PROOF101: Formal Verification & Proof Assistants

Google Developer Groups @ AUB
& AUB Math Society
Spring 2026

“Testing shows the presence, not the absence of bugs.”

— Edsger W. Dijkstra

Week 1 of 10

Why Our Code Breaks (and How to Fix It)

Daniel Dia & Guest Lecturers

https://danieldia-dev.github.io/proofs/

Section 1

Introduction

Welcome to PROOF101

Course Overview: An introduction to formal verification and proof assistants using Lean4

What you'll learn:

- Why software failures are so costly
- The fundamental limitations of testing
- What formal verification can do that testing cannot
- How to use Lean4 to write verified software
- The mathematics of program correctness

By the end of this course:

- Write proofs in Lean4
- Understand dependent type theory
- Verify properties of your programs
- Appreciate the power of formal methods

Section 2

The High Cost of Software Bugs

The Cost of Getting It Wrong

Software bugs aren't just annoying – they're catastrophic

We'll examine three disasters that could have been prevented:

- **Ariane 5 (1996)**: $370 million rocket explosion in 37 seconds
- **Therac-25 (1985-87)**: Radiation overdoses kill 3 patients
- **Heartbleed (2014)**: 17% of the internet's encryption compromised

Common threads:

- All were created by expert teams
- All had extensive testing
- All involved seemingly "simple" bugs
- All could have been caught by formal verification

Section 2

The High Cost of Software Bugs

Subsection 2.1

Ariane 5 Rocket (1996)

Ariane 5: $370 Million in 37 Seconds

June 4, 1996: European Space Agency's Ariane 5 rocket

The Disaster:

- 37 seconds after launch, rocket self-destructed
- $370 million lost (rocket + payload)
- Years of work destroyed
- No casualties (unmanned), but a massive setback

The Cause: Integer overflow

- Reused code from Ariane 4 (cost-saving measure)
- Converted 64-bit floating-point velocity to 16-bit signed integer
- Ariane 5 was faster than Ariane 4
- Velocity exceeded 16-bit integer range ($> 32,767$)
- Overflow caused system crash

Ariane 5: The Bug

Why Testing Missed It:

- Tested with Ariane 4 flight profiles (slower velocities)
- Never tested with Ariane 5's higher velocities
- Assumed reused code was "proven" by prior flights
- Testing can only check cases you think to test

The Problematic Code:

```
// 64-bit float velocity
double vel_x = get_velocity();

// Convert to 16-bit integer
// (Range: -32768 to 32767)
int16_t vel_int = (int16_t)vel_x;

// If vel_x > 32767: OVERFLOW
```

How formal verification could have helped:

- Type system enforces bounds
- Static analysis detects overflow
- Dependent types: `Int16Range`

Ariane 5: The Danger of Assumptions

The reuse fallacy: "It worked before, so it's safe"

What went wrong:

- Code was correct for Ariane 4's specifications
- Ariane 5 had different specifications (higher velocity)
- No verification that old code met new specifications
- The type system allowed an unsafe conversion

The lesson:

- Testing doesn't prove general correctness
- It proves correctness for tested scenarios only
- When requirements change, tests must change too
- But with formal verification, the types would have caught this immediately

Past success is not a guarantee of future correctness

Section 2

The High Cost of Software Bugs

Subsection 2.2

Therac-25 Medical Device (1985-1987)

Therac-25: When Software Kills

1985-1987: Radiation therapy machine

The Disaster:

- 6 patients received massive radiation overdoses (100x intended)
- 3 deaths directly attributed to device
- 3 more suffered serious injuries
- One of the worst medical device failures in history

The Cause: Race condition in safety-critical code

- Machine had two modes: low-power X-ray, high-power electron beam
- Operator could edit settings rapidly
- Software had timing-dependent bug
- Fast operator inputs triggered race condition
- Safety checks bypassed, high-power beam fired without shielding

Therac-25: The Race Condition Problem

The Race Condition:

- Bug only occurred if operator edited within 8 seconds
- Required specific sequence of keystrokes
- Happened once in thousands of uses
- Intermittent failure – hardest to debug

Testing Limitations:

- Tested with "normal" operator speeds
- Didn't test edge cases (fast editing)
- Timing-dependent bugs rarely caught in testing
- Manual testing can't explore all timing interleavings

