



**BENEFICIAL
AI FOUNDATION**

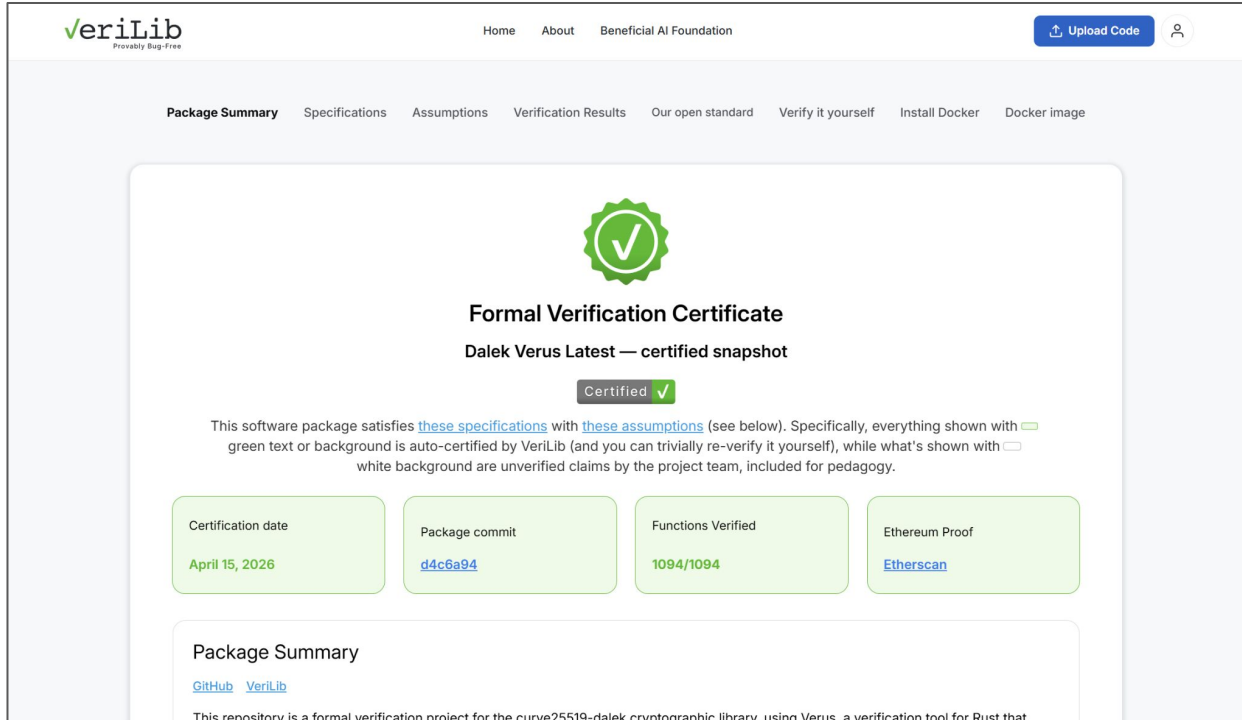
Open tools and standards for scaling software verification

Max Tegmark and BAIF

Paris, April 20, 2026



Trust Page - Dalek-Verus



The screenshot displays the VeriLib website interface. At the top left is the VeriLib logo with the tagline "Privately Bug-Free". The navigation bar includes links for Home, About, and Beneficial AI Foundation, along with an "Upload Code" button and a user profile icon. Below the navigation bar, a series of tabs are visible: Package Summary (selected), Specifications, Assumptions, Verification Results, Our open standard, Verify it yourself, Install Docker, and Docker image.

The main content area features a large green checkmark icon in a gear shape, followed by the heading "Formal Verification Certificate" and the sub-heading "Dalek Verus Latest — certified snapshot". A "Certified" badge with a green checkmark is positioned below the sub-heading.

The text below the badge states: "This software package satisfies [these specifications](#) with [these assumptions](#) (see below). Specifically, everything shown with green text or background is auto-certified by VeriLib (and you can trivially re-verify it yourself), while what's shown with white background are unverified claims by the project team, included for pedagogy."

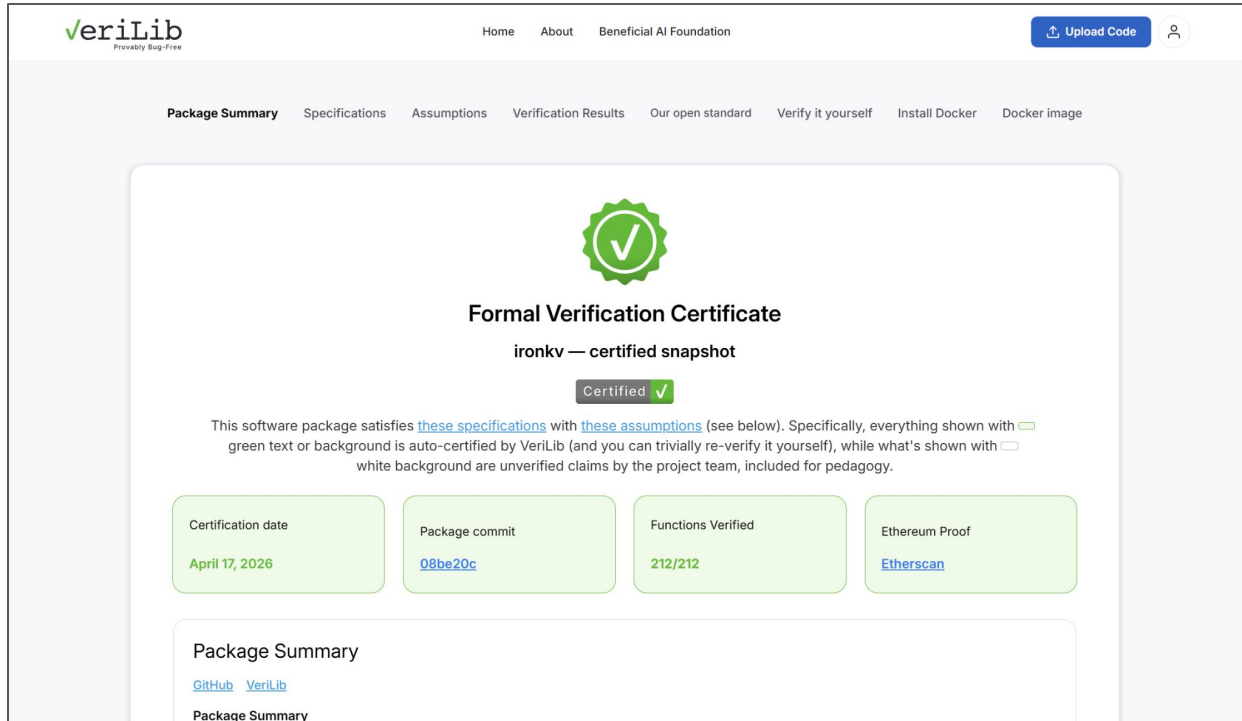
Four data points are presented in light green boxes:

