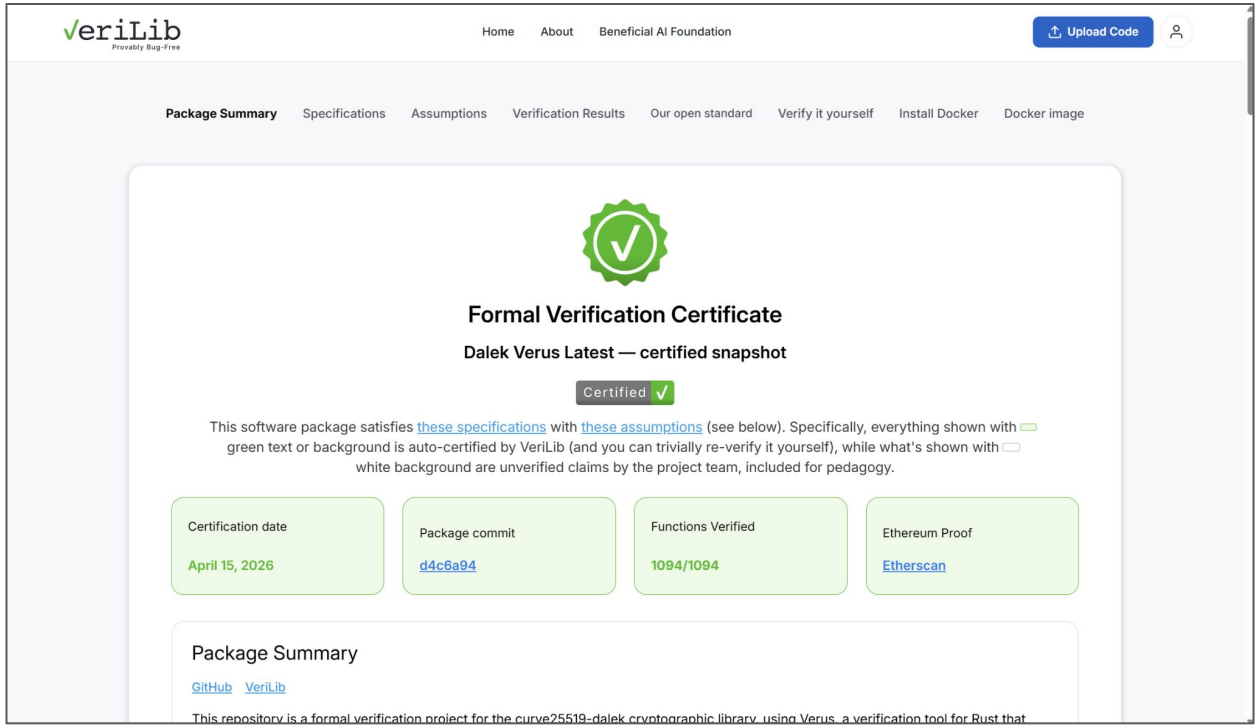
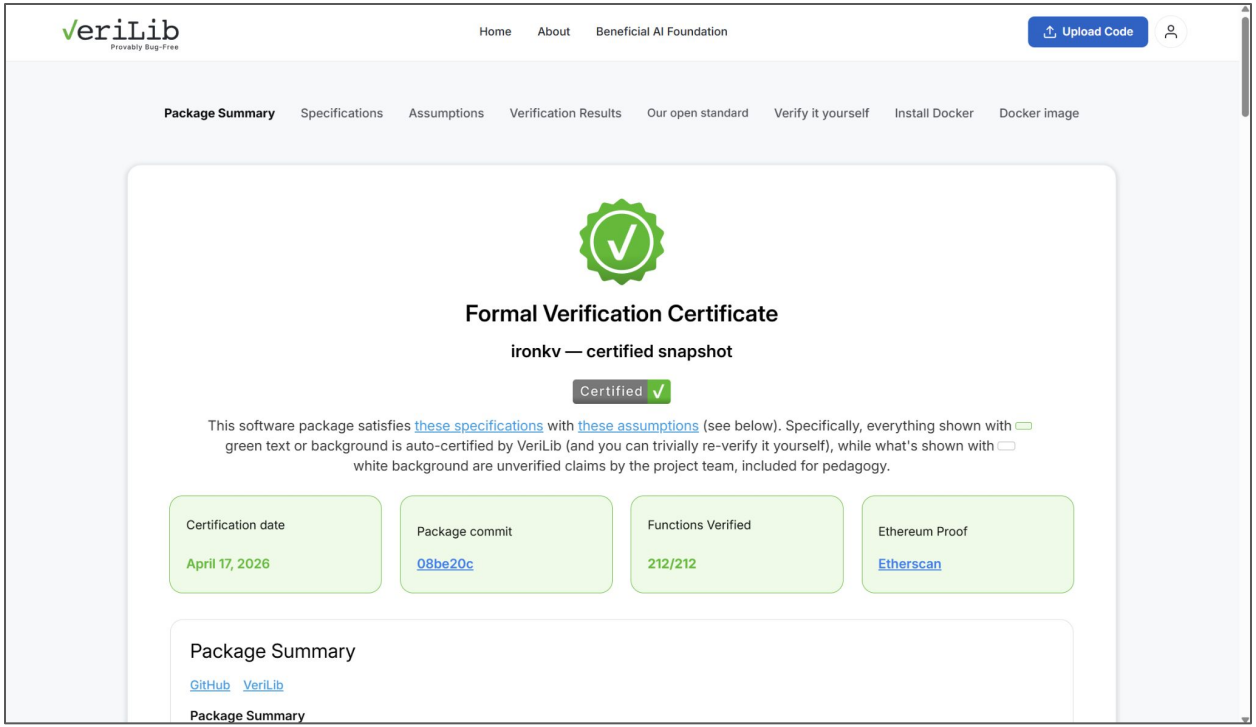

BENEFICIAL
AI FOUNDATION
Open tools and standards for scaling software verification
Max Tegmark and BAIF
Paris, April 20, 2026