The fundamental challenge:

- Concurrent systems have exponentially many execution orders
- Even two operations can interleave in many ways
- Testing explores only a tiny fraction

The bug was literally waiting for the right timing to kill someone

Concurrent Systems Are Hard

The state space explosion:

- With n operations, there are $n!$ possible orderings
- 10 operations = 3.6 million possible orderings
- 20 operations = 2.4×10^{18} orderings
- Testing can check maybe thousands of orderings

Real systems are worse:

- Operations can be interrupted mid-execution
- Multiple threads running on multiple cores
- Non-deterministic timing
- Hardware reordering of operations

Testing approach: Try some orderings, hope for the best
Formal verification: Prove properties hold for ALL orderings

Section 2

The High Cost of Software Bugs

Subsection 2.3

Heartbleed (2014)

Heartbleed: The Internet's Worst Day

April 2014: Critical vulnerability in OpenSSL

The Impact:

- Affected 17% of all secure web servers (500,000+)
- Exposed passwords, private keys, personal data
- Major sites affected: Google, Facebook, Yahoo, Amazon
- Existed for 2+ years before discovery
- Billions of dollars in damages

The Cause: Buffer over-read (bounds check missing)

- NOT a flaw in cryptography itself, but a Simple programming error
- Client sends: "My payload is N bytes" + actual payload
- Server blindly trusts N without checking actual size
- **Client lies**: claims 64KB payload, sends 1 byte
- Server returns 64KB – actual payload + 64KB of memory!

Heartbleed: The Bug

Simplified version of the vulnerable code:

```c
uint16_t payload_length = read_uint16(request);  // Client claims length
char* payload = read_bytes(request);

// VULNERABILITY: No check if payload_length matches reality!
char* response = malloc(payload_length);
memcpy(response, payload, payload_length);  // BUFFER OVER-READ
send_response(response, payload_length);
```

The Attack:

- Attacker sends: `payload_length = 64000`, actual payload: 1 byte
- `memcpy` copies 64000 bytes starting from payload
- Reads past end of payload into adjacent memory
- Returns 64KB of secret server memory to attacker
- Attacker retrieves passwords, encryption keys, personal data

Heartbleed: Why Testing Failed

Why Wasn't This Caught?

- OpenSSL is open source – 1000s of eyes on code
- Extensive test suite
- Used by major companies
- But: tests assumed honest clients
- Never tested malicious inputs

Testing Limitations:

- Can't test all possible inputs
- Adversarial testing requires security mindset
- Memory errors don't always crash immediately
- May pass all tests but still be exploitable

The deeper issue:

- Testers think like developers
- Attackers think differently
- Can you test for all malicious inputs?

You can't test what you don't think to test

Section 3

The Limitations of Testing

The Fundamental Problem: Finite Tests, Infinite Possibilities

The core limitation of testing:

- Programs have infinite possible inputs
- We can only run finite number of tests
- Bugs hide in the untested cases

Example: Simple addition function `add(x: Int64, y: Int64)`

- Possible inputs: $2^{64} \times 2^{64} = 2^{128}$ combinations
- That's roughly 3.4×10^{38} test cases
- At 1 billion tests/second: 10^{21} years to test all
- The universe is only 1.4×10^{10} years old!

Dijkstra's quote (1972):
"Testing shows the presence, not the absence of bugs"

Why Testing Isn't Enough

Testing can show the presence of bugs, never their absence. — Edsger Dijkstra

Fundamental Limitations:

- Can only check finite number of inputs
- Can't test all possible executions (timing, concurrency)
- Can't test adversarial behavior
- Doesn't prove correctness, only finds bugs

Example: Testing `add(x, y)`

- Infinite inputs: $2^{64} \times 2^{64}$ combinations (64-bit ints)
- Would take billions of years to test all inputs
- Test 1000 cases? Still leaves 10^{30+} untested
- Testing gives confidence, not certainty

What Testing Systematically Misses

Categories of bugs testing fails to catch:

1. **Edge cases you didn't think of**
   - Extreme values, boundary conditions, unusual combinations
   - Example: Ariane 5 (higher velocity than tested)
2. **Rare timing-dependent bugs**
   - Race conditions, deadlocks, timing windows
   - Example: Therac-25 (8-second window)
3. **Malicious inputs**
   - Adversarial testing requires attacker mindset
   - Example: Heartbleed (lying about payload size)
4. **Complex interactions**
   - Emergent behavior from component combinations
   - Exponential state space