- Certification date: April 15, 2026
- Package commit: [d4c6a94](#)
- Functions Verified: 1094/1094
- Ethereum Proof: [Etherscan](#)

Below this section is a "Package Summary" section with links to GitHub and VeriLib. The bottom of the page shows the beginning of a paragraph: "This repository is a formal verification project for the curve25519-dalek cryptographic library, using Verus, a verification tool for Rust that"

<https://verilib.org/cert/5132>

Trust Page - Verified-IronKV



verilib
Privately Sign-Free

Home About Beneficial AI Foundation [Upload Code](#)

Package Summary Specifications Assumptions Verification Results Our open standard Verify it yourself Install Docker Docker image

Formal Verification Certificate
ironkv — certified snapshot

Certified ✓

This software package satisfies [these specifications](#) with [these assumptions](#) (see below). Specifically, everything shown with green text or background is auto-certified by VeriLib (and you can trivially re-verify it yourself), while what's shown with white background are unverified claims by the project team, included for pedagogy.

Certification date	Package commit	Functions Verified	Ethereum Proof
April 17, 2026	08be20c	212/212	Etherscan

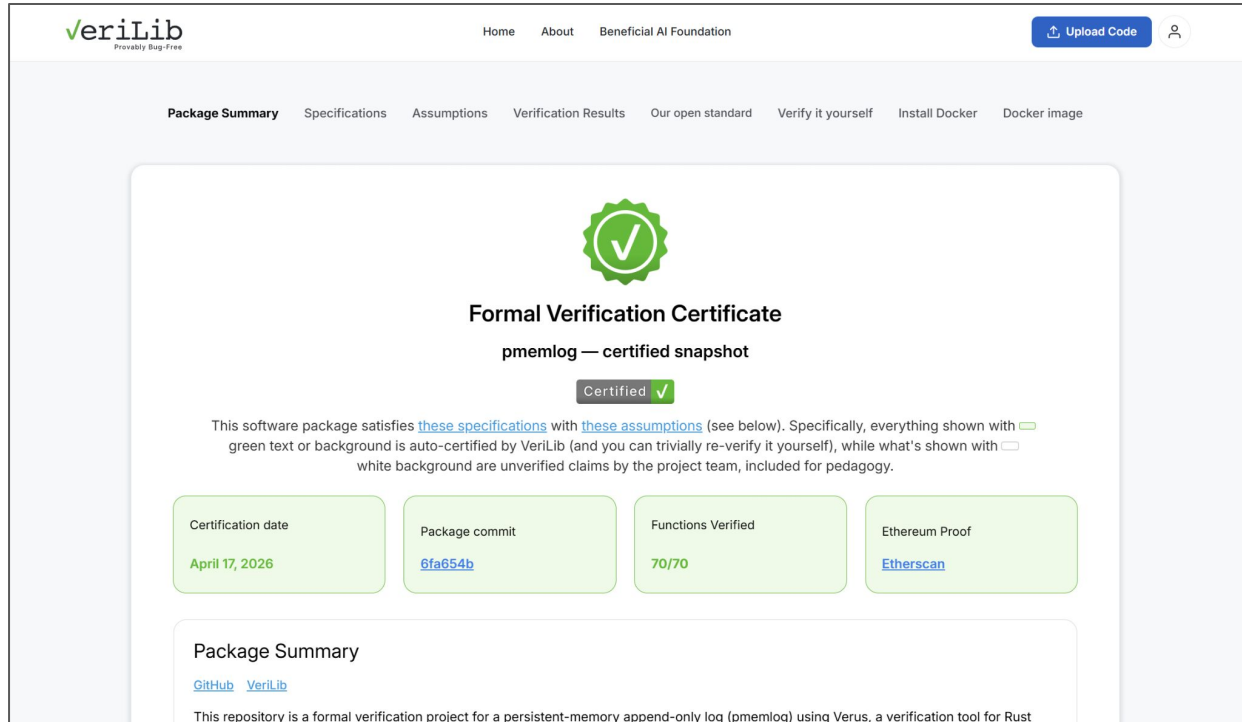
Package Summary

[GitHub](#) [VeriLib](#)

Package Summary

<https://verilib.org/cert/5167>

Trust Page - Microsoft-Verified-Storage-pmemlog



The screenshot displays the VeriLib website interface. At the top left is the VeriLib logo with the tagline "Provably Bug-Free". The navigation bar includes links for Home, About, and Beneficial AI Foundation, along with an "Upload Code" button and a user profile icon. Below the navigation bar, a series of tabs are visible: Package Summary (selected), Specifications, Assumptions, Verification Results, Our open standard, Verify it yourself, Install Docker, and Docker image.

