is-five.md (https://gocheckyourself.org/read/lean-verified/experts/#ch-chapters-03-two-plus-two-is-five), file demos/Demo/UsesSorry.lean; Lean theorem named in the code block above.

Demo 4: the one-number system, $2 + 2 = 5$ **below** 1

Credit: Sonny (2026).

$2 + 2 = 5$ in the numbers below 1, so only 0. In Lean, the type of numbers below 1 has only the number 0.

Status: Proved and checked in Lean (the Lean statement as written, accepted by the kernel; the meaning of the statement is not claimed). Build: passes. Axiom report: one standard axiom. leanchecker, fresh: accepts. nanoda (Lean 4.35.0 build): accepts. What catches it: reading the statement.

```
theorem Demo.small_number_system : (2 + 2 : Fin 1) = 5 := by decide
```

Proof: chapters/03-two-plus-two-is-five.md (https://gocheckyourself.org/read/lean-verified/experts/#ch-chapters-03-two-plus-two-is-five), file demos/Demo/SmallerNumberSystem.lean; Lean theorem named in the code block above.

Demo 5: the different plus sign, $2 + 2 = 5$

Credit: Sonny (2026).

$2 + 2 = 5$ where a local definition makes plus on natural numbers add one extra.

Status: Proved and checked in Lean (the Lean statement as written, accepted by the kernel; the meaning of the statement is not claimed). Build: passes. Axiom report: none. leanchecker, fresh: accepts. nanoda (Lean 4.35.0 build): accepts. What catches it: reading the definitions in scope.

```
-- name: Demo.redefined_plus
local instance : Add Nat := ⟨fun a b => Nat.succ (Nat.add a b)⟩

theorem redefined_plus : 2 + 2 = 5 := rfl
```

The full name in the code block above shows the namespace the theorem sits in.

Proof: chapters/03-two-plus-two-is-five.md (https://gocheckyourself.org/read/lean-verified/experts/#ch-chapters-03-two-plus-two-is-five), file demos/Demo/RedefinedPlus.lean; Lean theorem named in the code block above.

Demo 6: the faulty definition, 1 **is prime**

Credit: Sonny (2026).

1 is prime, where the definition of prime forgets the condition of being at least 2. In symbols, MyPrime(n): $\Leftrightarrow \forall d,\ d \mid n \Rightarrow d = 1 \vee d = n$, and the statement is MyPrime(1).

Status: Proved and checked in Lean (the Lean statement as written, accepted by the kernel; the meaning of the statement is not claimed). Build: passes. Axiom report: one standard axiom. leanchecker, fresh: accepts. nanoda (Lean 4.35.0 build): accepts. What catches it: reading the definition.

```
def MyPrime (n : Nat) : Prop := ∀ d : Nat, d ∣ n → d = 1 ∨ d = n

theorem Demo.one_is_prime : MyPrime 1 := by
  intro d h
  exact Or.inl (Nat.dvd_one.mp h)
```

Proof: chapters/03-two-plus-two-is-five.md (https://gocheckyourself.org/read/lean-verified/experts/#ch-chapters-03-two-plus-two-is-five), file demos/Demo/WrongDefinition.lean; Lean theorem named in the code block above.

Demo 7: the kernel-check-off file, False stored unchecked

Credit: Tools: Lean 4; leanchecker (Kim Morrison and Sebastian Ullrich); the nanoda kernel checker. This result (Sonny): computed.

False, stored with kernel checking turned off, then $2 + 2 = 5$ from it.

Status: Stored with kernel checking turned off, rejected by the re-check. Build: passes. Axiom report: none. leanchecker, fresh: rejects. nanoda (Lean 4.35.0 build): does not accept, internal assertion failure. What catches it: the re-check through the kernel.

```
set_option debug.skipKernelTC true in
run_cmd liftCoreM <| addDecl (Declaration.thmDecl
  { name := `Demo.unchecked, levelParams := [], type := mkConst ``False, va

theorem Demo.wrong : 2 + 2 = 5 := Demo.unchecked.elim
```

Proof: chapters/03-two-plus-two-is-five.md (https://gocheckyourself.org/read/lean-verified/experts/#ch-chapters-03-two-plus-two-is-five), file demos/Demo/SkipKernelTC.lean; Lean declarations named in the code block above.

Open questions

A second-kernel run with lean4lean

Credit: Tools: lean4lean (Mario Carneiro).

Whether the seven files are accepted by a run of lean4lean, a second separately written kernel. The copy in the Lean 4.35.0 toolchain folder could not read the compiled files here ("incompatible header"), so no lean4lean result is claimed.