Trust Page - Dalek-Verus



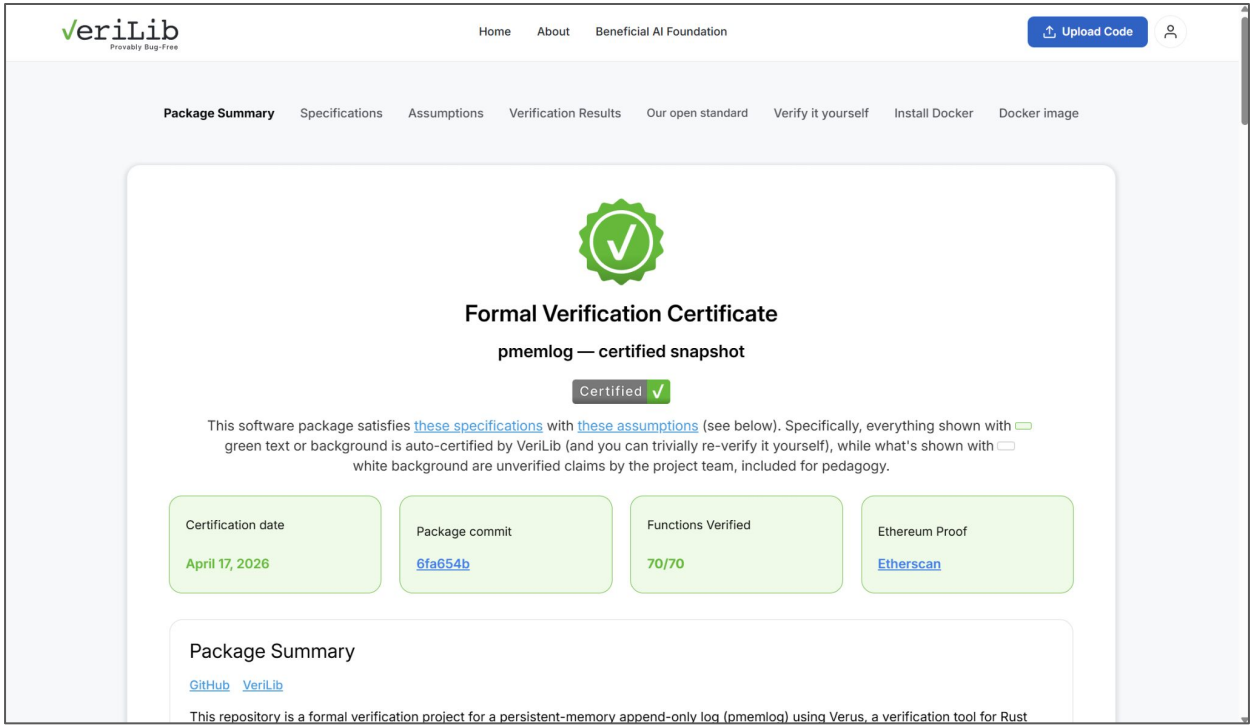
https://verilib.org/cert/5132

Trust Page - Verified-IronKV



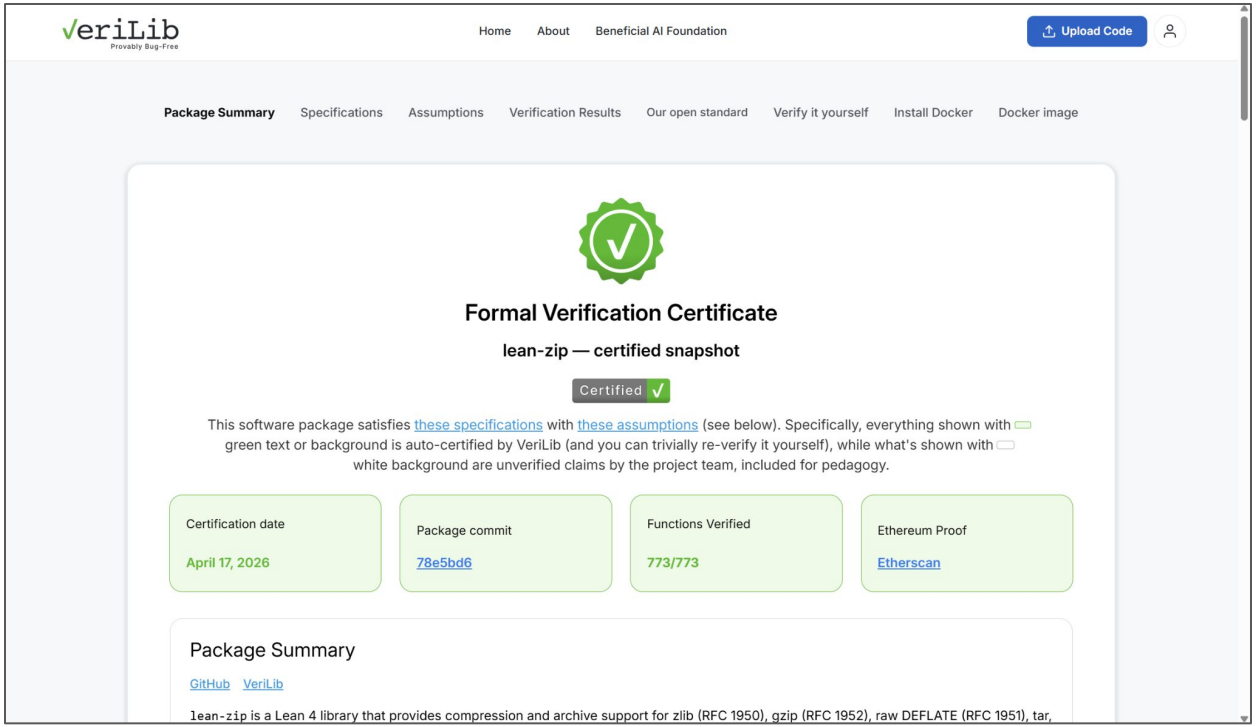
https://verilib.org/cert/5167

Trust Page - Microsoft-Verified-Storage-pmemlog



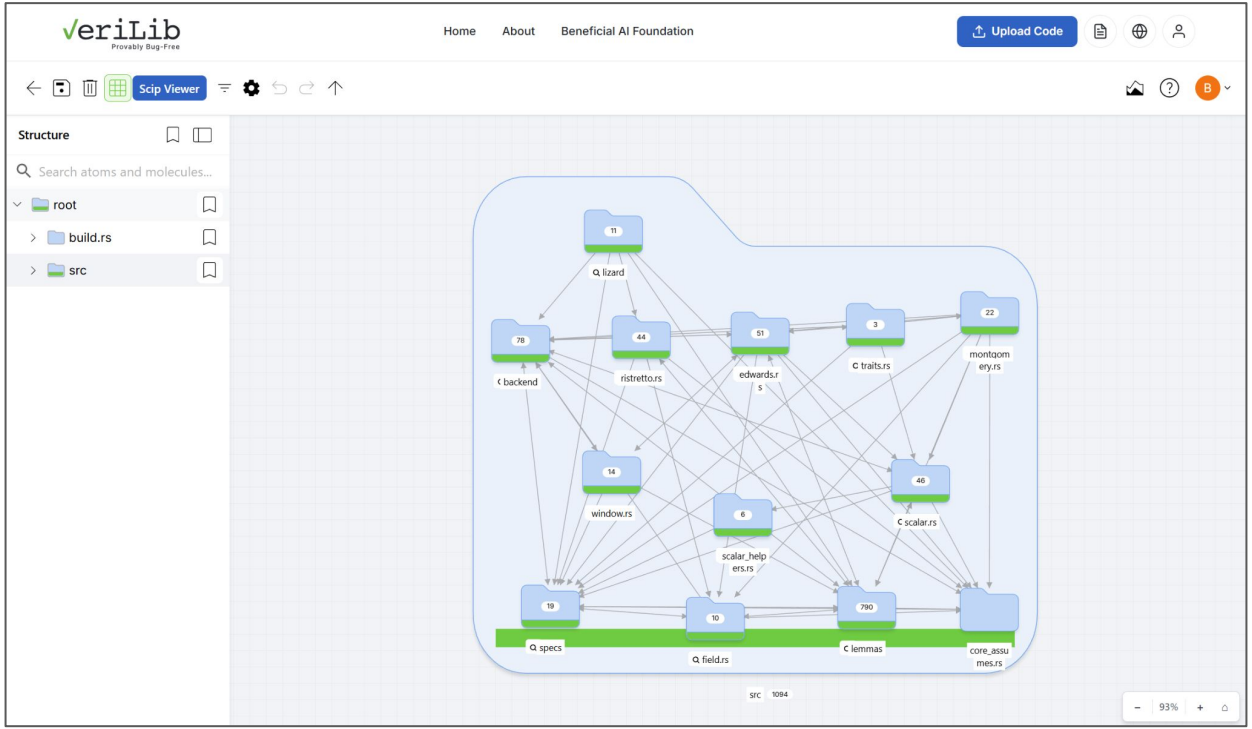
https://verilib.org/cert/5171

Trust Page - Lean-Zip



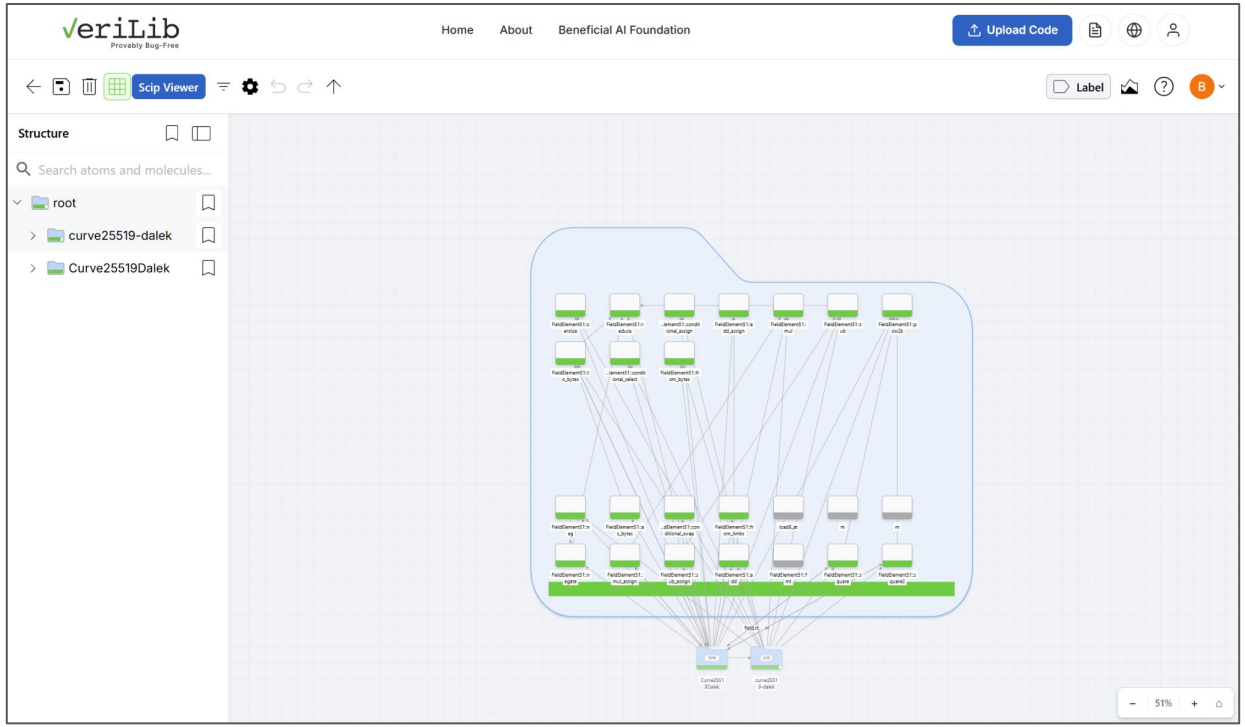
https://verilib.org/cert/5165

Dalek-Verus



https://verilib.org/repobrowser?id=5132

Dalek-Lean

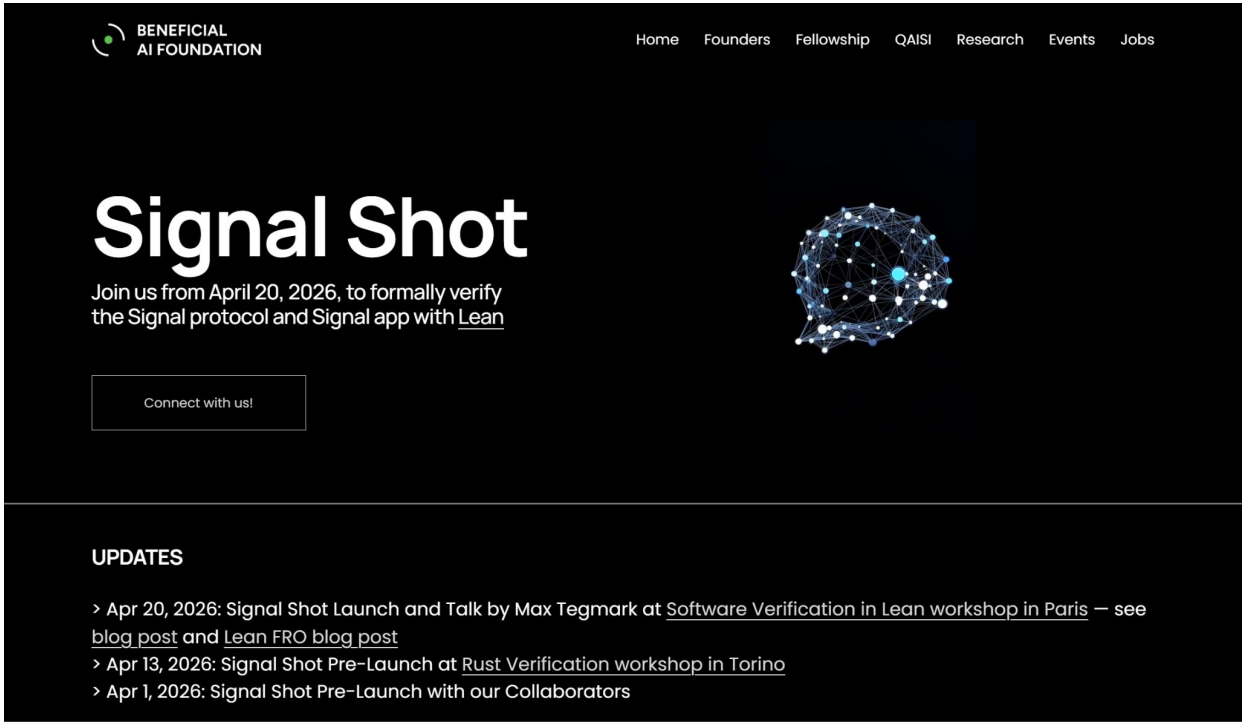


https://verilib.org/repobrowser?id=5174

Signal Shot

How We Are Organized

Signal Shot - Main Site



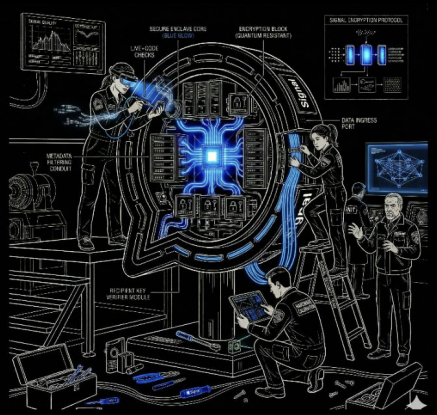
https://www.beneficialaifoundation.org/signal-shot

Signal Shot - Intro Blog

20 Apr

Signal Shot: One Giant Lean for Protocol Security

Can we prove that Signal's cryptography is secure — not just on paper, but in actual (source and proof) code? Signal Shot aims to find out. We're verifying the Signal protocol and its Rust implementation using the Lean Proof Assistant, in a public moonshot to show that formal verification of major applications is doable today. Think of it as the Liquid Tensor Experiment (LTE) for cryptography — where a complex math theorem was formalized and proved with the help of the mathematics community — except this time, we also have steadily improving AI on our side.

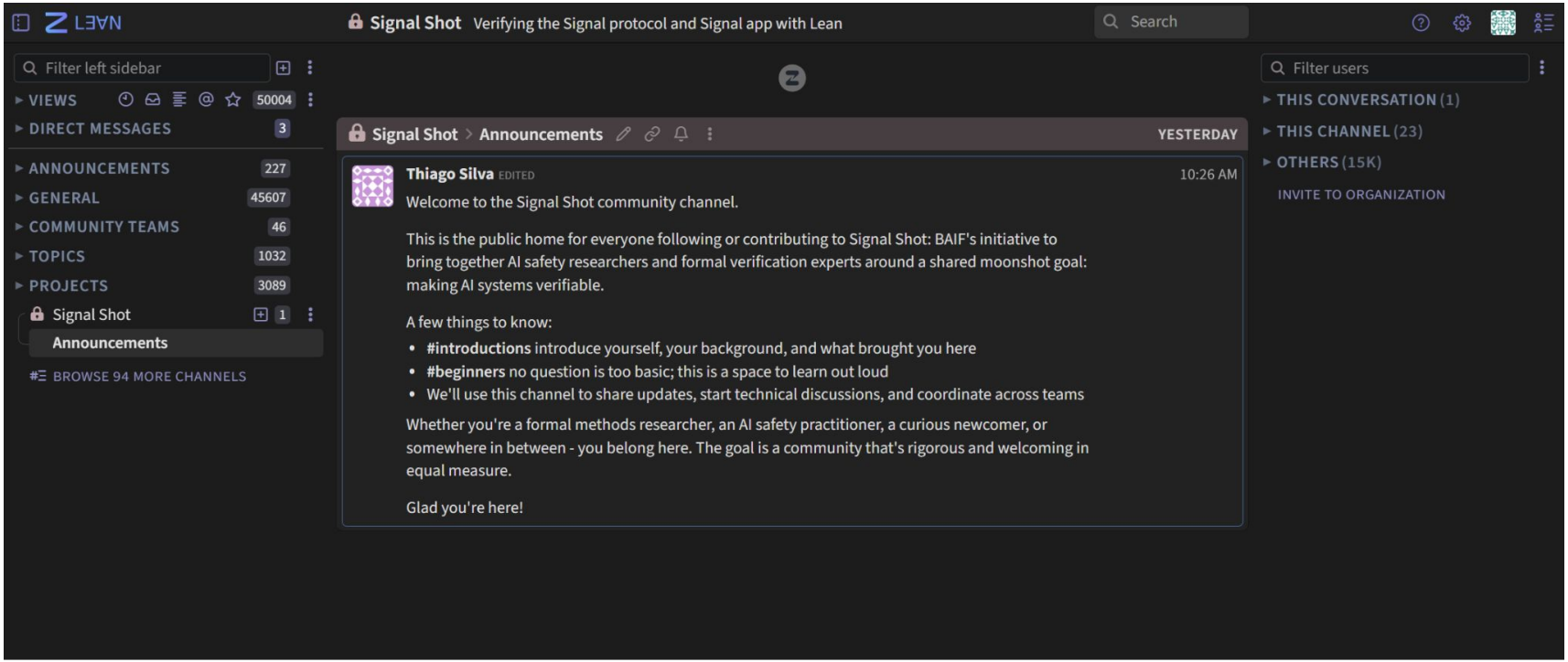


Here's how we'll do it: first, we write down a mathematical model of the protocol and prove its security properties. Then, we take Signal's existing Rust implementation and translate it to Lean using the Aeneas tool. Finally, we verify that the translated code actually satisfies the assumptions of our proven model — bridging the gap from theory to real-world code.

For more details visit our Signal Shot page.