The main content area features a large green checkmark icon inside a gear-like shape, signifying a formal verification certificate. Below this icon, the text reads "Formal Verification Certificate" and "pmemlog — certified snapshot". A "Certified" badge with a green checkmark is positioned below the title.

A paragraph of text explains the certification process: "This software package satisfies [these specifications](#) with [these assumptions](#) (see below). Specifically, everything shown with green text or background is auto-certified by VeriLib (and you can trivially re-verify it yourself), while what's shown with white background are unverified claims by the project team, included for pedagogy."

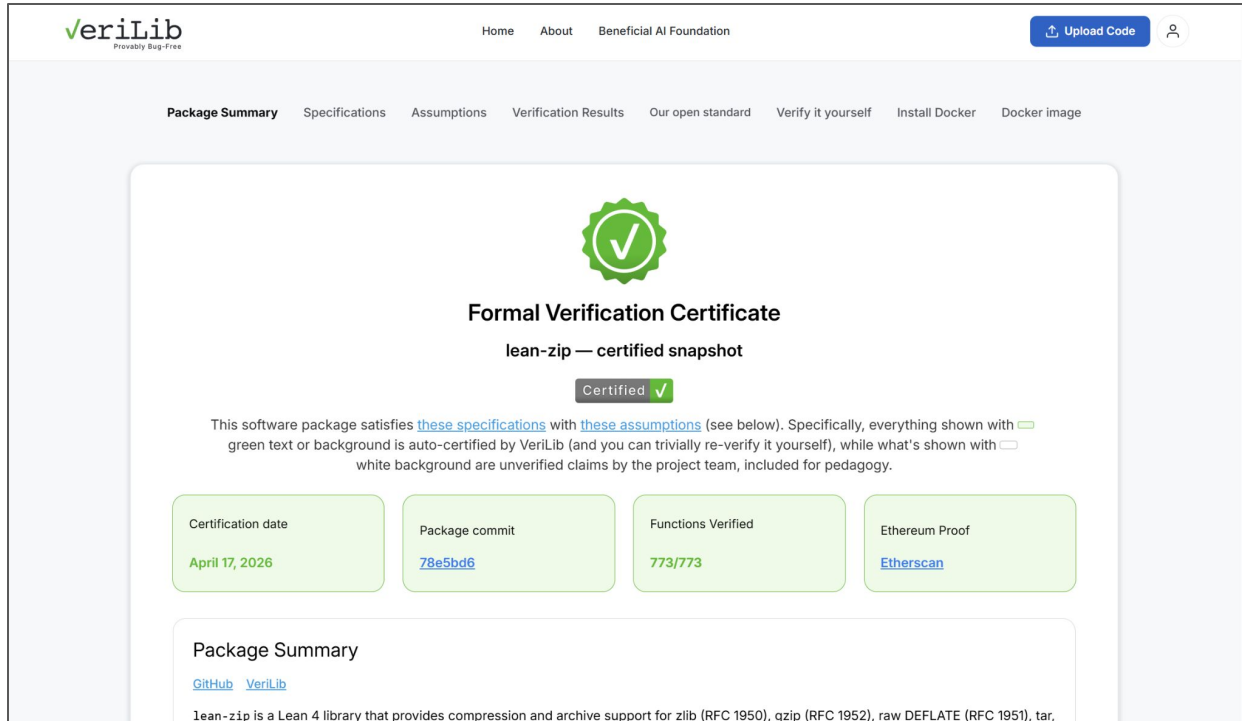
Four key metrics are displayed in light green boxes:

- Certification date: April 17, 2026
- Package commit: [6fa654b](#)
- Functions Verified: 70/70
- Ethereum Proof: [Etherscan](#)

Below these metrics is a "Package Summary" section with links to GitHub and VeriLib. At the bottom of the page, a note states: "This repository is a formal verification project for a persistent-memory append-only log (pmemlog) using Verus, a verification tool for Rust."

<https://verilib.org/cert/5171>

Trust Page - Lean-Zip




The screenshot displays the VeriLib website's trust page for the 'lean-zip' package. The page features a green checkmark icon and the text 'Formal Verification Certificate' and 'lean-zip — certified snapshot'. A 'Certified' badge with a checkmark is visible. Below this, a paragraph explains that the software package satisfies specifications and assumptions, with green text indicating auto-certified content and white text indicating unverified claims. A table of verification details is provided, including the certification date (April 17, 2026), package commit (78e5bd6), functions verified (773/773), and an Ethereum proof link (Etherscan). The page also includes a 'Package Summary' section with links to GitHub and VeriLib, and a brief description of the lean-zip library.

verilib
Privacy Bug-Free


Home About Beneficial AI Foundation [Upload Code](#)



Package Summary Specifications Assumptions Verification Results Our open standard Verify it yourself Install Docker Docker image



Formal Verification Certificate

lean-zip — certified snapshot

Certified 

This software package satisfies [these specifications](#) with [these assumptions](#) (see below). Specifically, everything shown with  green text or background is auto-certified by VeriLib (and you can trivially re-verify it yourself), while what's shown with  white background are unverified claims by the project team, included for pedagogy.