Status: Open.

Proof: none; see chapters/04-kernel-bugs-and-second-checkers.md (https://gocheckyourself.org/read/lean-verified/experts/#ch-chapters-04-kernel-bugs-and-second-checkers).

9 Check it yourself: what "Lean-verified" covers

Check it yourself: what "Lean-verified" covers

This guide is a written recipe for each of the seven demonstrations: the command to run, the answer you should see, and which standard public tool gives it. It needs Lean, its build tool and its replay tool, and nothing else. It uses no program of ours, just Lean and its own tools. Each recipe also comes as a step-by-step page with commands for Windows, macOS and Linux and the exact output to expect.

What is claimed

The question these claims answer: which checks accept each of seven small Lean files, and which catches what is wrong in each?

Seven small Lean files were built, re-checked and (for the Lean 4.35.0 builds) given to a separately written kernel. The results are:

Demo	Build	Axiom report	leanchecker and its fresh mode	nanoda (Lean 4.35.0 build)
Honest file	passes	none	accepts	accepts
Added axiom	passes	the added axiom	accepts	refuses: axiom not permitted
Unfinished proof	passes, with a warning	the placeholder axiom	accepts	refuses: axiom not permitted
One-number system	passes	one standard axiom	accepts	accepts
Different plus sign	passes	none	accepts	accepts
Faulty definition	passes	one standard axiom	accepts	accepts
Kernel check off	passes	none	rejects	does not accept (assertion failure, exit code 101)

The per-demo statements are in the statement file.

Status

Seven Lean demonstrations; not yet refereed. Each result was produced on one Windows PC and produced again, after edits to the comments of six files and to one declaration name; the output in this guide is from the second run. Lean 4.34.1 (commit 5045d005) was used for the builds and leanchecker; Lean 4.35.0 (commit eb6cafb7) for the nanoda step, because leanexport (lean4export 3.1.0) and nanoda_bin (0.4.19) are in that release toolchain and not in 4.34.1's. No reader outside this project has reviewed the files or the text.

Commands

Start with the fingerprints. The certificates folder holds one file, a SHA-256 list of the seven Lean files and the three build files. The three build files are in the download lean-verified-lean.zip (unpack it as the lean folder). The seven Lean files are in the demos folder; download them into a demos folder next to the lean folder. From the book folder, run the command below (on Windows,

CertUtil -hashfile with SHA256 does the same one file at a time); with all ten files, every line must report OK.

```
sha256sum -c certificates/SHA256SUMS.txt (macOS: shasum -a 256 -c certifica
```

You need elan (the Lean version manager). No Mathlib. The lean folder pins Lean 4.34.1. Run from the lean folder of this book. For each demo name in the table below, run:

```
lake build Demo.<Name>
lake env leanchecker Demo.<Name>
lake env leanchecker --fresh Demo.<Name>
```

The build command builds the file and prints Lean's messages, including the axiom report. The replay command replays every stored declaration through the kernel. The fresh command also replays everything the file imports (about 100 seconds and 1 GB of memory per demo). leanchecker prints nothing and exits with code 0 when it accepts; it prints a message and exits with code 1 when it rejects.

Optional second kernel (nanoda). Build the same files with Lean 4.35.0, export the named theorem, and give the export to nanoda with a configuration that permits only the three standard axioms:

```
lake env leanexport Demo.<Name> -- <theorem name> > export.ndjson
lake env nanoda_bin config.json
```

where config.json is:

```
{
  "export_file_path": "export.ndjson",
  "permitted_axioms": ["propext", "Classical.choice", "Quot.sound"],
  "unpermitted_axiom_hard_error": true,
  "nat_extension": true,
  "string_extension": true,
  "print_success_message": true,
  "print_axioms": false,
  "num_threads": 1,
  "pp_to_stdout": false,
  "pp_declars": []
}
```

Recipe for each demo

Each recipe gives the module name, the theorem to export for nanoda, and the expected output.

Honest file. The module, and the theorem to export for nanoda:

```
module:  Demo.Honest
theorem: Demo.honest
```

Expected build output:

```
ℹ Built Demo.Honest
info: ../demos/Demo/Honest.lean:4:0: 'Demo.honest' does not depend on any a
Build completed successfully (2 jobs).
```

leanchecker: accepts, both modes. nanoda: Checked 38 declarations with no errors.