The Testing Paradox

What testing IS good for:

- Finding bugs during development
- Regression testing (so fixes don't break things)
- Integration testing (components work together)
- Performance testing
- User acceptance testing

What testing CANNOT guarantee:

- Absence of bugs
- Correctness for all inputs
- Freedom from race conditions
- Memory safety

Testing is necessary but insufficient

For safety-critical systems:
Testing alone is not acceptable

- Medical devices, aerospace, autonomous vehicles
- Financial systems, cryptography, infrastructure
- Need mathematical proof of correctness

Section 4

Formal Methods: A Better Way

What is Formal Verification?

Using mathematical logic to prove program correctness

Core Idea:

- Specify desired behavior mathematically
- Prove program meets specification
- Computer checks the proof
- *If proof is valid, program is correct – for ALL inputs*

Not just for math theorems:

- Hardware correctness (CPU designs)
- Software correctness (compilers, OS)
- Network protocols (TLS, consensus)
- Cryptographic implementations
- Control systems (aircraft, medical)

The line blurs:

- Systems require math to describe
- Theorems may require computation
- Programs and proofs are deeply connected (Curry-Howard)

What Does "Specification" Mean?

A specification is a precise mathematical description of what a program should do

Examples:

- `sort(list)` returns a permutation of the input where each element $\leq$ the next
- `encrypt(key, plaintext)` produces ciphertext that can only be decrypted with the same key
- `transfer(from, to, amount)` decreases `from` by `amount` and increases `to` by `amount`

Specifications can be:

- Functional properties (what output for what input)
- Safety properties (bad things never happen)
- Liveness properties (good things eventually happen)
- Security properties (attackers can't learn secrets)

Verification proves: implementation matches specification

The Gold Standard: Proof

What is a mathematical proof?

- Chain of logical reasoning from axioms to conclusion
- Each step justified by previously proven facts
- No gaps, no hand-waving
- Checkable by an independent verifier

20th century logic breakthrough:

- All mathematical reasoning reducible to small set of rules
- Foundation: axiomatic set theory or type theory
- Proofs are mechanical – computers can check them
- Enables automated reasoning

Two complementary approaches:

1. **Automated Theorem Proving**: Computer finds proofs
2. **Interactive Theorem Proving**: Human guides, computer verifies

A Brief History of Formal Methods

The journey from mathematics to verified software:

- **1930s**: Church, Turing, Gödel – foundations of computation
- **1960s**: Dijkstra, Hoare – program correctness logic
- **1968**: De Bruijn – Automath, first proof assistant
- **1970s-80s**: Development of type theory (Martin-Löf, Coquand)
- **1984**: Rocq proof assistant created
- **2000s**: Verified compilers (CompCert), OS kernels (seL4)
- **2010s**: Lean development begins at Microsoft Research
- **2020s**: Industry adoption, Lean 4 release, Lean FRO formed

From theoretical curiosity to practical tool for building trustworthy systems

Approach 1: Automated Theorem Proving

Goal: Find proofs automatically

Technologies:

- **SAT solvers**: Boolean satisfiability (billions of variables)
- **SMT solvers**: Satisfiability modulo theories (arithmetic, arrays, etc.)
- **Resolution provers**: First-order logic
- **Computer algebra systems**: Symbolic mathematics
- **Model checkers**: Explore all possible states

Strengths:

- Fully automated – no human intervention
- Very fast for specific domains
- Industrial applications (hardware verification)

Trade-offs:

- Power & efficiency at expense of *guaranteed soundness*
- May have bugs in implementation
- Limited to specific domains
- Can fail to find proofs that exist

SAT/SMT Solvers in the Real World

Where automated theorem proving shines:

Hardware verification:

- Intel, AMD use SAT solvers to verify CPU designs
- Find bugs in circuits with billions of transistors
- Prevents Pentium FDIV-style disasters

Software analysis:

- Microsoft's Static Driver Verifier uses SMT solvers
- Finds bugs in Windows device drivers
- Amazon's S2N uses SMT to verify crypto implementations

The trade-off:

- Automated tools themselves might have bugs
- No mathematical guarantee of soundness
- But: extremely practical for finding bugs quickly

Approach 2: Interactive Theorem Proving

Goal: Verify every step is correct

How it works:

- Human provides high-level proof strategy
- Computer fills in details and checks correctness
- Every step justified (axioms & prior thms)
- Produces **proof objects** – independently checkable

Used for:

- Safety-critical systems (e.g. verified compilers)
- Mathematical theorems

Strengths:

- **Guaranteed soundness** – small trusted kernel
- Works for *any* domain
- Proofs are artifacts – can be shared/checked
- Extremely high assurance

Trade-offs:

- Requires human expertise and time
- Steep learning curve

Interactive Theorem Proving: Success Stories

Real systems verified with proof assistants:

CompCert (using Rocq):

- Verified C compiler
- Proves: compiled code behaves exactly as source code specifies
- No compiler bugs can silently introduce errors

seL4 (using Isabelle/HOL):

- Verified OS microkernel
- Proves: kernel has no buffer overflows, null pointer dereferences, etc.
- Used in critical systems (autonomous vehicles, medical devices)

Mathematical achievements:

- Four Color Theorem (Rocq)
- Feit-Thompson Theorem (Rocq)
- Kepler Conjecture (Isabelle/HOL and Lean)

Automated vs Interactive: When to Use Each

Automated (SAT/SMT):

- **Use when:** Finding bugs quickly
- **Guarantee:** Might miss bugs, might have false positives
- **Effort:** Low (push-button)
- **Domain:** Specific (bounded model checking)

Interactive (Proof Assistants):

- **Use when:** Need absolute certainty
- **Guarantee:** Mathematical soundness
- **Effort:** High (requires expertise)
- **Domain:** General (any property you can state)

The spectrum:

- **Left** (Automated): Fast, practical, but less certain
- **Right** (Interactive): Slow, demanding, but provably correct
- **Lean's approach**: Combine both! Automation within proof assistant framework

Section 5

The Tool: Proof Assistants

What is a Proof Assistant?

Interactive software for constructing formal proofs:

- User provides guidance, system ensures correctness
- Type system = logical system
- Programs = Proofs (Curry-Howard correspondence)
- Automated tactics for common patterns

Supports both:

- Mathematical reasoning (theorems, lemmas, propositions)
- Systems reasoning (software/hardware correctness)

Major proof assistants:

- **Rocq** (France) – used for CompCert verified C compiler
- **Isabelle/HOL** (Cambridge, TU Munich) – seL4 verified OS kernel
- **Agda** (Sweden) – Homotopy type theory
- **Lean** (Microsoft Research → Lean FRO) – mathematical proofs + systems

The Curry-Howard Correspondence

An insane discovery: Programs and proofs are (secretly) the same thing!

In Logic:

- Propositions
- Proofs
- Implication ($A \to B$)
- Conjunction ($A \wedge B$)
- Truth

In Programming:

- Types
- Programs
- Function types
- Product types (tuples)
- Unit type

The correspondence:

- Writing a program of type T = proving proposition T
- Type-checking = proof-checking
- Running a program = simplifying a proof

This is why we can use programming languages as proof assistants!

The Lean Theorem Prover

Bridging automated and interactive approaches

Lean's Philosophy:

- Situate automated tools in framework supporting user interaction
- Construct fully specified axiomatic proofs
- Support reasoning about math AND complex systems
- Based on dependent type theory (Calculus of Constructions)

Unique Features:

- Powerful automation (tactics, decision procedures)
- Small trusted kernel (De Bruijn criterion)
- Metaprogramming in Lean itself

Real-world Impact:

- **Lean FRO**: Non-profit development
- Used in industry for verified systems
- **Mathlib**: Massive math library
- Active formal verification research

Section 6

Why Lean4?

Lean's Dual Nature

As a Proof Assistant:

- Interactive theorem proving
- Dependent type theory
- Verified mathematical proofs
- Guarantees correctness
- Curry-Howard correspondence

As a Programming Language:

- Functional programming
- Programs with precise semantics
- Reason about computations
- Extract verified code
- Lean implemented in itself!

"Programs are proofs, proofs are programs"

— (Oversimplified) Curry-Howard Correspondence

Why Lean4 for This Course?