Certification date	Package commit	Functions Verified	Ethereum Proof
April 17, 2026	78e5bd6	773/773	Etherscan

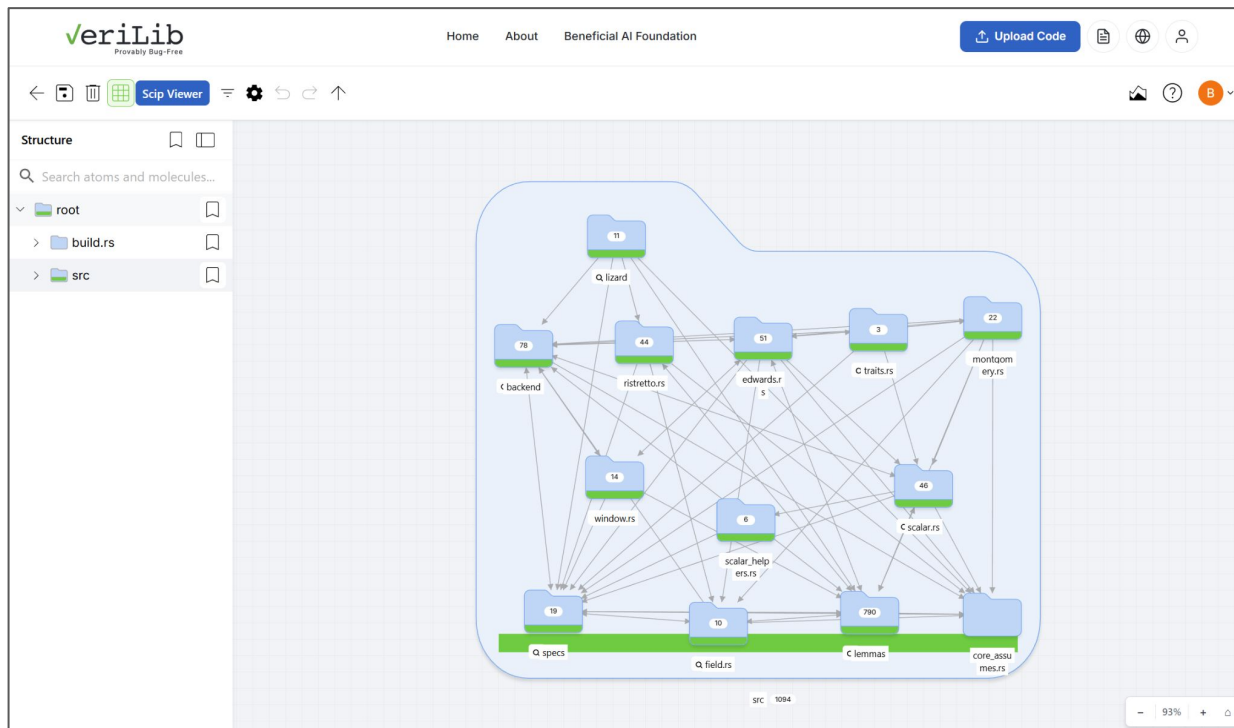
Package Summary

[GitHub](#) [VeriLib](#)

lean-zip is a Lean 4 library that provides compression and archive support for zlib (RFC 1950), gzip (RFC 1952), raw DEFLATE (RFC 1951), tar,