Added axiom. The module, and the theorem to export for nanoda:

```
module:  Demo.AddedAxiom
theorem: Demo.two_plus_two_is_five
```

Expected build output:

```
ℹ Built Demo.AddedAxiom
info: ../demos/Demo/AddedAxiom.lean:7:0: 'Demo.two_plus_two_is_five' depend
Build completed successfully (2 jobs).
```

leanchecker: accepts, both modes. nanoda refuses the export:

```
Error: export file declares unpermitted axiom "Demo.bad"
```

Unfinished proof. The module, and the theorem to export for nanoda:

```
module:  Demo.UsesSorry
theorem: Demo.two_plus_two_is_five_sorry
```

Expected build output:

```
⚠ Built Demo.UsesSorry
warning: ../demos/Demo/UsesSorry.lean:3:8: declaration uses `sorry`
info: ../demos/Demo/UsesSorry.lean:5:0: 'Demo.two_plus_two_is_five_sorry' d
Build completed successfully (2 jobs).
```

leanchecker: accepts, both modes. nanoda refuses the export:

```
Error: export file declares unpermitted axiom "sorryAx"
```

One-number system. The module, and the theorem to export for nanoda:

```
module:  Demo.SmallerNumberSystem
theorem: Demo.small_number_system
```

Expected build output:

```
ℹ Built Demo.SmallerNumberSystem
info: ../demos/Demo/SmallerNumberSystem.lean:7:0: 'Demo.small_number_syster
info: ../demos/Demo/SmallerNumberSystem.lean:8:0: 5 : Fin 1
info: ../demos/Demo/SmallerNumberSystem.lean:9:0: 0
info: ../demos/Demo/SmallerNumberSystem.lean:10:0: 0
Build completed successfully (2 jobs).
```

leanchecker: accepts, both modes. nanoda: Checked 294 declarations with no errors.

Different plus sign. The module, and the theorem to export for nanoda:

```
module:  Demo.RedefinedPlus
theorem: Demo.redefined_plus
```

Expected build output:

```
ℹ Built Demo.RedefinedPlus
info: ../demos/Demo/RedefinedPlus.lean:12:0: 'Demo.redefined_plus' does not
info: ../demos/Demo/RedefinedPlus.lean:13:0: 5
info: ../demos/Demo/RedefinedPlus.lean:14:0: instAddNat_demo
info: ../demos/Demo/RedefinedPlus.lean:19:0: instAddNat
Build completed successfully (2 jobs).
```

leanchecker: accepts, both modes. nanoda: Checked 38 declarations with no errors.

Faulty definition. The module, and the theorem to export for nanoda:

```
module:  Demo.WrongDefinition
theorem: Demo.one_is_prime
```

Expected build output:

```
ℹ Built Demo.WrongDefinition
info: ../demos/Demo/WrongDefinition.lean:10:0: 'Demo.one_is_prime' depends
Build completed successfully (2 jobs).
```

leanchecker: accepts, both modes. nanoda: Checked 198 declarations with no errors.

Kernel check off. The module, and the theorem to export for nanoda:

```
module:  Demo.SkipKernelTC
theorem: Demo.wrong
```

Expected build output:

```
ℹ Built Demo.SkipKernelTC
info: ../demos/Demo/SkipKernelTC.lean:13:0: 'Demo.unchecked' does not deper
info: ../demos/Demo/SkipKernelTC.lean:18:0: 'Demo.wrong' does not depend or
Build completed successfully (2 jobs).
```

leanchecker (both modes) rejects, with exit code 1 (the fresh mode prints the same message without its opening line):

```
leanchecker found a problem in Demo.SkipKernelTC
uncaught exception: while replaying declaration 'Demo.unchecked':
(kernel) declaration type mismatch, 'Demo.unchecked' has type
  True
but it is expected to have type
  False
```

nanoda does not accept; it stops with exit code 101:

```
thread 'main' panicked at src\tc.rs:955:71:
assertion failed: self.def_eq(u, v)
```

What the answers show

Only the last demo is rejected by an automatic re-check. The added axiom and the unfinished proof are caught by the axiom report (and by nanoda's axiom rule). The one-number system, the different plus sign and the faulty definition pass every automatic check, because their proofs are valid; they are caught only by reading the statement and the definitions.

lean4lean was also tried. The copy in the Lean 4.35.0 toolchain folder could not read the compiled files ("incompatible header"), so no lean4lean result is claimed.