1. **Modern & Active**
   - Developed at Lean FRO, rapidly evolving
   - Regular releases, active development, growing ecosystem
2. **Best of Both Worlds**
   - Powerful automation (tactics)
   - Verification guarantees (kernel)
   - SMT solver integration
3. **Practical Programming Language**
   - Real language, not just toy $\longrightarrow$ can write actual applications!
   - Good performance
4. **Extensible**
   - Metaprogramming in Lean itself
   - Create custom tactics, domain-specific automation
5. **Strong Community**
   - Excellent documentation with active forums and Discord
   - Industry adoption

Mathlib: Lean's Mathematical Library

One of Lean's killer features: Mathlib

What is Mathlib?

- Massive library of formalized mathematics
- Over 1 million lines of code
- Thousands of definitions and theorems
- From basic algebra to more advanced topics in math

Why it matters:

- Don't start from scratch – build on existing work
- Community-maintained, peer-reviewed
- Used in both research mathematics and verified systems
- We'll explore contributing to Mathlib in Week 6

Standing on the shoulders of giants

Dependent Types: The Key Innovation

What are Dependent Types?

- Types that can **depend on values**
- Types become first-class citizens
- Express properties impossible in traditional type systems

Power:

- Catch errors at compile-time that testing would miss
- Encode invariants in types
- Eliminate entire classes of bugs
- *"If it compiles, it's correct"*

Examples:

- `Vector` α `n` – list of exactly n elements
- `Matrix m n` – $m \times n$ matrix
- `Fin n` – numbers less than n
- `{x : Nat // x < 10}` – numbers < 10

We'll explore this deeply in Week 2!

Dependent Types: Making Invalid States Unrepresentable

Traditional types:

- `List Int` – any list of integers (could be empty, length 5, length 1000)
- `Int` – any integer (could be -1000, 0, 1000000)

Dependent types encode constraints:

- `Vector Int 5` – list of exactly 5 integers (can't be empty or length 4)
- `Fin 10` – integer from 0 to 9 (can't be 10 or -1)
- `{x : Int // x > 0}` – positive integers (can't be 0 or negative)

The power:

- Array access with `Fin n` index – no bounds checks needed!
- Division by `{x : Int // x ≠ 0}` – no division by zero!
- Matrix multiplication with compatible dimensions – no dimension mismatch!

The type system prevents you from writing buggy code

Lean's Approach: Strict, Pure, Functional

Three key properties:

1. Strict

- Arguments fully computed before function execution
- Like most languages (C, Java, Python)
- Predictable evaluation order

2. Pure

- No hidden side effects
- Same input always gives same output
- Enables equational reasoning

3. Functional

- Functions are first-class values
- Evaluation like mathematical expressions
- Immutable data structures
- Higher-order functions

Result: Programs easier to reason about and verify

Why Purity and Immutability Matter for Verification

Pure functions are easier to reason about:

Impure (with side effects):

- `f(x)` might return different values each time
- Might modify global state, file system, network
- Difficult to predict behavior
- Hard to verify mathematically

Pure (no side effects):

- `f(x)` always returns same value for same input
- No hidden modifications
- Can replace `f(x)` with its result (referential transparency)
- Easy to reason about mathematically: if $x = y$, then $f(x) = f(y)$

Purity enables equational reasoning – the foundation of formal verification

Section 7

Snapshot from Week 2: The Lean Architecture

From Source Code to Kernel

Lean's architecture:
Trusted kernel with untrusted elaborator

1. Source code (what you write):

- High-level, readable syntax
- Type inference, implicit arguments
- Tactics, notation, macros

2. Elaborator (untrusted):

- Fills in implicit arguments
- Resolves type class instances
- Expands macros and notation
- Compiles tactics to proof terms

3. Kernel (trusted):

- Small, verified core ($\sim$10k lines)
- Type checks all terms
- Ensures logical soundness
- De Bruijn indices, no names

The De Bruijn Criterion

De Bruijn criterion: Trust only a small, verified kernel

Named after: Nicolaas Govert de Bruijn (1918-2012)

- Dutch mathematician
- Created Automath (first proof assistant)
- Emphasized minimal trusted base

The principle:

- Keep the trusted core as small as possible
- All proof terms flow through the kernel
- Bugs in elaborator don't compromise soundness
- Kernel is small enough to verify by hand

Why This Separation Matters

Benefits:

- Elaborator can be complex without risking soundness
- Bugs in tactics don't compromise proofs
- Easy to add new features (tactics, notation)
- Kernel is small enough to verify by hand

Example: The `simp` tactic

- Elaborator: Complex rewrite engine (thousands of lines)
- Output: Simple chain of rewrite proof terms
- Kernel: Verifies each rewrite is valid
- If `simp` has a bug, kernel rejects the proof!