https://www.beneficialaifoundation.org/blog/signal-shot

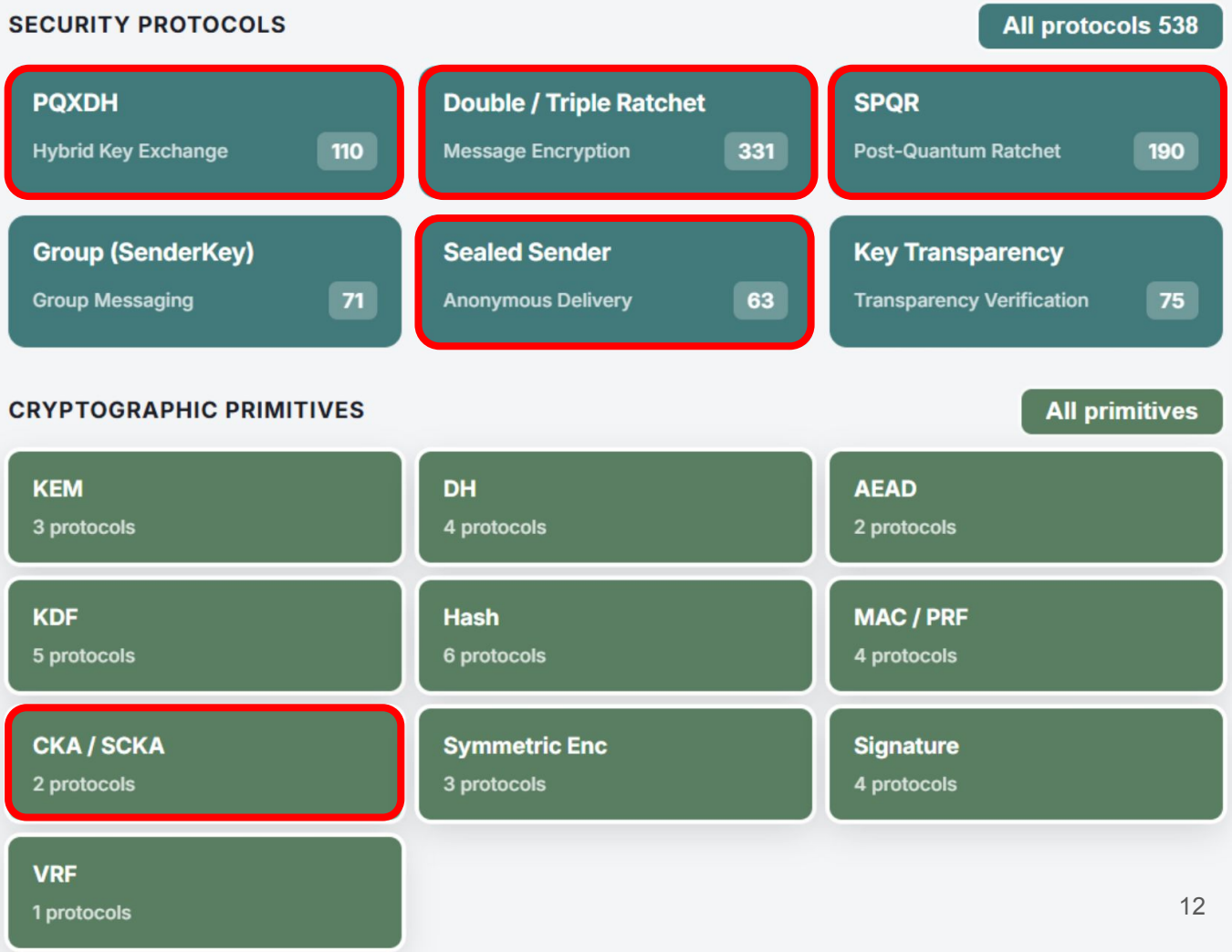
Signal Shot - Lean Prover Zulip Chat



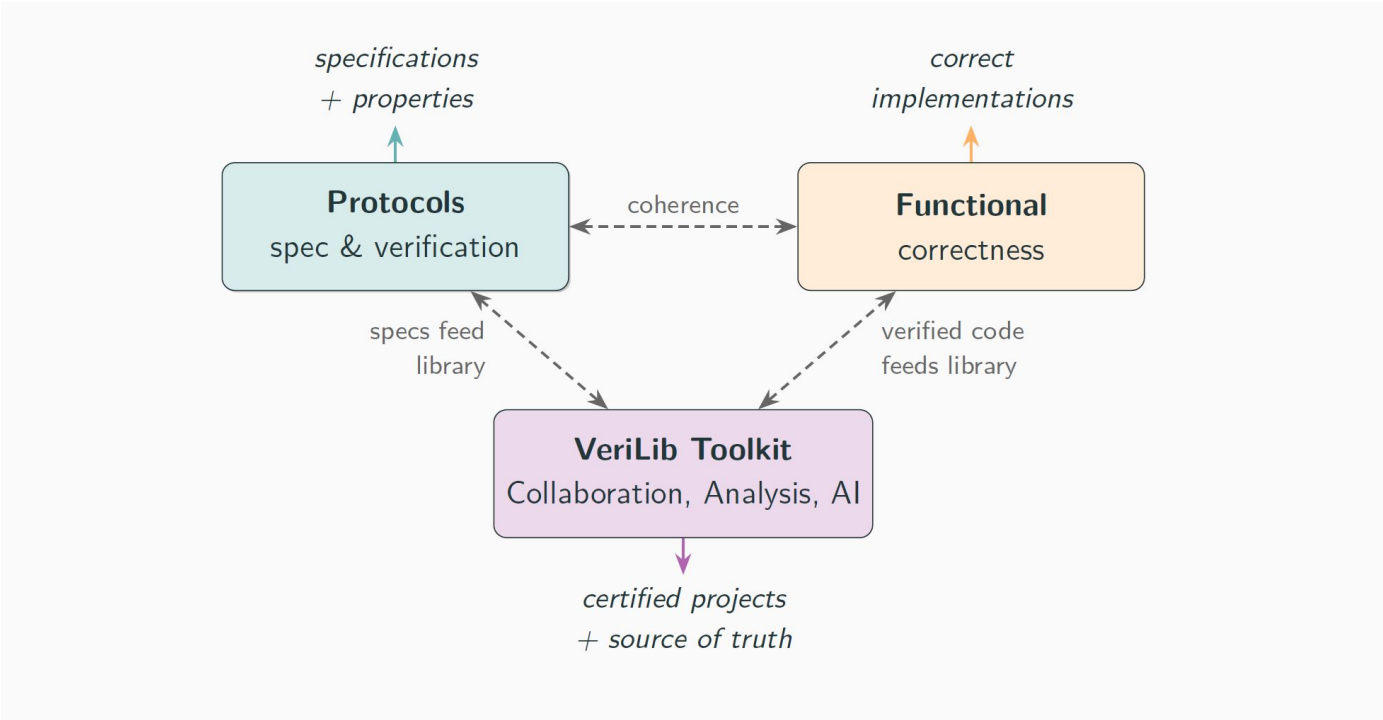
https://leanprover.zulipchat.com/#narrow/channel/583276-Signal-Shot

Verifying Protocols

Christiano Braga



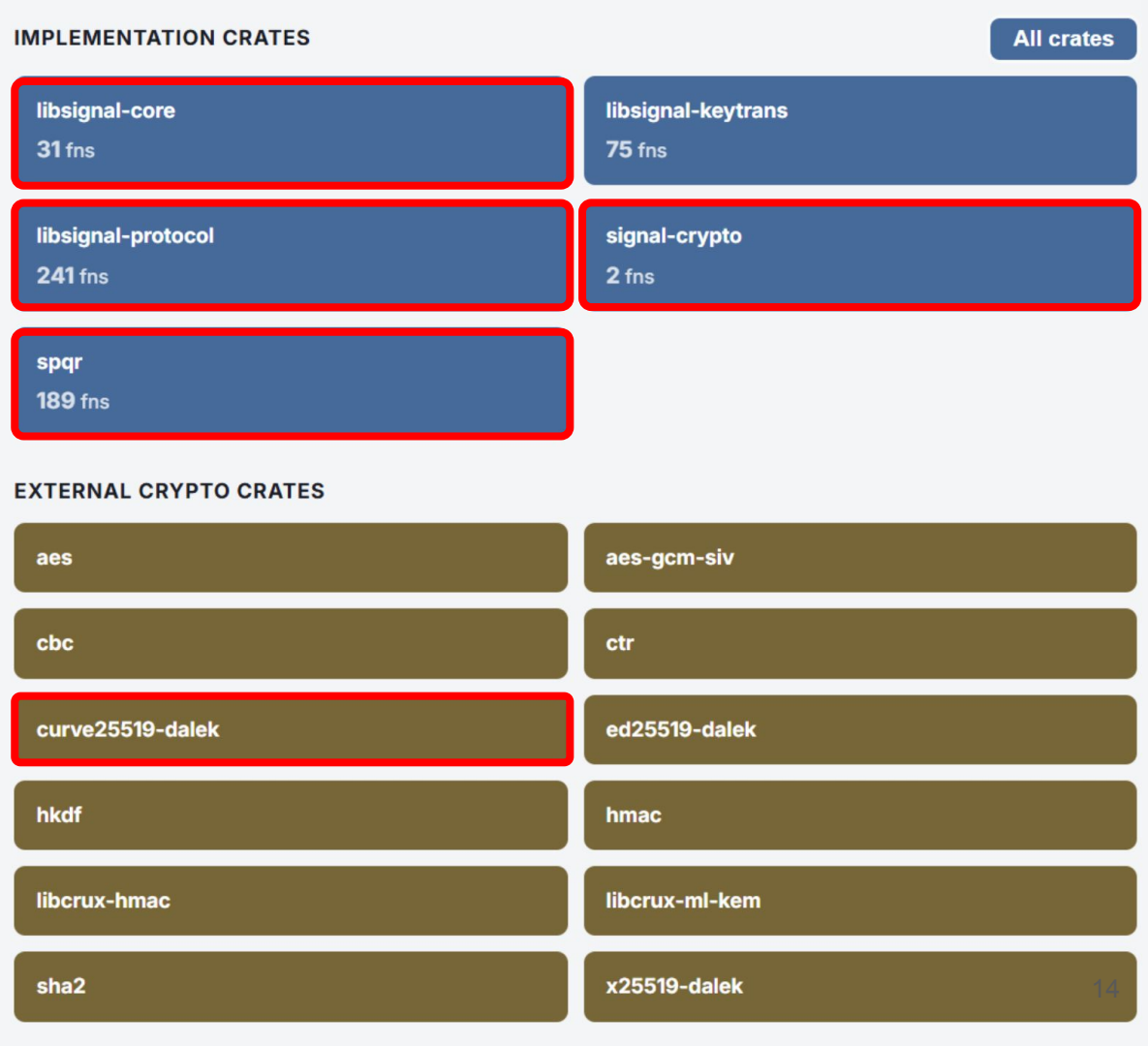
Protocol Specification and Verification



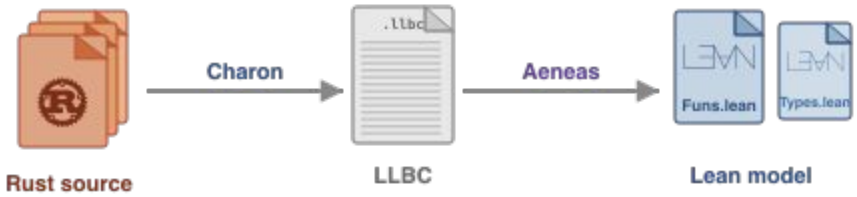
https://drive.google.com/file/d/1_IbCIN-euSruTCb29j99pIgV1FYrYwZ8

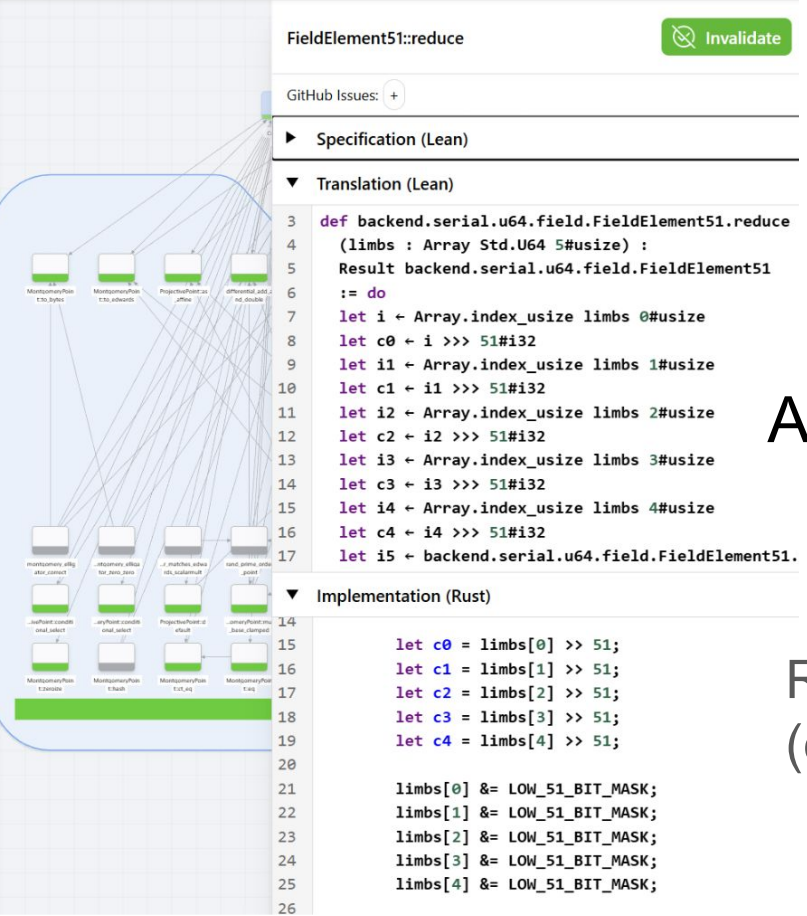
Verifying Functions

Jure Kukovec



Lean extraction pipeline





FieldElement51::reduce

Invalidate

GitHub Issues: +

▶ Specification (Lean)

▼ Translation (Lean)

```
3  def backend.serial.u64.field.FieldElement51.reduce
4    (limbs : Array Std.U64 5#usize) :
5    Result backend.serial.u64.field.FieldElement51
6    := do
7    let i ← Array.index_usize limbs 0#usize
8    let c0 ← i >>> 51#i32
9    let i1 ← Array.index_usize limbs 1#usize
10   let c1 ← i1 >>> 51#i32
11   let i2 ← Array.index_usize limbs 2#usize
12   let c2 ← i2 >>> 51#i32
13   let i3 ← Array.index_usize limbs 3#usize
14   let c3 ← i3 >>> 51#i32
15   let i4 ← Array.index_usize limbs 4#usize
16   let c4 ← i4 >>> 51#i32
17   let i5 ← backend.serial.u64.field.FieldElement51.
```

▼ Implementation (Rust)

```
15        let c0 = limbs[0] >> 51;
16        let c1 = limbs[1] >> 51;
17        let c2 = limbs[2] >> 51;
18        let c3 = limbs[3] >> 51;
19        let c4 = limbs[4] >> 51;
20
21        limbs[0] &= LOW_51_BIT_MASK;
22        limbs[1] &= LOW_51_BIT_MASK;
23        limbs[2] &= LOW_51_BIT_MASK;
24        limbs[3] &= LOW_51_BIT_MASK;
25        limbs[4] &= LOW_51_BIT_MASK;
26
```