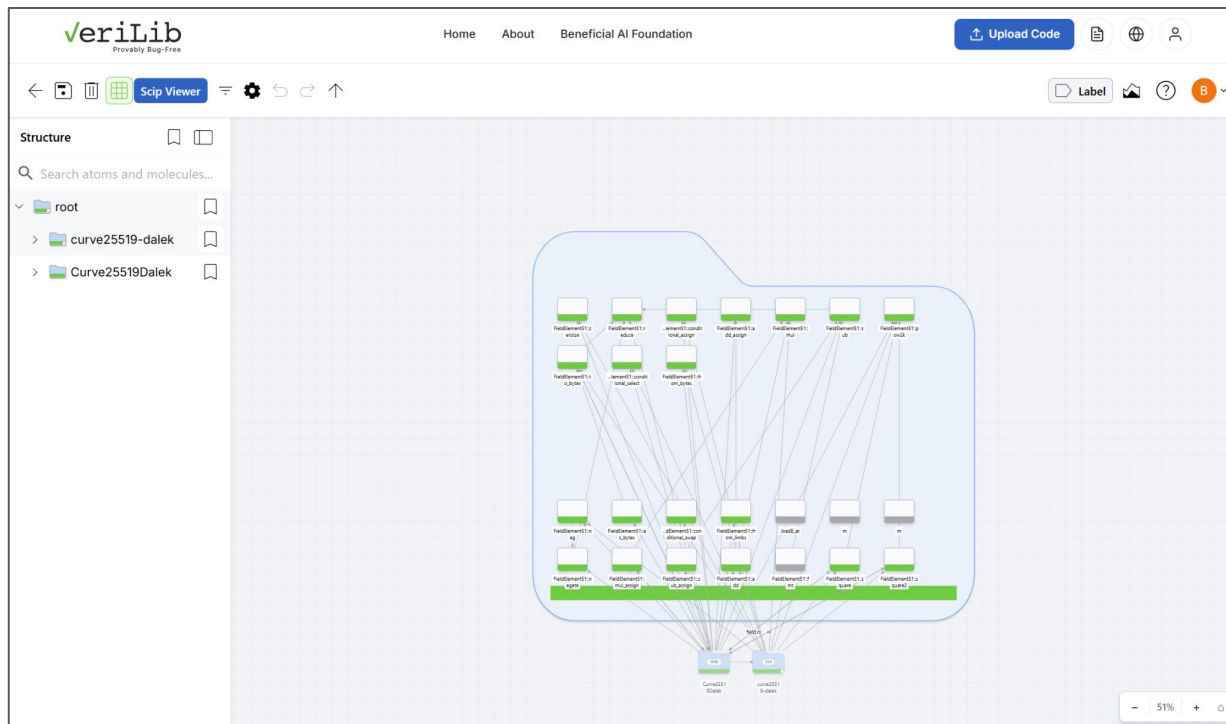
<https://verilib.org/cert/5165>

Dalek-Verus



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

Dalek-Learn



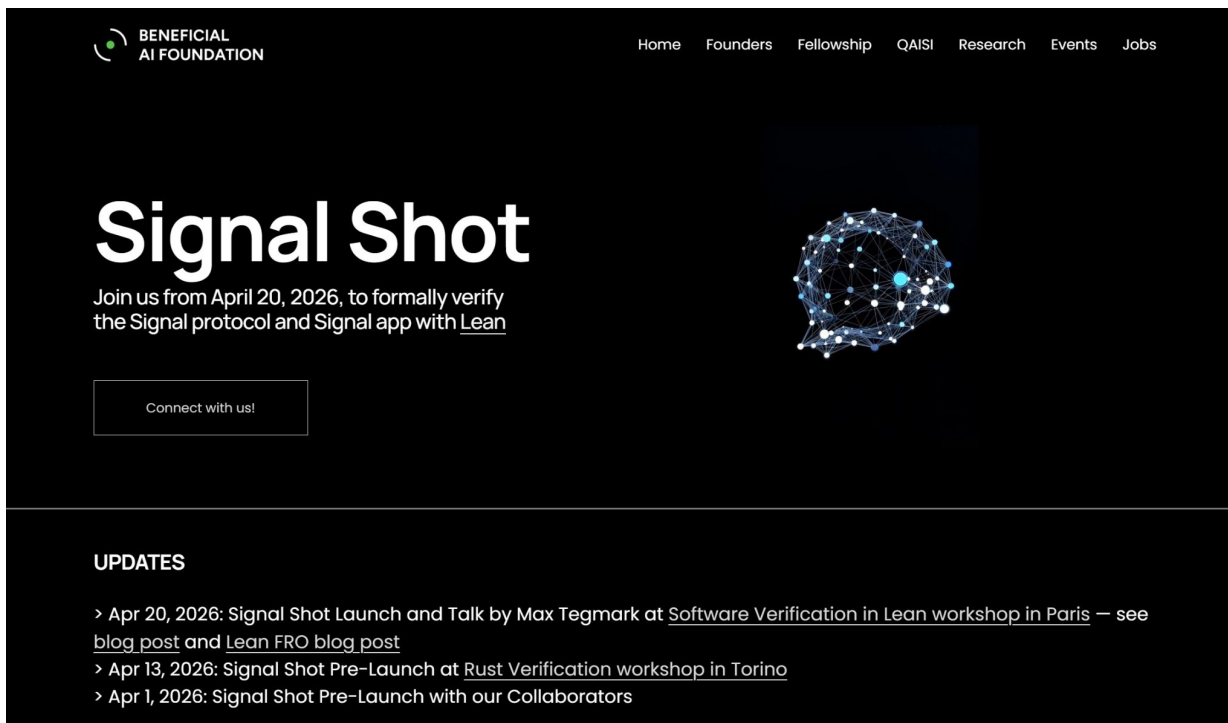
<https://verilib.org/repobrowser?id=5174>

A composite image of a solar system. In the center, a yellow sun is surrounded by several planets on elliptical orbits. One planet is orange, another is white, and another is yellow. In the foreground, a satellite with a large blue dish antenna and a yellow body is shown. To the right, the planet Saturn is visible with its rings. The background is a dark space filled with stars.