Words to live by: "Don't trust, verify" – Even if elaborator is buggy, kernel catches it!

What Do We Actually Trust?

The trust stack in Lean:

1. **The kernel** ($\sim$10,000 lines of code)
   - Small enough to audit by hand
   - Formally specified
   - Multiple independent implementations exist
2. **The type theory** (Calculus of Inductive Constructions)
   - Well-studied mathematical foundations
   - Known to be consistent (relative to weaker systems)
3. **The compiler/interpreter**
   - For execution (not for proofs)
   - Can be buggy – doesn't affect soundness of proofs

What we DON'T need to trust:

- The elaborator, tactics, or any automation
- External libraries (Mathlib)
- Our own proof code

Section 8

What Could Have Been Prevented?

From Runtime Disasters to Compile-Time Errors

Without FV:

- Write some code
- Test it (hoping you test the right cases)
- Deploy it
- Bug triggers at runtime
- System crashes / people may die / a lot of money is lost

With FV:

- Write code with formal specifications
- Attempt to compile
- Type system / proof checker rejects invalid code
- Bug caught at compile-time
- Fix it before it ever runs

The fundamental shift formal verification enables:
Turning catastrophic runtime crashes into compile-time type errors

Formal Verification: The Preventions

What could FV have prevented?

Ariane 5 ($370M loss):

- `BoundedInt 0 32767`
- Type system catches overflow
- Assignment must type-check
- Error at **compile-time**

Therac-25 (6 deaths):

- Verify all interleavings
- Prove: "beam $\implies$ shield"
- LTL: $\Box(\text{beam} \rightarrow \text{shield})$
- Race condition: **Proof fails**

Heartbleed (billions lost):

- `Array` α `n` – length in type
- Type error on bounds violation
- `n` $\leq$ `src.length` enforced
- Violation: **Type error**

The Bottom Line:
Formal verification turns
catastrophic crashes
into **compiler errors**.

Ariane 5: How Types Would Have Prevented the Disaster

The bug in traditional C:

- `double` (64-bit float) cast to `int16_t` (16-bit signed int)
- Type system allows this dangerous conversion
- Overflow is silent – no error, just wrong value
- Wrong value causes system crash

With dependent types:

- Velocity type: `Velocity : Float`
- Target type: `BoundedInt16 : {x : Int // -32768` $\leq$ `x` $\leq$ `32767}`
- Conversion requires proof: `-32768` $\leq$ `velocity` $\leq$ `32767`
- Proof fails for Ariane 5's velocity
- **Compilation fails** – developer forced to fix before launch

Type error instead of $370M explosion

Therac-25: How Model Checking Would Have Prevented Deaths

How formal verification could have helped:

- Model checking can explore all possible interleavings
- Linear temporal logic to specify safety properties
- *"Safety interlock must ALWAYS engage before beam fire"*

The specification in temporal logic:

- $\Box(\text{beam\_active} \rightarrow \text{shield\_engaged})$
- "Always" ($\Box$) means in every possible execution
- "Implies" ($\rightarrow$) means beam cannot fire without shield
- Model checker would explore all possible timing interleavings
- Race condition would cause the proof to FAIL

Result:

- Bug found during development, not after deployment
- Fix the race condition before any patient is harmed

Heartbleed: How Dependent Types Would Have Prevented Exposure

How formal verification could have helped:

- Dependent types: length encoded in type
- `Array` α `n` – array of exactly n elements
- Type system prevents out-of-bounds access at compile-time
- *Don't rely on programmers remembering to check bounds – make it impossible to forget*

With dependent types, the bug is impossible to express:

- The actual payload size is part of its type
- Compiler enforces: can't copy more than actual size
- Attempting to violate bounds = compile-time type error
- No runtime check needed – guaranteed safe by construction

The guarantee:

- If code compiles, bounds are correct
- No way to express buffer over-read in well-typed code

The Limits of Verification

The Core Guarantee

If it compiles in a verified system, certain classes of bugs *cannot exist*

What can be eliminated:

- Null pointer dereferences
- Buffer overflows
- Integer overflows
- Type mismatches
- Race conditions (with proper modeling)
- Logic errors (if specification is correct)

What still requires work:

- Specification correctness
- Performance bugs
- Hardware failures
- User errors

"The weakest link in the security chain is the human element."