Anatomy of a spec theorem and proof

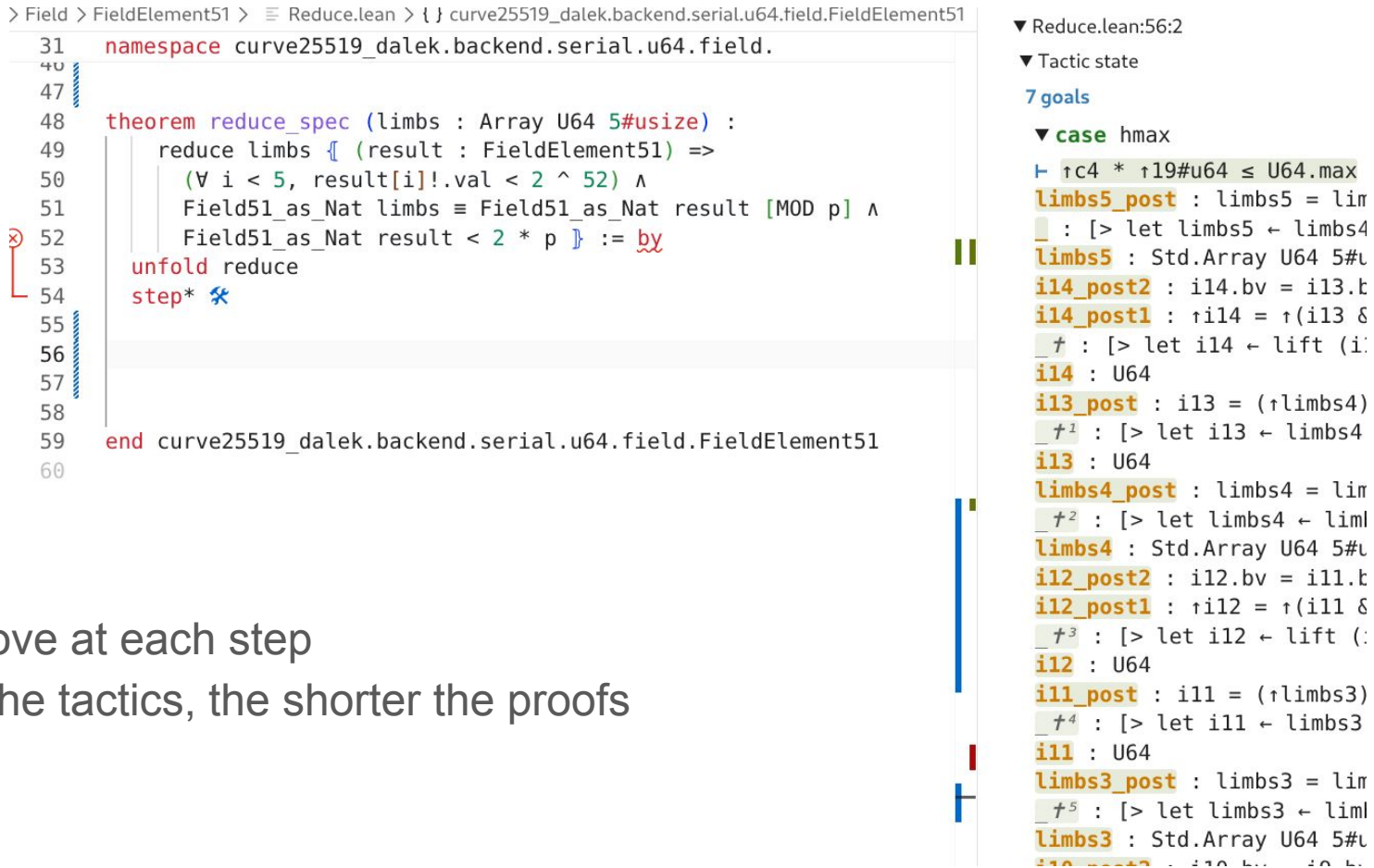
Rust source code (example from curve25519-dalek)

- Arithmetic for large numbers (using 5 64-bit numbers)
- Optimised for speed, not always easy to see correctness
- Is it correct, is the output equal to input MOD p?

Spec theorems

```
theorem reduce_spec (limbs : Array U64 5#usize) :
    reduce limbs ⦃ (result : FieldElement51) =>
      (∀ i < 5, result[i]!.val < 2 ^ 52) ∧
      Field51_as_Nat limbs ≡ Field51_as_Nat result [MOD p] ∧
      Field51_as_Nat result < 2 * p ⦄ := by
```

- Spec theorems are written in Lean (ultimate mathematical expressibility)
- No pre conditions (in this case)
- Postconditions:
  - Each of the 5 parts of the output are bounded
  - Equality MOD p
  - Output is bounded



- Goals to prove at each step
- The better the tactics, the shorter the proofs

Complete proof

Takes advantage of multiple tactics to close the goals

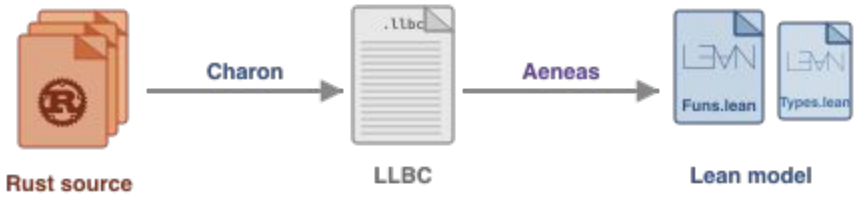
```
theorem reduce_spec (limbs : Array U64 5#usize) :
    reduce limbs ⦃ (result : FieldElement51) =>
      (∀ i < 5, result[i]!.val < 2 ^ 52) ∧
      Field51_as_Nat limbs ≡ Field51_as_Nat result [MOD p] ∧
      Field51_as_Nat result < 2 * p ⦄ := by
  unfold reduce
  step*
  · scalar_tac
  · simp [*]; scalar_tac
  · simp [*]; scalar_tac
  · simp [*]; scalar_tac
  · simp [*]; scalar_tac
  · simp [*]; scalar_tac
  constructor
  · intro i _
    interval_cases i
    all_goals
      simp_lists; simp [*]; scalar_tac
  · simp_lists
    simp [Nat.ModEq, Field51_as_Nat, Finset.sum_range_succ, p, *];
    scalar_tac
```

The role of AI (today)

- Humans write the specs ❗
- Coding agents can write proofs
  - Lean checks proofs for correctness
  - Humans check proofs for readability/maintainability/performance ❗