Signal Shot

How We Are Organized

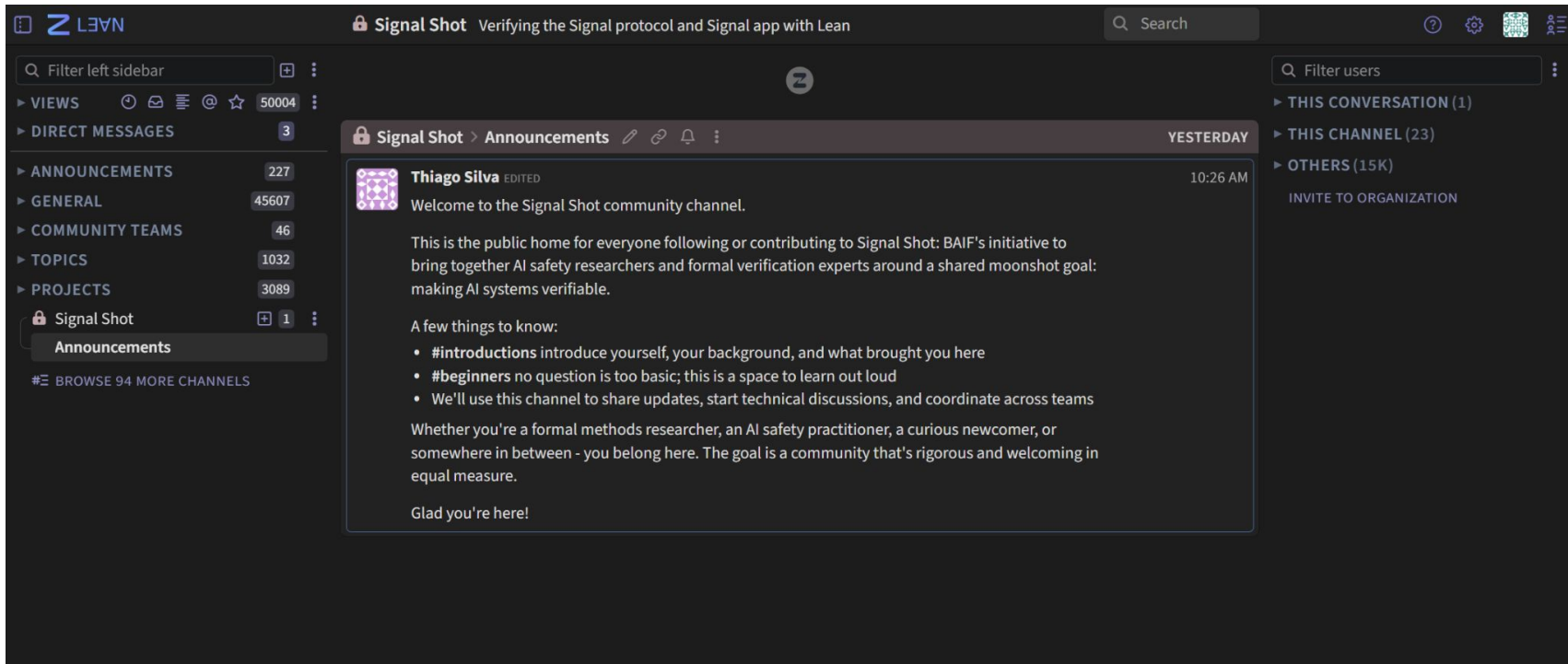
Signal Shot - Main Site



The screenshot shows the main landing page for Signal Shot. At the top left is the logo for the Beneficial AI Foundation, which consists of a stylized 'B' icon and the text 'BENEFICIAL AI FOUNDATION'. To the right of the logo is a navigation menu with links for 'Home', 'Founders', 'Fellowship', 'QAISI', 'Research', 'Events', and 'Jobs'. The main content area features the title 'Signal Shot' in a large, white, sans-serif font. Below the title is a sub-headline: 'Join us from April 20, 2026, to formally verify the Signal protocol and Signal app with [Lean](#)'. To the right of the text is a graphic of a brain composed of a network of white and blue nodes connected by thin lines. Below the sub-headline is a rectangular button with the text 'Connect with us!'. At the bottom of the page, there is a section titled 'UPDATES' in white, uppercase letters. This section contains three bullet points, each starting with a right-pointing chevron: '> Apr 20, 2026: Signal Shot Launch and Talk by Max Tegmark at [Software Verification in Lean workshop in Paris](#) — see [blog post](#) and [Lean FRO blog post](#)', '> Apr 13, 2026: Signal Shot Pre-Launch at [Rust Verification workshop in Torino](#)', and '> Apr 1, 2026: Signal Shot Pre-Launch with our Collaborators'.

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

Signal Shot - Lean Prover Zulip Chat



The screenshot shows the Zulip chat interface for the Signal Shot community channel. The channel is titled "Signal Shot" and is described as "Verifying the Signal protocol and Signal app with Lean". The channel is locked and has a lock icon. The channel name is "Signal Shot" and the channel type is "Announcements". The channel is located in the "Signal Shot" organization. The channel has 1 member and is currently active. The channel is part of the "Signal Shot" organization, which has 15K other members. The channel is currently active and has 1 member. The channel is part of the "Signal Shot" organization, which has 15K other members. The channel is currently active and has 1 member. The channel is part of the "Signal Shot" organization, which has 15K other members.

Thiago Silva EDITED 10:26 AM
Welcome to the Signal Shot community channel.

This is the public home for everyone following or contributing to Signal Shot: BAIF's initiative to bring together AI safety researchers and formal verification experts around a shared moonshot goal: making AI systems verifiable.

A few things to know:

- **#introductions** introduce yourself, your background, and what brought you here
- **#beginners** no question is too basic; this is a space to learn out loud
- We'll use this channel to share updates, start technical discussions, and coordinate across teams

Whether you're a formal methods researcher, an AI safety practitioner, a curious newcomer, or somewhere in between - you belong here. The goal is a community that's rigorous and welcoming in equal measure.

Glad you're here!

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

Verifying Protocols

Christiano Braga

SECURITY PROTOCOLS

All protocols 538

PQXDH

Hybrid Key Exchange

110

Double / Triple Ratchet

Message Encryption

331

SPQR

Post-Quantum Ratchet

190

Group (SenderKey)