— Kevin Mitnick

The Specification Challenge

Verification is only as good as the specification

The problem:

- Formal verification proves: implementation matches specification
- But what if the specification itself is wrong?
- Ex: specifying “system should encrypt data” but forgetting to specify “key must be secret”

Real-world example:

- Boeing 737 MAX crashes (2018-2019)
- Software worked as specified, but **specification** didn’t account for sensor failures
- 346 people died

The lesson:

- Verification requires correct specifications (that are still written by humans)
- This is why domain expertise + formal methods = crucial

Section 9

Course Structure

What We'll Cover in PROOF101

Week 1: Why Our Code Breaks (and How to Fix It) (today!)

Week 2: Dependent Type Theory (Daniel Dia)

Week 3: Functional Programming (Daniel Dia)

Week 4: Intro to Tactic-based Proving (Guest Session, Dr. Nadim Kobeissi)

Week 5: Proofs & Semantics (Daniel Dia)

Week 6: Contributing to Mathlib (Guest Session, Rida Hamadani)

Week 7: Proving Security in Software (Guest Session, Dr. Nadim Kobeissi)

Week 8: Monads, Tactics, and Applications (Guest Session, Dr. Robert Lewis)

Week 9: The Broader Landscape (Rust) and Project Kick-off (Daniel Dia)

TBD: The Mathematical Foundations of Proof Assistants (Guest Session, Dr. Assaf Kfoury)

Week 10: Project Showcase & Wrap-up (Entire team)

How to Succeed in This Course

Time commitment (different every week):

- Attend weekly lectures (1.5–2 hours)
- Complete weekly programming challenges (2–3 hours)
- Read assigned textbook chapters

Tips:

- Start assignments early – Lean has a bit of a learning curve
- Practice regularly – programming requires repetition
- Don't be discouraged – everyone struggles at first

Resources & Textbooks:

- **Theorem Proving in Lean 4**
- **Functional Programming in Lean**
- **The Hitchhiker's Guide to Logical Verification**
- Course website: `https://danieldia-dev.github.io/proofs/`
- Other links are on the course website (Discord & WhatsApp)

Section 10

Summary

Week 1 Summary

The Problem:

- Software bugs are catastrophically expensive
- Testing alone cannot guarantee correctness
- Safety-critical systems need stronger guarantees

The Solution:

- Formal verification uses mathematical proofs
- Proof assistants make verification practical
- Dependent types catch bugs at compile-time
- Interactive theorem proving provides highest assurance

Why Lean4:

- Combines automation with verification guarantees
- Modern, practical programming language
- Strong type system with dependent types
- Active community and ecosystem

Section 11

Assignments & Next Steps

This Week's Assignments

Readings (see the course website)

- Theorem Proving in Lean 4 – Chapter 1
- Functional Programming in Lean – Chapter 1
- Course website introduction and Lean setup guide

"Hand-in" Assignments (see the course website)

- Install the Lean4 VSCode extension
- Say "Hi" in Lean Zulip community (Lebanon or/and New Members channels)
- Complete PROOF101 Quiz 1

Questions & Discussion

Questions?

Join our community:

Discord: Link on website

WhatsApp: Link on website

Website: https://danieldia-dev.github.io/proofs/

Email: dmd13@mail.aub.edu

PROOF101: Formal Verification & Proof Assistants

Google Developer Groups @ AUB
& AUB Math Society
Spring 2026

"Testing shows the presence, not the absence of bugs."

— Edsger W. Dijkstra

Google Developer Groups
On Campus • American University of Beirut

MATHEMATICS SOCIETY
AMERICAN UNIVERSITY OF BEIRUT

Week 1 of 10

Why Our Code Breaks (and How to Fix It)

Daniel Dia & Guest Lecturers

`https://danieldia-dev.github.io/proofs/`