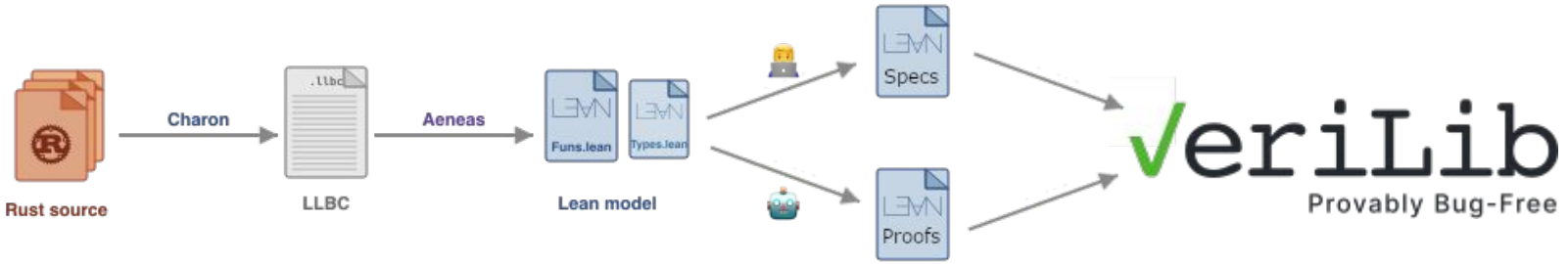
We want to go from this …

Lean extraction pieline

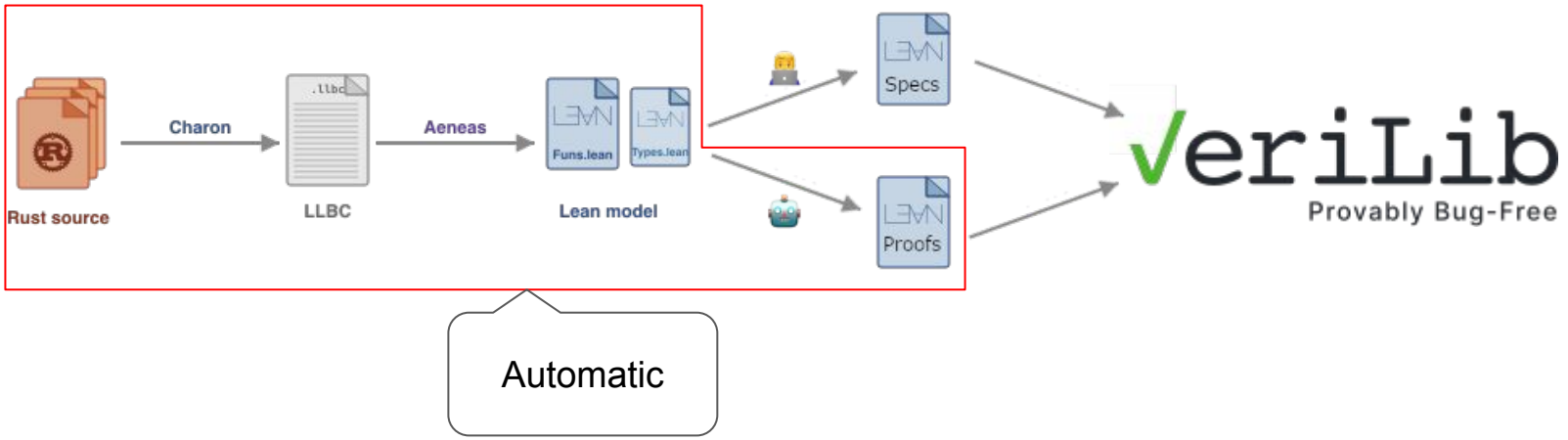


… to this

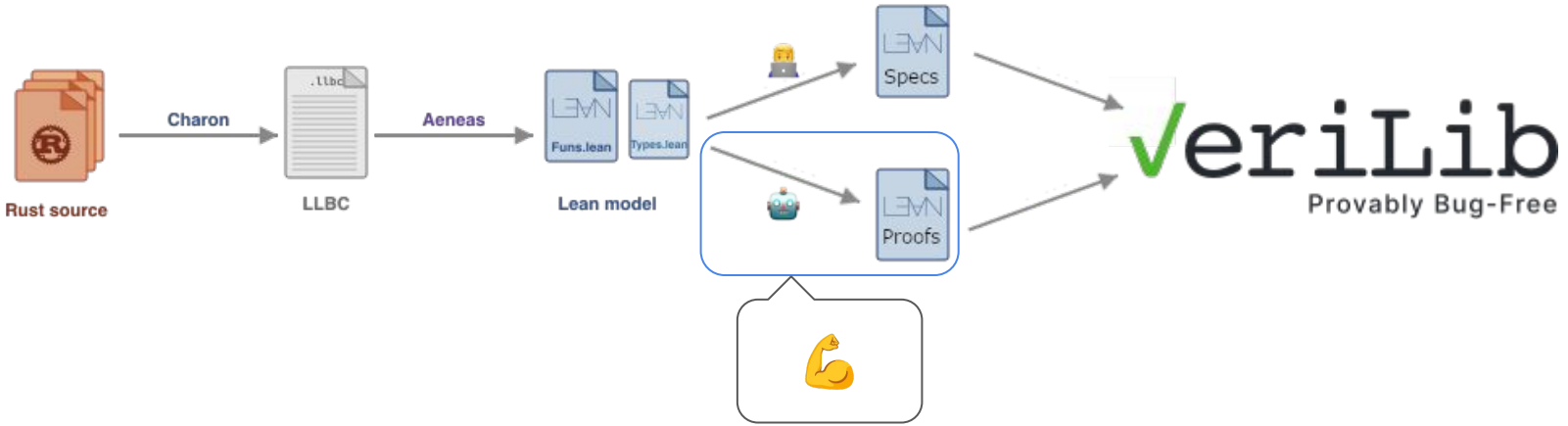
Full verification pipeline



Full verification pipeline

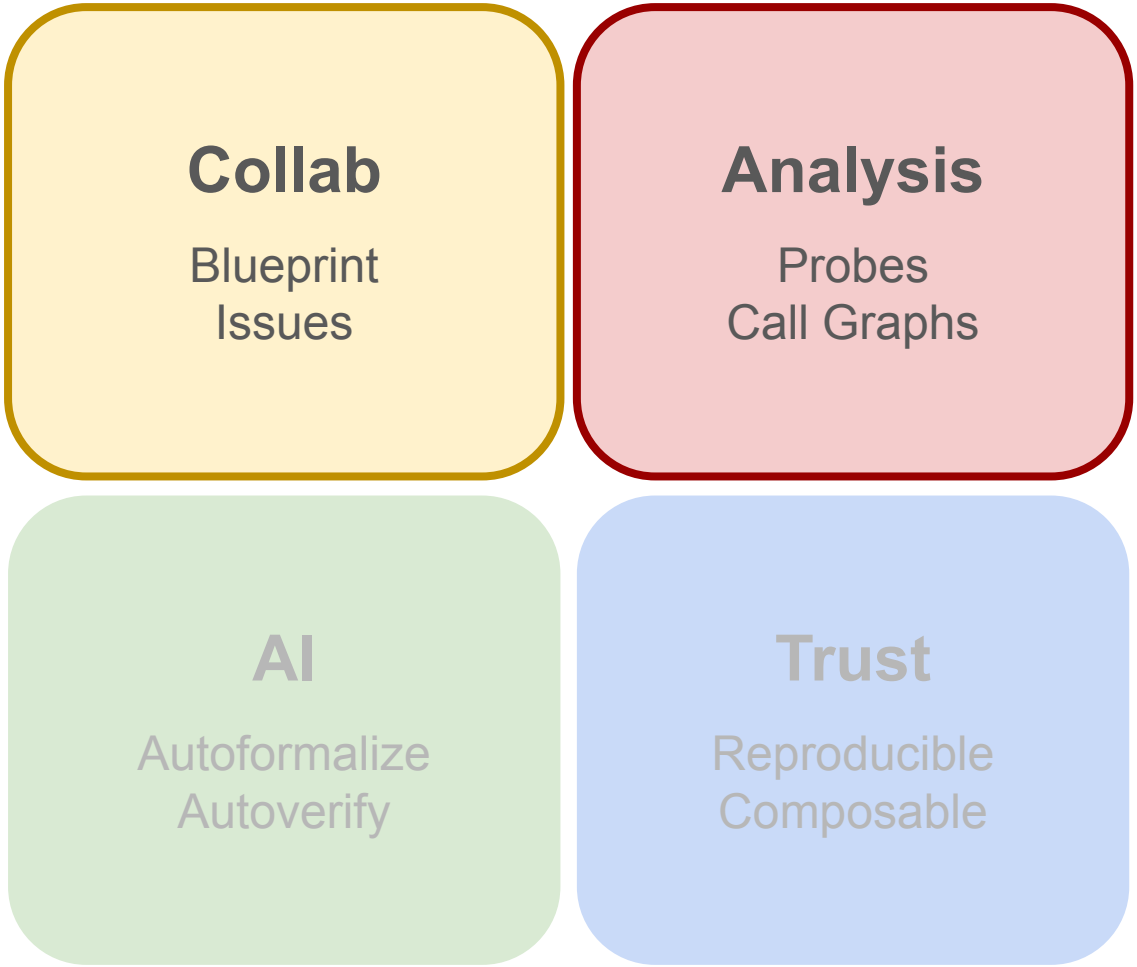


Full verification pipeline



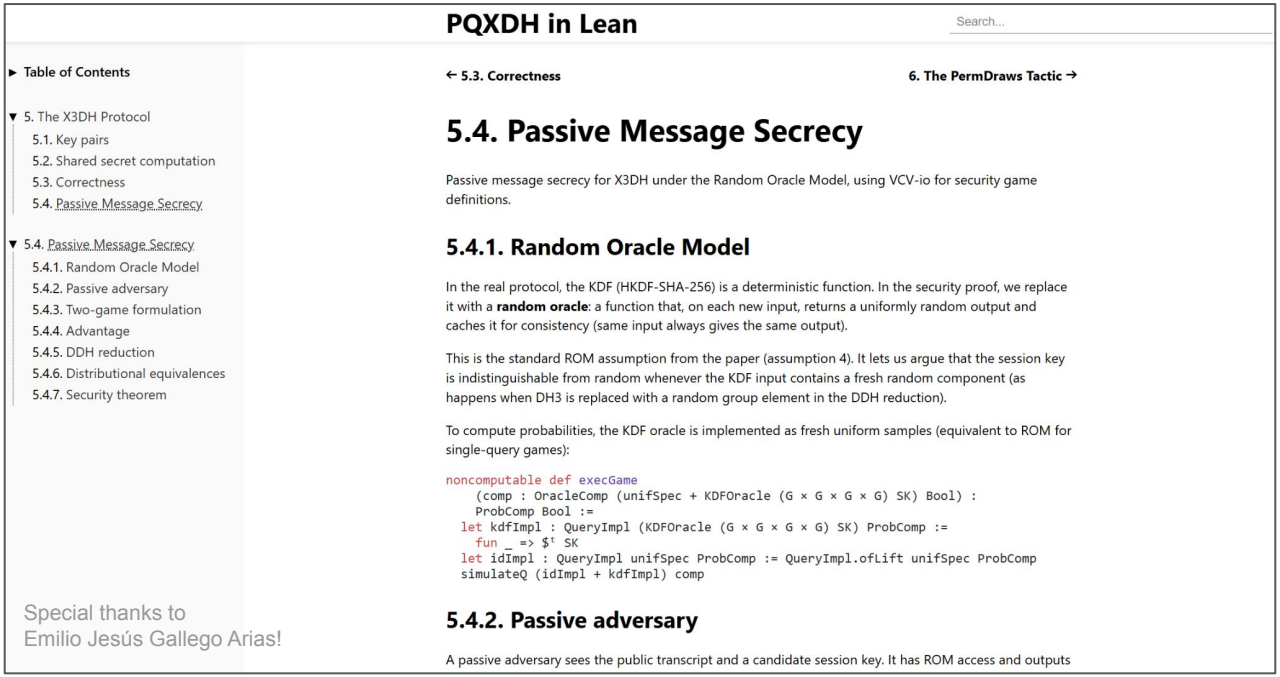
VeriLib Toolkit

Lacramioara Astefanoaei



Verso-Blueprint

How do you plan a verification project?



PQXDH in Lean

Search...

▸ Table of Contents

▾ 5. The X3DH Protocol
- 5.1. Key pairs
- 5.2. Shared secret computation
- 5.3. Correctness
- 5.4. Passive Message Secrecy

▾ 5.4. Passive Message Secrecy
- 5.4.1. Random Oracle Model
- 5.4.2. Passive adversary
- 5.4.3. Two-game formulation
- 5.4.4. Advantage
- 5.4.5. DDH reduction
- 5.4.6. Distributional equivalences
- 5.4.7. Security theorem

Special thanks to
Emilio Jesús Gallego Arias!

← 5.3. Correctness 6. The PermDraws Tactic →

5.4. Passive Message Secrecy

Passive message secrecy for X3DH under the Random Oracle Model, using VCV-io for security game definitions.

5.4.1. Random Oracle Model

In the real protocol, the KDF (HKDF-SHA-256) is a deterministic function. In the security proof, we replace it with a **random oracle**: a function that, on each new input, returns a uniformly random output and caches it for consistency (same input always gives the same output).

This is the standard ROM assumption from the paper (assumption 4). It lets us argue that the session key is indistinguishable from random whenever the KDF input contains a fresh random component (as happens when DH3 is replaced with a random group element in the DDH reduction).

To compute probabilities, the KDF oracle is implemented as fresh uniform samples (equivalent to ROM for single-query games):

```
noncomputable def execGame
    (comp : OracleComp (unifSpec + KDFOracle (G × G × G × G) SK) Bool) :
    ProbComp Bool :=
  let kdfImpl : QueryImpl (KDFOracle (G × G × G × G) SK) ProbComp :=
    fun _ => $ᵗ SK
  let idImpl : QueryImpl unifSpec ProbComp := QueryImpl.ofLift unifSpec ProbComp
  simulateQ (idImpl + kdfImpl) comp
```

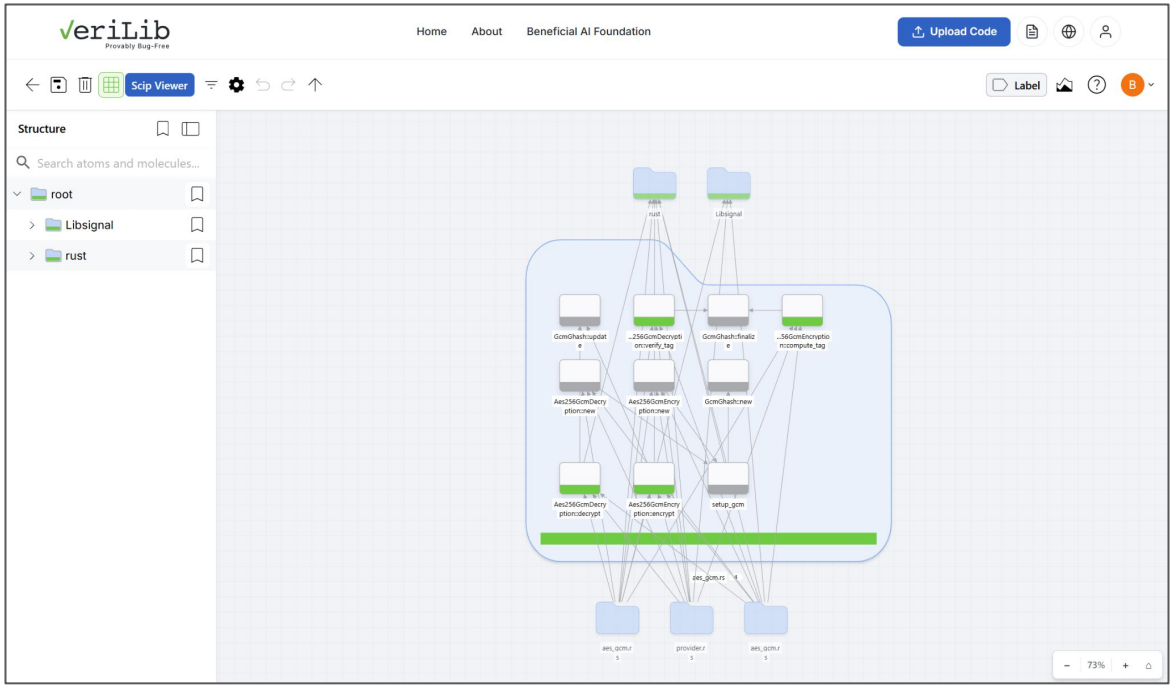
5.4.2. Passive adversary

A passive adversary sees the public transcript and a candidate session key. It has ROM access and outputs

https://beneficial-ai-foundation.github.io/PQXDH

VeriLib - `libsignal` (dependencies)

How do you display a verification project?

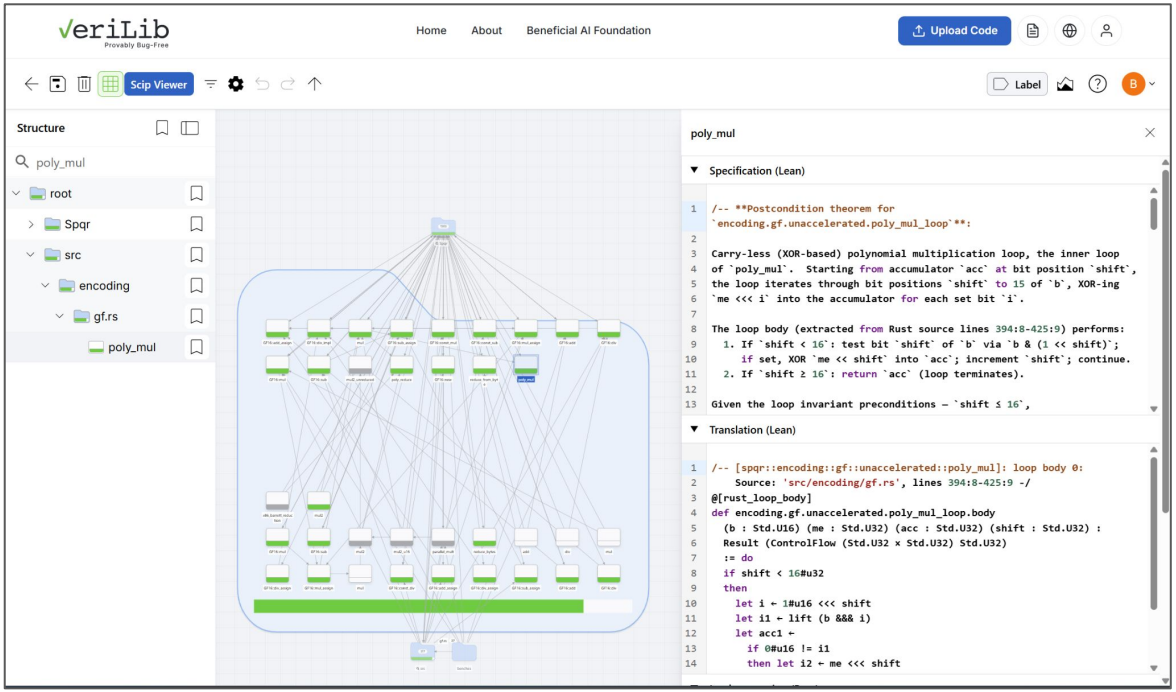


https://verilib.org/repobrowser?id=5172

VeriLib - SPQR (translations, specs, proofs)

How do you display a verification project?

src/encoding/
  gf.rs/add
  gf.rs/sub
  gf.rs/poly_mul
  polynomial.rs/serialize
  polynomial.rs/deserialize



https://verilib.org/repobrowser?id=5173

VeriLib and the Probes

How do you display a verification project?

Rust + Verus

Verified in-place with specs and proofs alongside the implementation.

`dalek-verus (VP:5050)`

Lean 4 Translation

Aeneas-transpiled version for independent proof efforts.

`dalek-lean (VD:4603)`

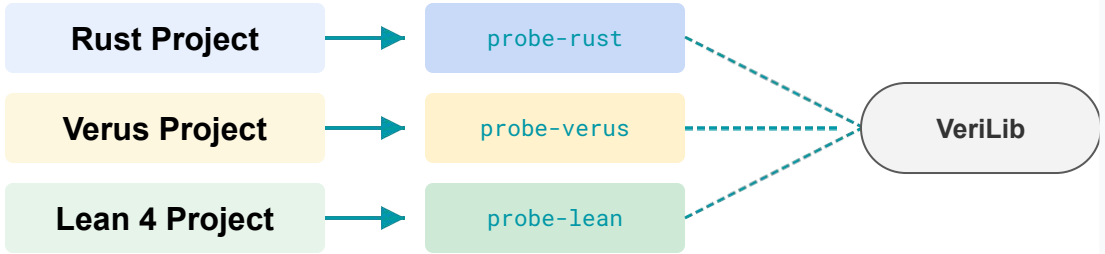
The Interoperability Challenge

- **Call Graphs:** What functions exist and what do they call?
- **Spec Extraction:** Which have specifications? What do they say?
- **Verification Status:** Which are verified? Which fail? Which use *sorry*?
- **Cross-Language Links:** How do Rust definitions map to Lean?

No single tool answers all of these. Language analyzers produce siloed data in incompatible formats with no standard interchange format.

The Probe Ecosystem

"5 tools, 1 schema, 1 merge operator"



Tool	Extraction Data
`probe-rust`	Call graph (SCIP) (1)
`probe-verus`	(1)+ Specs + Verification
`probe-lean`	(1)+ Deps + Sorry check
`probe-aeneas`	Probe-rust + probe-aeneas via Cross-lang mappings
`merge op`	Graph Composition

Unified Output: Schema 2.0 JSON

Self-describing envelopes with full provenance data. The merge operator replaces stubs with real data and injects cross-language edges for deterministic results.

- **Deterministic:** Same inputs always yield identical unified graphs.
- **Aeneas Mapping:** Automated link between Rust implementation and Lean 4 proofs.
- **Scalable:** Schema handles small modules to full cryptographic libraries.

Key Takeaways: What makes this work

Extending verification visibility through modular extraction and explicit trust

1. Specialized Tools

Each tool stays in its lane. `probe-rust` sees only Rust functions, while `probe-lean` focuses on Lean declarations.

No tool tries to understand another language—they each extract what they can from their own world.

2. Bridging the Gap

`probe-aeneas` reads `functions.json` to generate cross-language mappings.

The merge operator applies these generically to stitch a single unified graph across both languages, linking Rust implementation to Lean 4 proofs.