Group Messaging

71

Sealed Sender

Anonymous Delivery

63

Key Transparency

Transparency Verification

75

CRYPTOGRAPHIC PRIMITIVES

All primitives

KEM

3 protocols

DH

4 protocols

AEAD

2 protocols

KDF

5 protocols

Hash

6 protocols

MAC / PRF

4 protocols

CKA / SCKA

2 protocols

Symmetric Enc

3 protocols

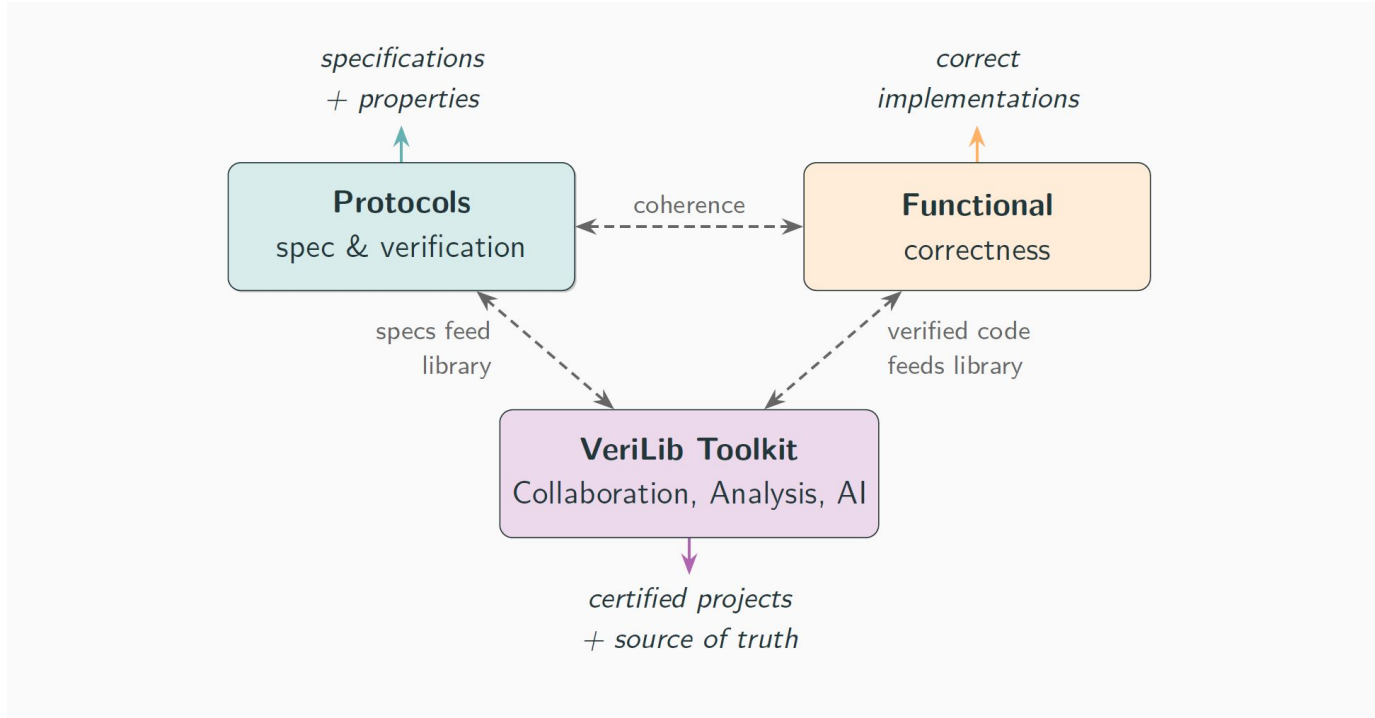
Signature

4 protocols

VRF

1 protocols

Protocol Specification and Verification



https://drive.google.com/file/d/1_IbCIN-euSruTCb29j99plgV1FYrYwZ8

Verifying Functions

Jure Kukovec

IMPLEMENTATION CRATES

All crates

libsignal-core

31 fns

libsignal-keytrans

75 fns

libsignal-protocol

241 fns

signal-crypto

2 fns

spqr

189 fns

EXTERNAL CRYPTO CRATES

aes

aes-gcm-siv

cbc

ctr

curve25519-dalek

ed25519-dalek

hkdf

hmac

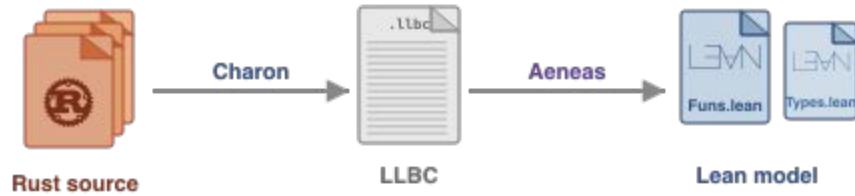
libcrux-hmac

libcrux-ml-kem

sha2

x25519-dalek

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)

```

14
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

```

> Field > FieldElement51 > ≡ Reduce.lean > {} curve25519_dalek.backend.serial.u64.field.FieldElement51
31 namespace curve25519_dalek.backend.serial.u64.field.
40
47
48 theorem reduce_spec (limbs : Array U64 5#usize) :
49   reduce limbs { (result : FieldElement51) =>
50     (∀ i < 5, result[i]!.val < 2 ^ 52) ∧
51     Field51_as_Nat limbs ≡ Field51_as_Nat result [MOD p] ∧
52     Field51_as_Nat result < 2 * p } := by
53   unfold reduce
54   step* ✖
55
56
57
58
59 end curve25519_dalek.backend.serial.u64.field.FieldElement51
60

```

▼ Reduce.lean:56:2

▼ Tactic state

7 goals

▼ case hmax

```

└─ r4 * r19#u64 ≤ U64.max
limbs5_post : limbs5 = limbs4
_ : [> let limbs5 ← limbs4
limbs5 : Std.Array U64 5#u
i14_post2 : i14.bv = i13.t
i14_post1 : r14 = r13 &
_ † : [> let i14 ← lift (i
i14 : U64
i13_post : i13 = (r14)
_ †² : [> let i13 ← limbs4
i13 : U64
limbs4_post : limbs4 = limbs3
_ †² : [> let limbs4 ← limbs3
limbs4 : Std.Array U64 5#u
i12_post2 : i12.bv = i11.t
i12_post1 : r12 = r11 &
_ †³ : [> let i12 ← lift (i
i12 : U64
i11_post : i11 = (r12)
_ †⁴ : [> let i11 ← limbs3
i11 : U64
limbs3_post : limbs3 = limbs2
_ †⁵ : [> let limbs3 ← limbs2
limbs3 : Std.Array U64 5#u

```

- 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

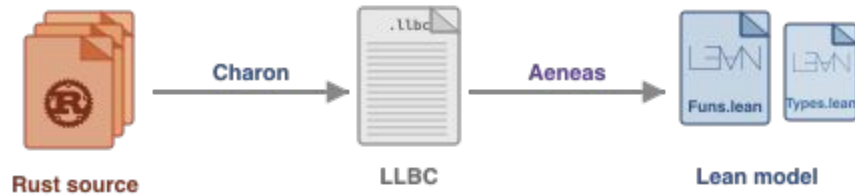
```
theorem reduce_spec (limbs : Array U64 5#usize) :
  reduce limbs { (result : FieldElement51) =>
    (∀ i < 5, result[i]!.val < 2 ^ 52) ∧
    Field51_as_Nat limbs ≡ Field51_as_Nat result [MOD p] ∧
    Field51_as_Nat result < 2 * p } := by
  unfold reduce
  step*
  · scalar_tac
  · simp [*]; scalar_tac
  · 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 !