How I direct this work, and my own tools, are not part of this book. Everything you need to check the results is public: the results, their certificates and fingerprints, the Lean proofs, and a written recipe for each check. In this book the results are the seven Lean files and what each check printed; there are no certificate files beyond the fingerprint list of those files. The seven Lean files are in the demos folder.

Supporting records are available on request.

Expected output

The expected output of every command is given in the recipe for each demo above. In short: six of the seven files pass every leanchecker run, the seventh (kernel check off) is rejected by both runs with a type mismatch between True and False, nanoda accepts the four files whose proofs are valid, refuses the two with extra axioms, and does not accept the seventh.

Time and memory

Without the fresh mode, a few minutes for all seven demos. With the fresh mode, about 100 seconds and up to about 1 GB of memory per demo. The nanoda step takes a few seconds per demo. Setting the environment variable LEAN_NUM_THREADS to 2 keeps memory use modest.

What is not claimed

- That Lean, leanchecker or nanoda is unreliable. Each behaved as designed; the demonstrations show the limits of what each check covers.
- That these seven files cover every way a Lean result can mislead.
- A result about the 2026 kernel bug. It was not reproduced, and both Lean versions used contain its fix (chapter 4).
- A result from lean4lean (not usable here), from Lean's comparator tool, or from any kernel other than Lean's own and nanoda.
- That nanoda checked the Lean 4.34.1 builds. It checked the same source files built with Lean 4.35.0, because the export tool is in that toolchain.
- That nanoda's behaviour on the kernel-check-off file is a diagnosis. It stopped with an internal assertion failure. The book gives no cause for it; it notes only that the failure is consistent with the type mismatch that leanchecker reports.

Where to go next

- Why do three of the seven files pass every automatic check? See What "Lean-verified" does not mean.
- What happens if the kernel itself has a bug? See Kernel bugs, and why a second checker helps.

We hope it helps others find more results and keeps their curiosity going.
We'd love to collaborate with anyone.

10 Files and fingerprints

This chapter is the book's file SHA256SUMS.txt.

11 Trust and verification

This chapter is the book's file VERIFY.md.

12 Lean sources: how to check them

This chapter is the book's file THEOREMS.md.

13 People and works

This chapter is the book's file CREDITS.md.

14 Files and rights

Licence

The files Sonny wrote for this book are under the Creative Commons Attribution 4.0 International licence (CC BY 4.0). You may copy, share and adapt them for any purpose, as long as you give credit.

How to give credit: "Sonny (Sonny), gocheckyourself.org", with a link to the page or file you used, and a note saying what you changed, if anything.

What it covers: the chapters, the statement pages, the check guide, the credits, the seven short Lean demonstration files and the project files that build them. The Lean files contain no code copied from Lean, Mathlib or the checkers; they use Lean's public language and library.

What it does not cover:

- Lean and the tools named in CREDITS.md (Lean, nanoda, lean4lean, lean4export and leanchecker are Apache 2.0). None of their code ships in this book.
- Programs that are not part of this book, and work that is not published on this site.
- Trade marks, and the other rights that section 2(b) of the licence leaves out.

The licence file is LICENSE-CC-BY-4.0.txt; the full legal code is at https://creativecommons.org/licenses/by/4.0/legalcode.

Where to go next

- Which tools and texts does this book rely on, and under what licences? See Credits.
- How do you rerun the seven demonstrations yourself? See Check it yourself.

15 Each result on its own page

1. Demo 1: the honest file, $2 + 2 = 4$ STATUS: LEAN-CHECKED (SECOND-KERNEL)
2. Demo 2: the added axiom, $2 + 2 = 5$ from a declared axiom STATUS: COMPUTED (CALIBRATION-CONTROL)
3. Demo 3: the unfinished proof, $2 + 2 = 5$ by a placeholder STATUS: COMPUTED (CALIBRATION-CONTROL)
4. Demo 4: the one-number system, $2 + 2 = 5$ below 1 STATUS: LEAN-CHECKED (SECOND-KERNEL)
5. Demo 5: the different plus sign, $2 + 2 = 5$ STATUS: LEAN-CHECKED (SECOND-KERNEL)
6. Demo 6: the faulty definition, 1 is prime STATUS: LEAN-CHECKED (SECOND-KERNEL)
7. Demo 7: the kernel-check-off file, False stored unchecked STATUS: COMPUTED (CALIBRATION-CONTROL)
8. Open question: A second-kernel run with lean4lean STATUS: OPEN

© 2026 Sonny. The books are under CC BY 4.0 (Creative Commons Attribution 4.0 International): reuse them with credit and a link. Each book's licence file lists the files that keep their own licences.