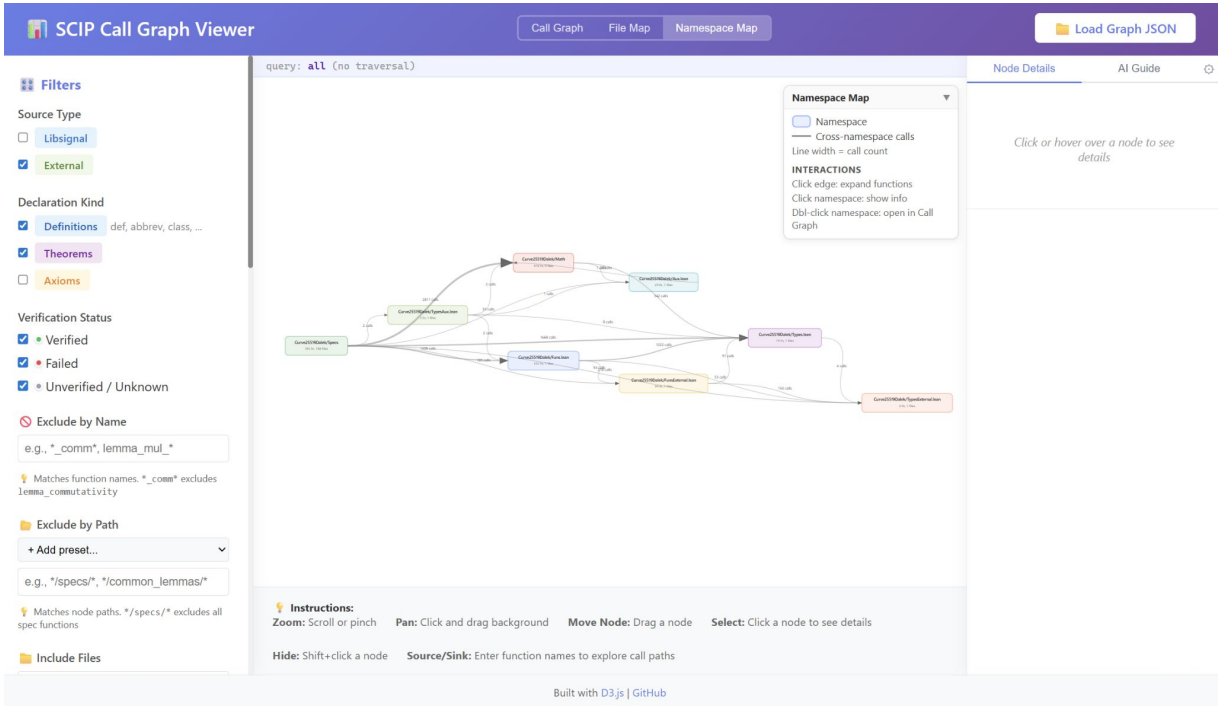
3. Explicit Trust Base

Every `sorry`, `admit`, and `axiom` is surfaced with a classified reason.

VeriLib doesn't just show "verified"—it reveals what you're trusting and why, making the verification surface transparent.

Call Graph Explorer

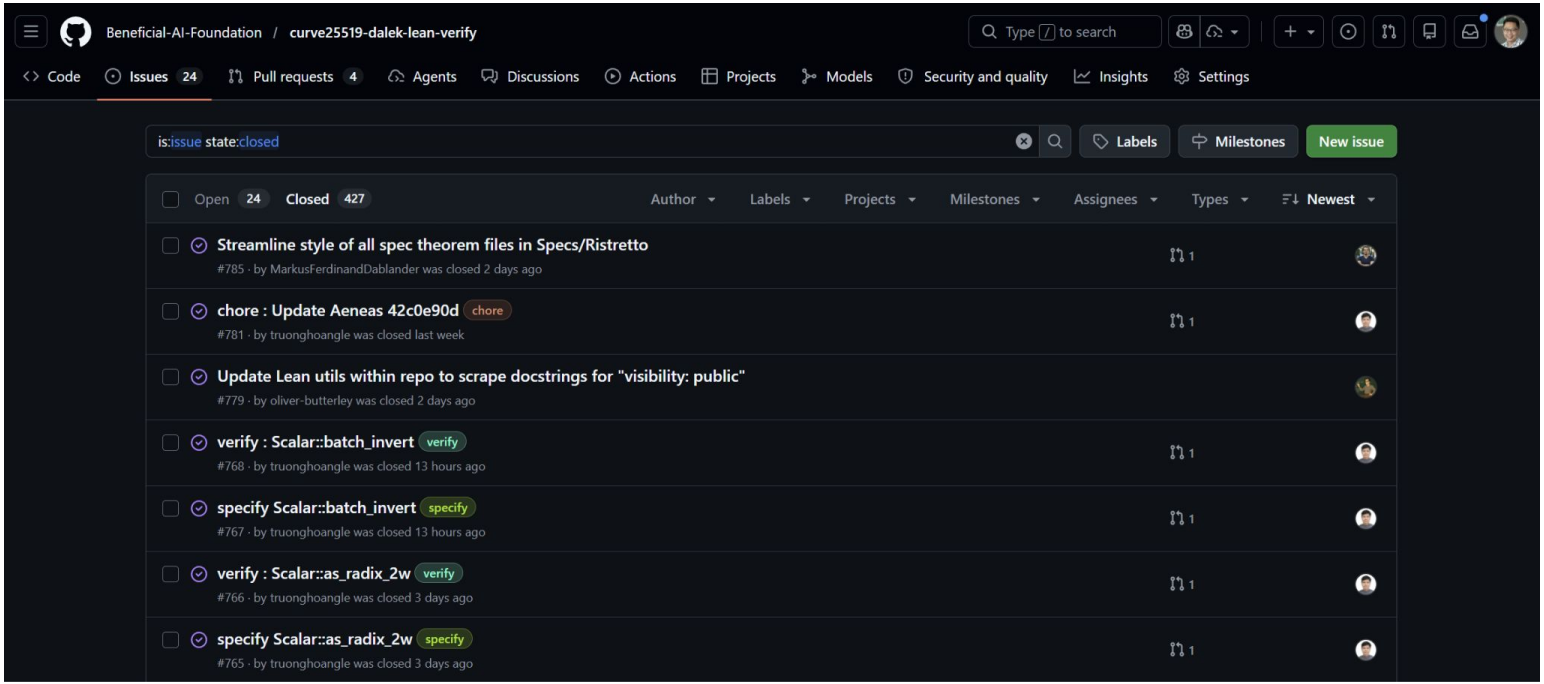
Visualizing and understanding dependencies with verification statuses



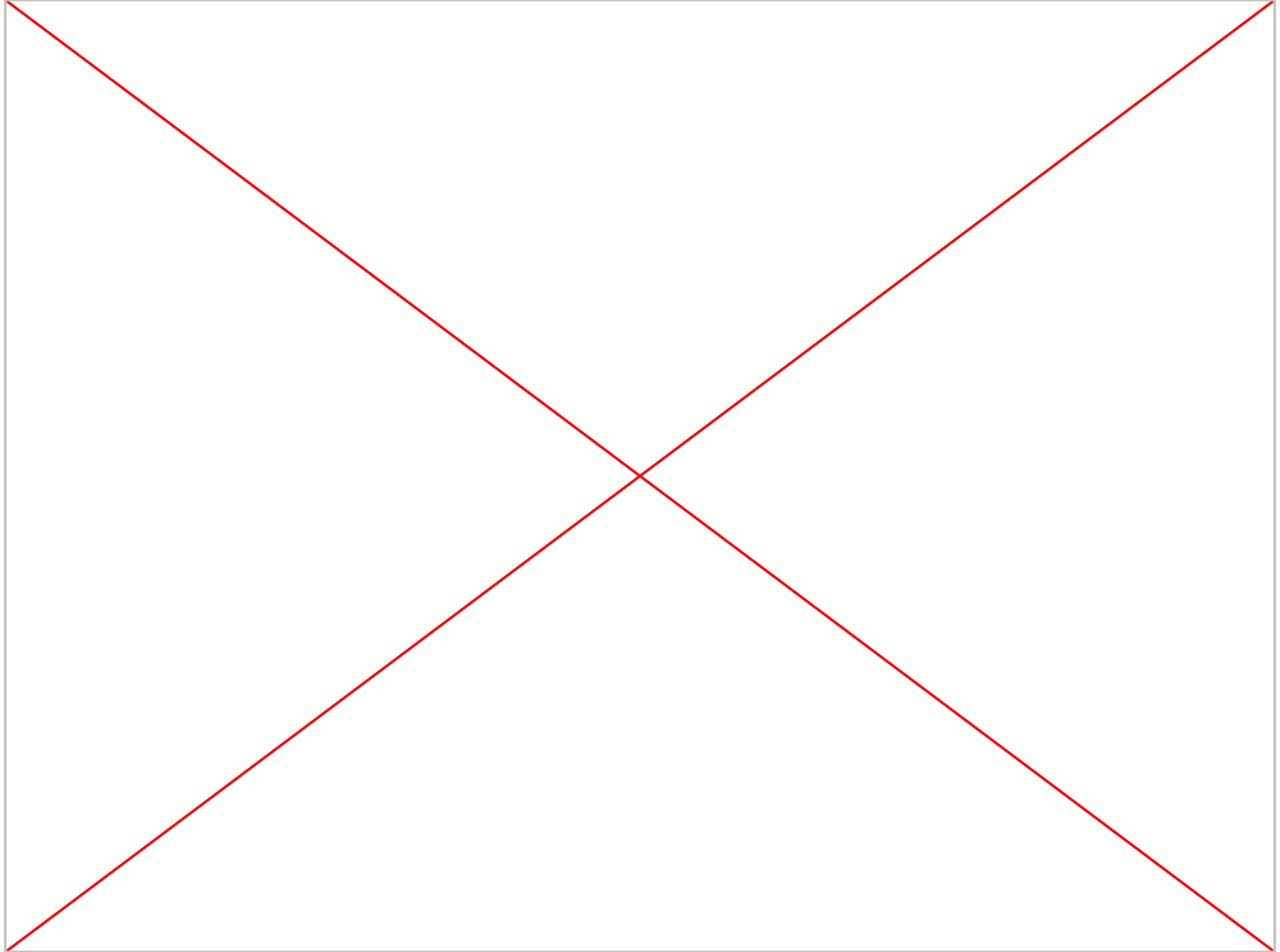
https://beneficial-ai-foundation.github.io/curve25519-dalek-lean-verify/callgraph

GitHub Issues

How do you coordinate a verification project?

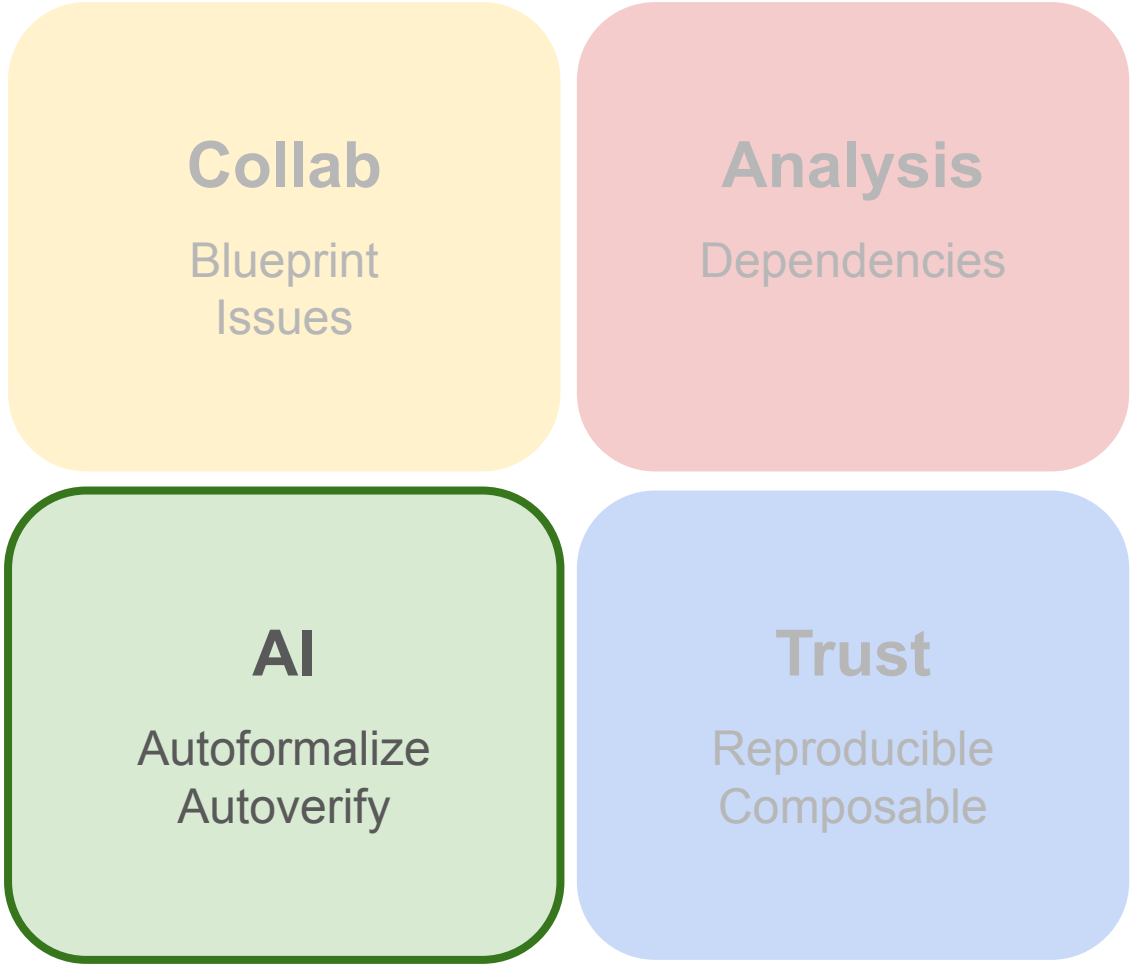


https://github.com/Beneficial-AI-Foundation/curve25519-dalek-lean-verify/issues

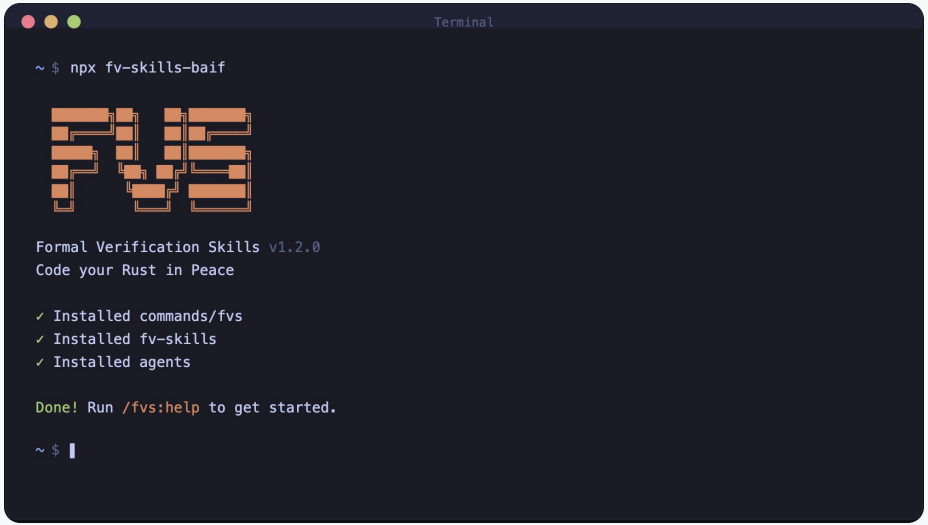


VeriLib Toolkit

Alessandro D'Angelo



AI Tools: Formal Verification Skills



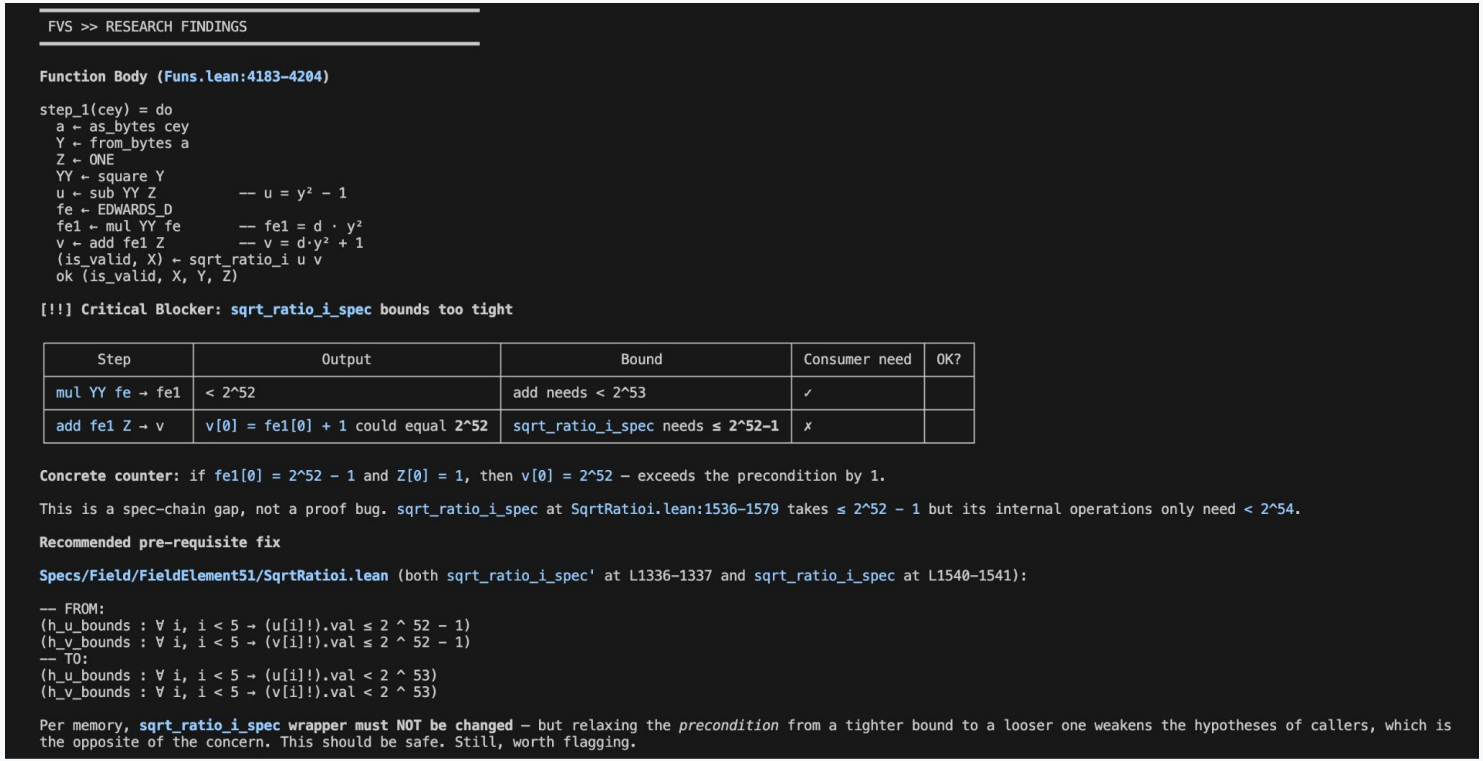
Open Source Repository

Explore the full source code and documentation for the formal verification skill set on GitHub.

Check it out at:

https://github.com/Beneficial-AI-Foundation/formal-verification-skills

AI Tools: Formal Verification Skills



FVS >> RESEARCH FINDINGS

Function Body (Funs.lean:4183–4204)

```
step_1(cey) = do
  a ← as_bytes cey
  Y ← from_bytes a
  Z ← ONE
  YY ← square Y
  u ← sub YY Z          -- u = y² − 1
  fe ← EDWARDS_D
  fe1 ← mul YY fe       -- fe1 = d · y²
  v ← add fe1 Z         -- v = d·y² + 1
  (is_valid, X) ← sqrt_ratio_i u v
  ok (is_valid, X, Y, Z)
```

[!!] Critical Blocker: sqrt_ratio_i_spec bounds too tight

Step	Output	Bound	Consumer need	OK?
mul YY fe → fe1	< 2^52	add needs < 2^53	✓	
add fe1 Z → v	v[0] = fe1[0] + 1 could equal **2^52**	sqrt_ratio_i_spec needs **≤ 2^52−1**	✗	

Concrete counter: if fe1[0] = 2^52 − 1 and Z[0] = 1, then v[0] = 2^52 — exceeds the precondition by 1.

This is a spec-chain gap, not a proof bug. sqrt_ratio_i_spec at SqrtRatioi.lean:1536–1579 takes ≤ 2^52 − 1 but its internal operations only need < 2^54.

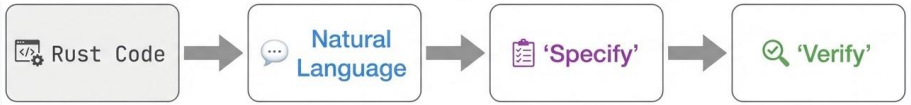
Recommended pre-requisite fix

Specs/Field/FieldElement51/SqrtRatioi.lean (both sqrt_ratio_i_spec' at L1336–1337 and sqrt_ratio_i_spec at L1540–1541):

```
-- FROM:
(h_u_bounds : ∀ i, i < 5 → (u[i]!).val ≤ 2 ^ 52 − 1)
(h_v_bounds : ∀ i, i < 5 → (v[i]!).val ≤ 2 ^ 52 − 1)
-- TO:
(h_u_bounds : ∀ i, i < 5 → (u[i]!).val < 2 ^ 53)
(h_v_bounds : ∀ i, i < 5 → (v[i]!).val < 2 ^ 53)
```

Per memory, **sqrt_ratio_i_spec wrapper must NOT be changed** — but relaxing the *precondition* from a tighter bound to a looser one weakens the hypotheses of callers, which is the opposite of the concern. This should be safe. Still, worth flagging.

AI Tools: Formal Verification Skills



.claude > fv-skills > workflows > lean-specify.md

```
<purpose>
Orchestrate specification generation for a single Lean function using two-phase
subagent dispatch (research -> execute).

Takes a verification target (function name), dispatches fvs-researcher to gather
context (Funs.lean, Types.lean, Rust source, existing stubs, similar specs), then
dispatches fvs-executor to write the spec file.

Output: Specs/{path}/{FunctionName}.lean with @[step] theorem and sorry placeholder.
</purpose>
```

Skill Specification & Generation

.claude > fv-skills > workflows > lean-verify.md

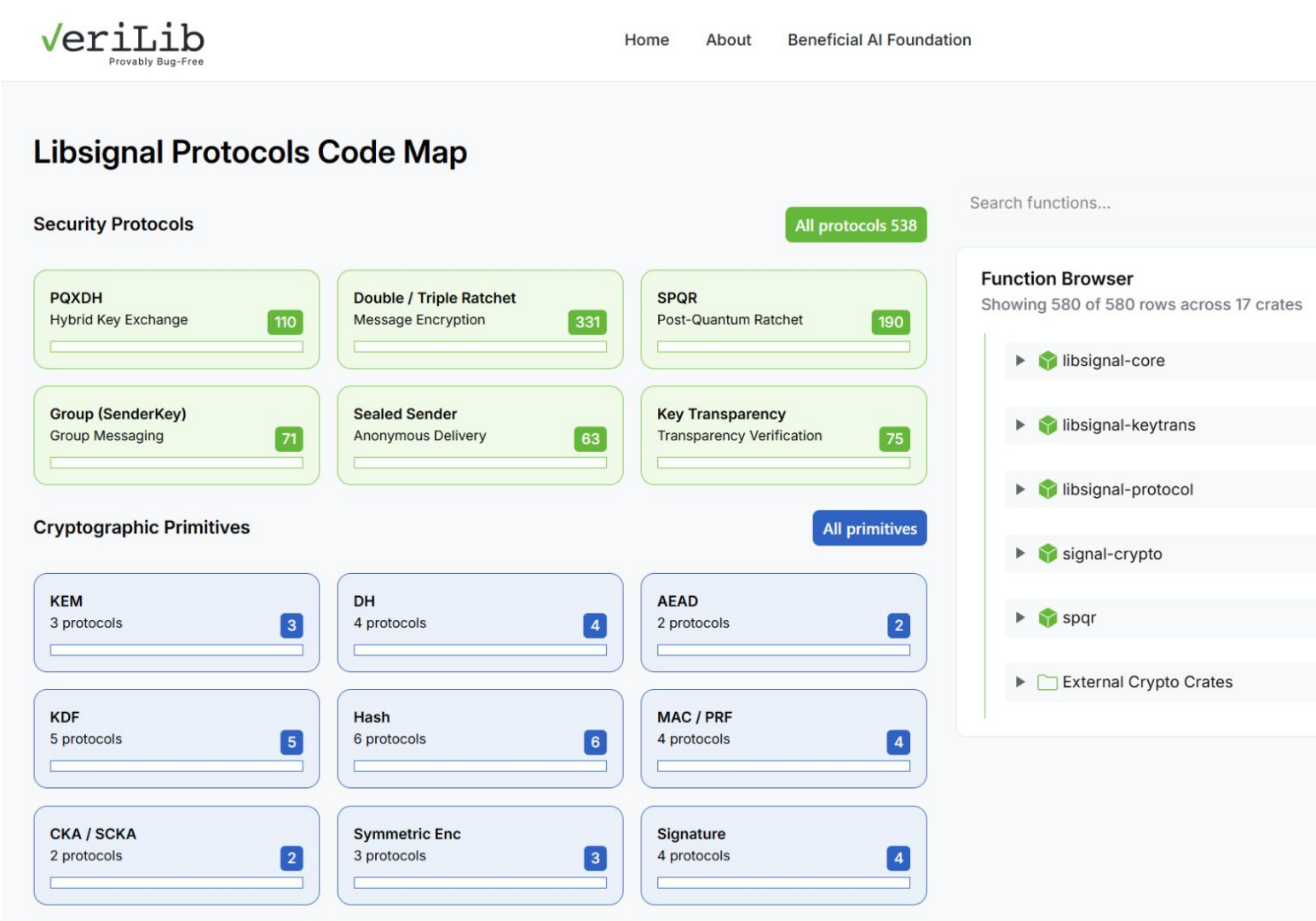
```
<purpose>
Orchestrate interactive proof development for a Lean specification using two-phase
subagent dispatch (research -> iterative execute).

Dispatches fvs-researcher to analyze sorry locations and recommend proof strategies,
then iteratively dispatches fvs-executor to replace each sorry ONE AT A TIME with
small tactic blocks. The user checks Lean compiles between each step.

This is the most interactive workflow -- it feels like pair programming. Small changes,
user approval, Lean compile check, repeat.

Output: Spec file with sorry replaced by complete proof, or clear report of stuck goals.
</purpose>
```

Interactive Verification & Analysis

Orchestrating specialized AI agents to generate, specify, and verify formal logic for high-assurance systems.

Next Steps

Shaowei Lin



Signal Shot - Stages

Stage 1A - Verifying Protocols
Abstract protocol and attacker models; Security properties
`PQXDH, DoubleRatchet, SPQR, CKA, Crypto, Credentials`

Stage 1B - Verifying Functions
Aeneas translation; Verifying shovel-ready functions
`libsignal-core, libsignal-protocol, signal-crypto, spqr`

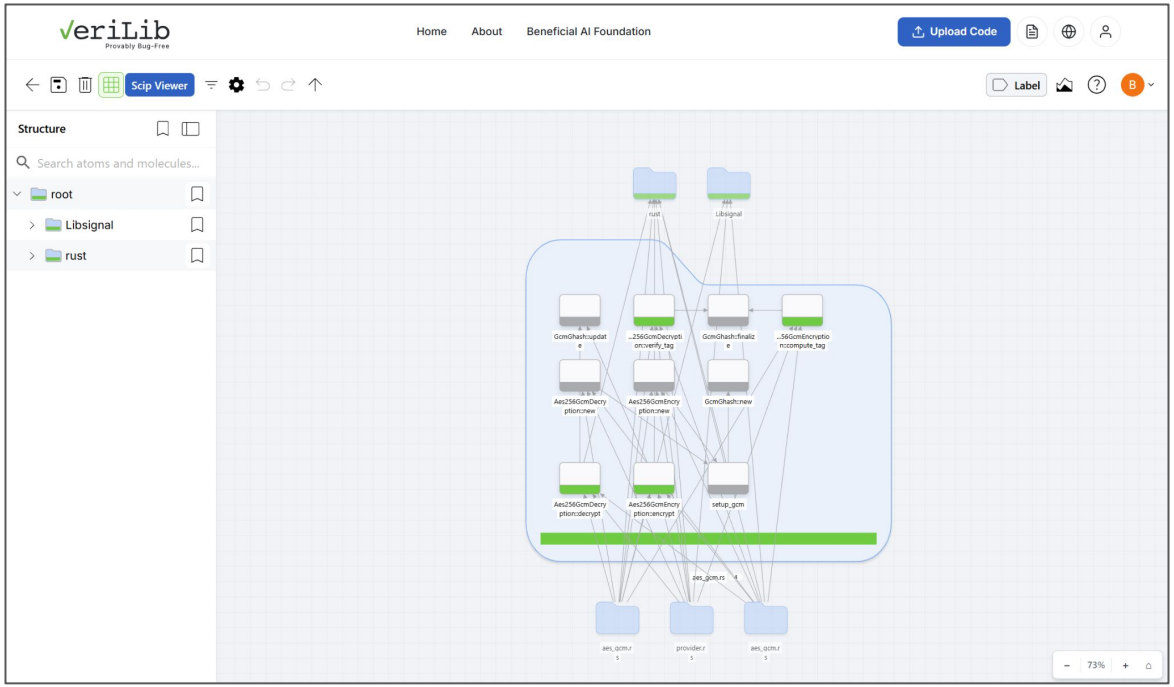
Stage 2 - Bridging Protocols and Functions

Stage 3A - Session Management

Stage 3B - Protocol Composition

VeriLib - `libsignal` (dependencies)

How do you display a verification project?



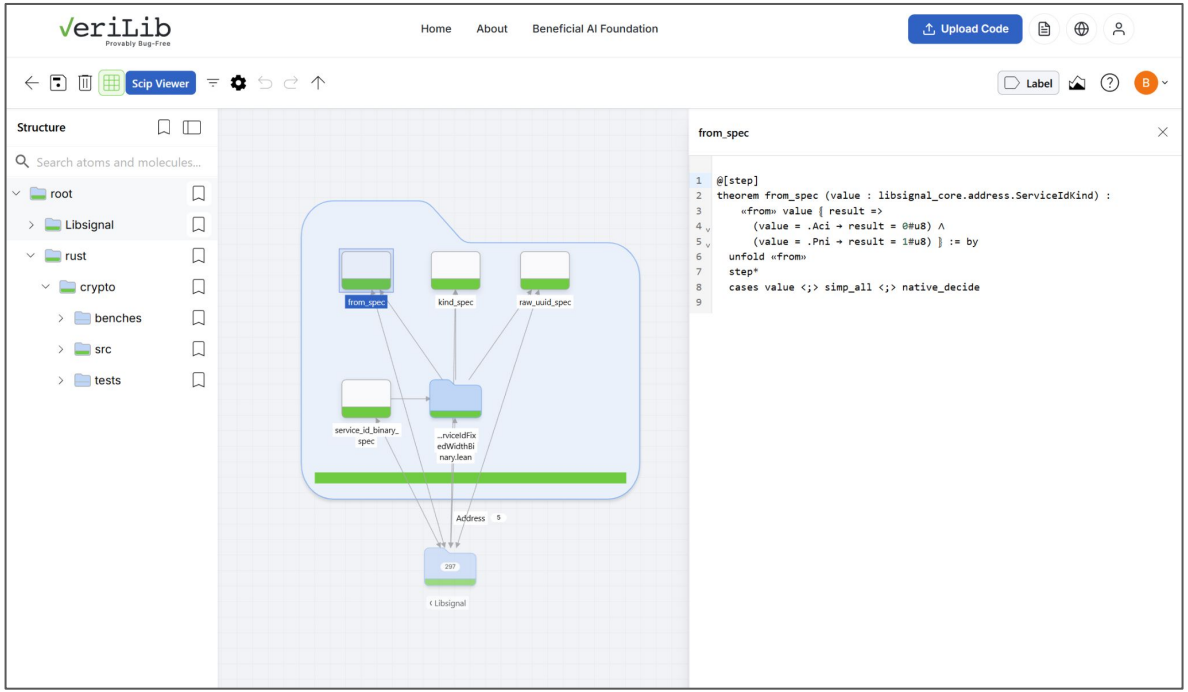
https://verilib.org/repobrowser?id=5172

VeriLib - `libsignal` (translations, specs, proofs)

How do you display a verification project?

Help us fill out specs and proofs!

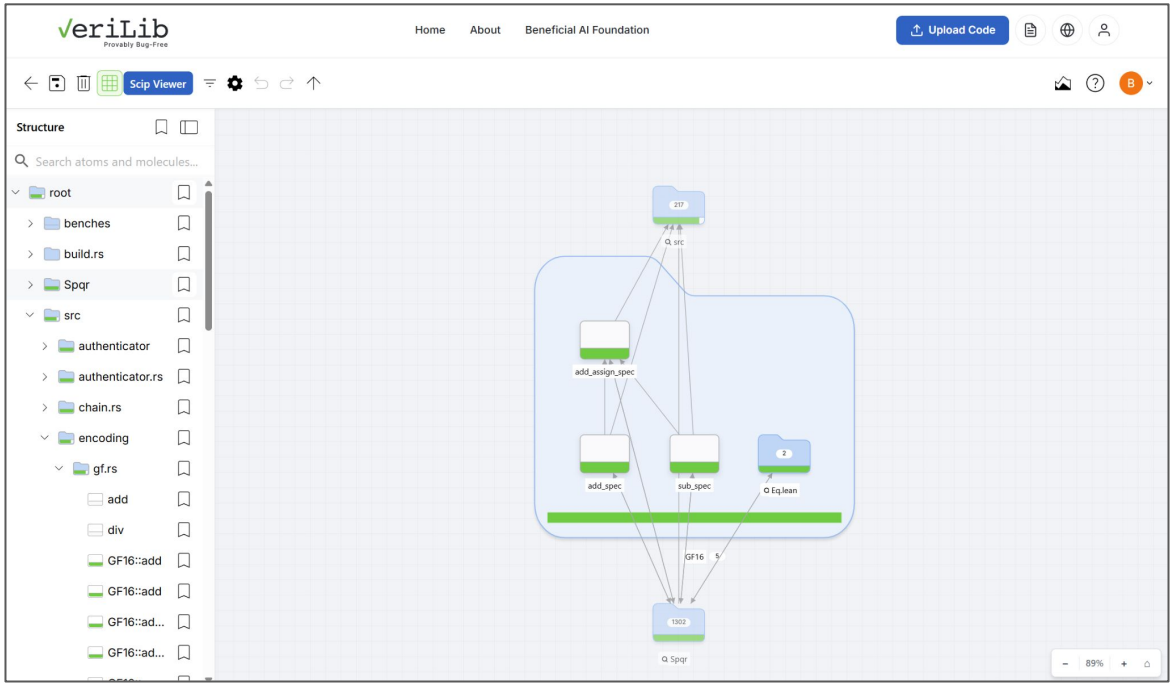
rust/core/src/address.rs/
- from
- kind
- raw_uuid
- service_id_binary
- service_id_fixed_width_binary



https://verilib.org/repobrowser?id=5172

VeriLib - SPQR (dependencies)

How do you display a verification project?



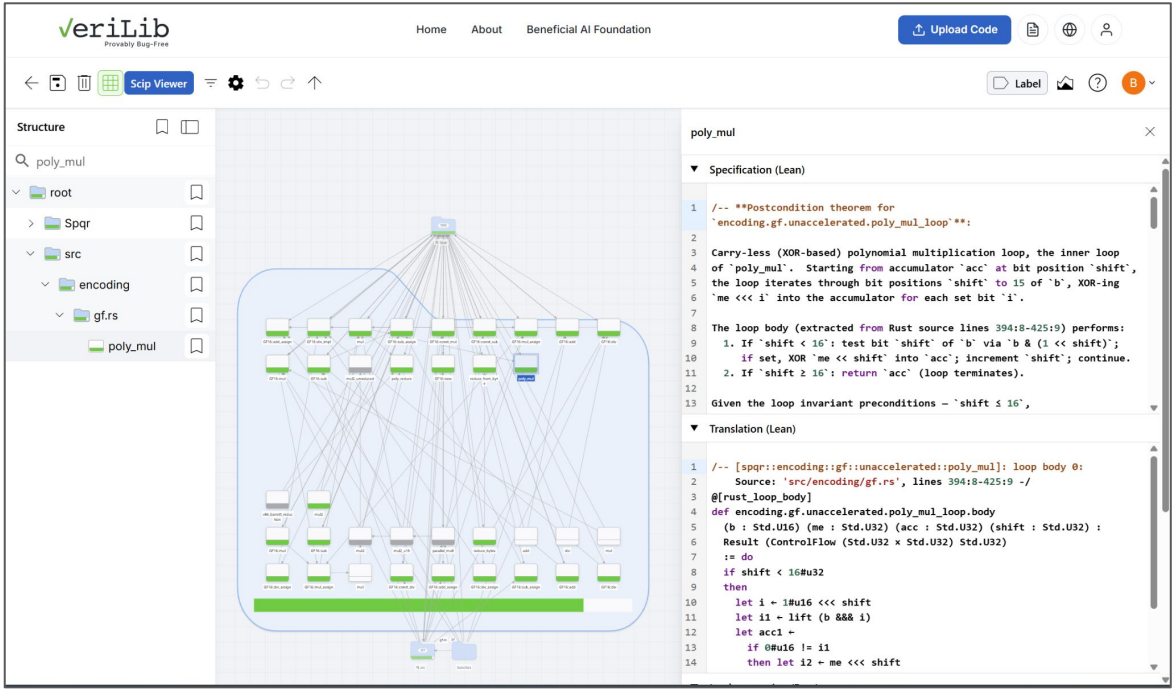
https://verilib.org/repobrowser?id=5173

VeriLib - SPQR (translations, specs, proofs)

How do you display a verification project?

Help us fill out specs and proofs!

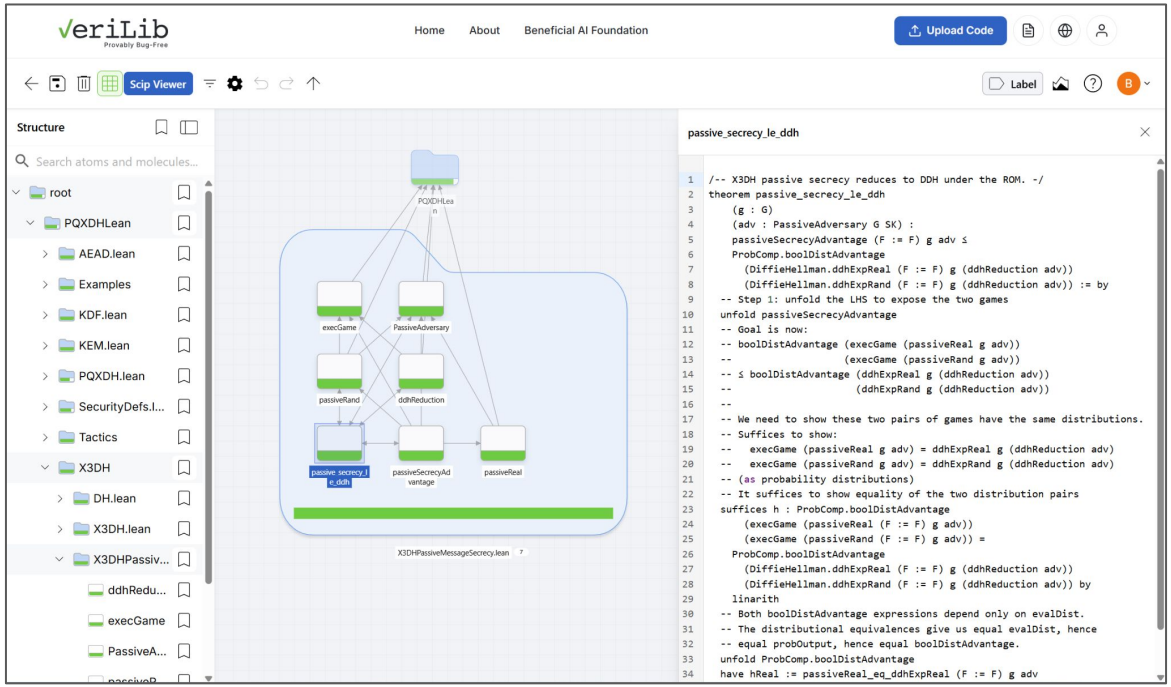
src/encoding/
  gf.rs/add
  gf.rs/sub
  gf.rs/poly_mul
  polynomial.rs/serialize
  polynomial.rs/deserialize



https://verilib.org/repobrowser?id=5173

VeriLib - PQXDH (protocol security)

How do you display a verification project?



https://verilib.org/repobrowser?id=5168

Collaborators

- Bas Spitters, Aarhus
- Son Ho, Aeneas, Microsoft
- Cas Cremers, CISPA
- Théophile Wallez, CISPA
- Quang Dao, CMU
- Karthik Bhargavan, Cryspen
- Clark Barrett, CSLib, Stanford
- Alexandre Rademaker, CSLib, Atlas
- Mike Dodds, Galois
- Samuel Schlesinger, Chrome, Google
- Yevgeniy Dodis, NYU
- Gopal Sarma, RAND
- Dawn Song, UC Berkeley
- Ilya Sergey, VERSE, NUS
- Linard Arquint, VERSE, NUS
- Vladimir Gladshtein, VERSE, NUS


Make it so!
beneficialaifoundation.org/signal-shot