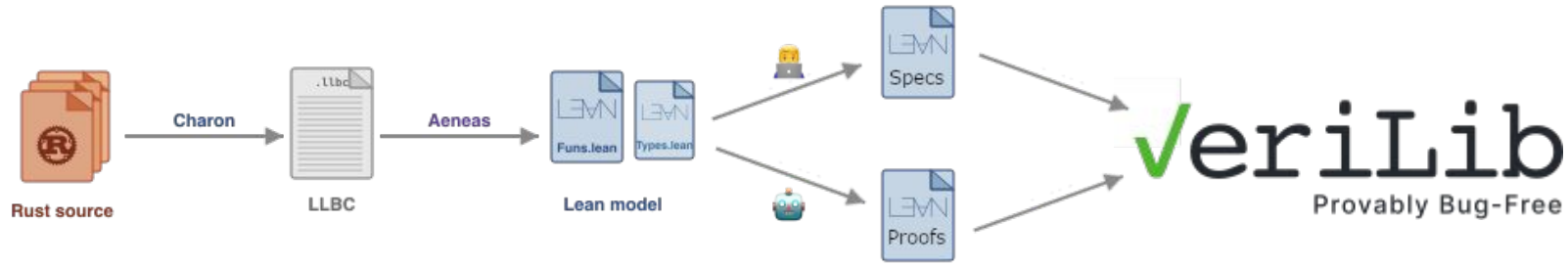
We want to go from this ...

Lean extraction pipeline

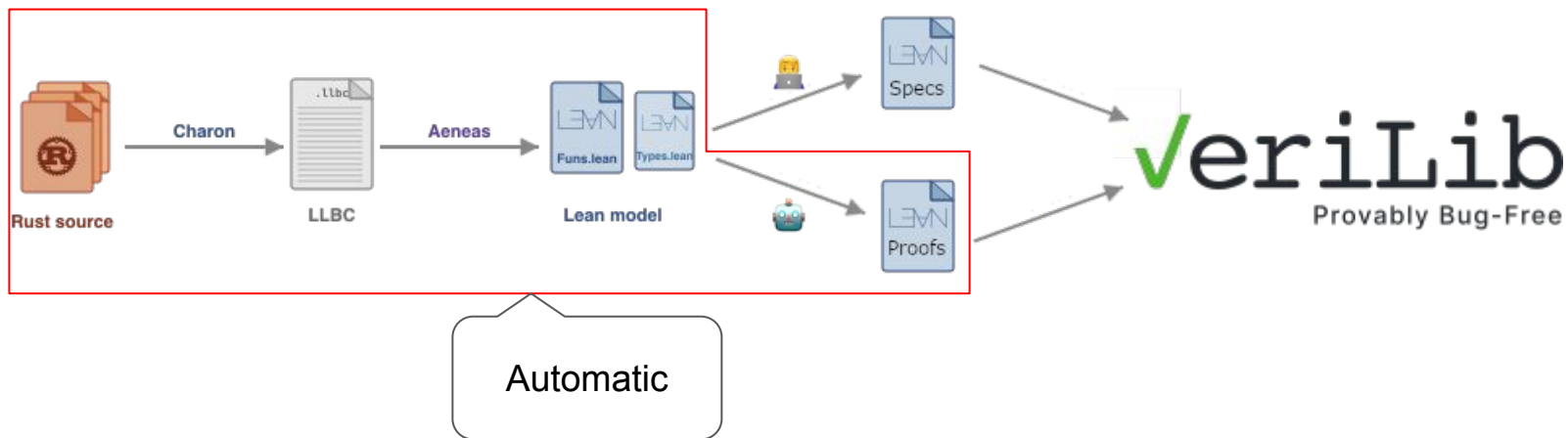


... to this

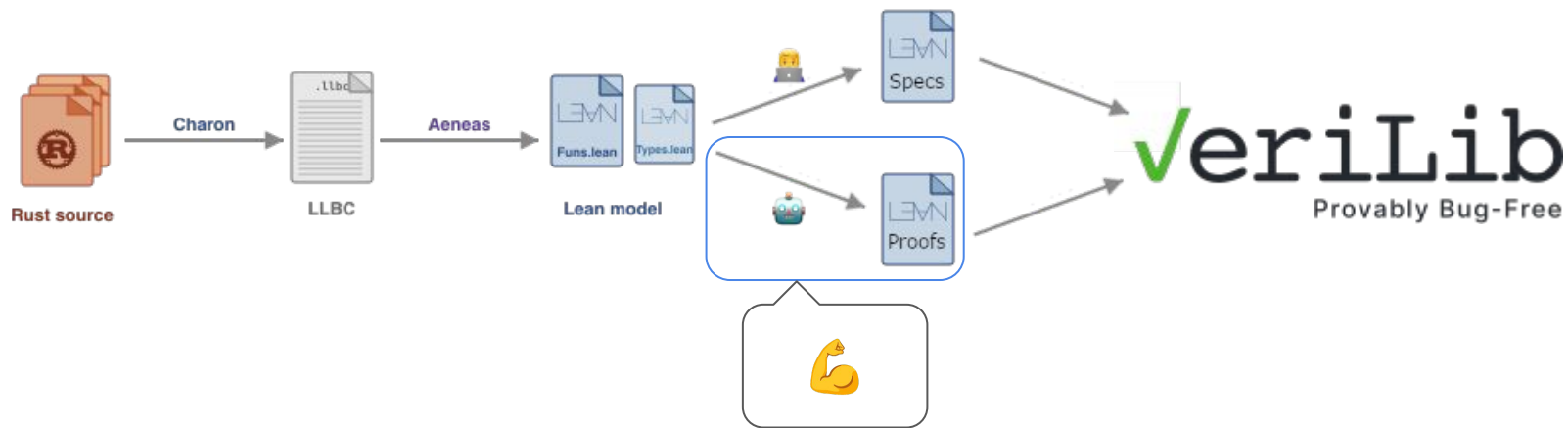
Full verification pipeline



Full verification pipeline



Full verification pipeline



VeriLib Toolkit

Lacramioara Astefanoaei

Collab

Blueprint
Issues

Analysis

Probes
Call Graphs

AI

Autoformalize
Autoverify

Trust

Reproducible
Composable

Verso-Blueprint

How do you plan a verification project?

PQXDH in Lean

Search...

◀ 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 _ => $! 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

► Table of Contents

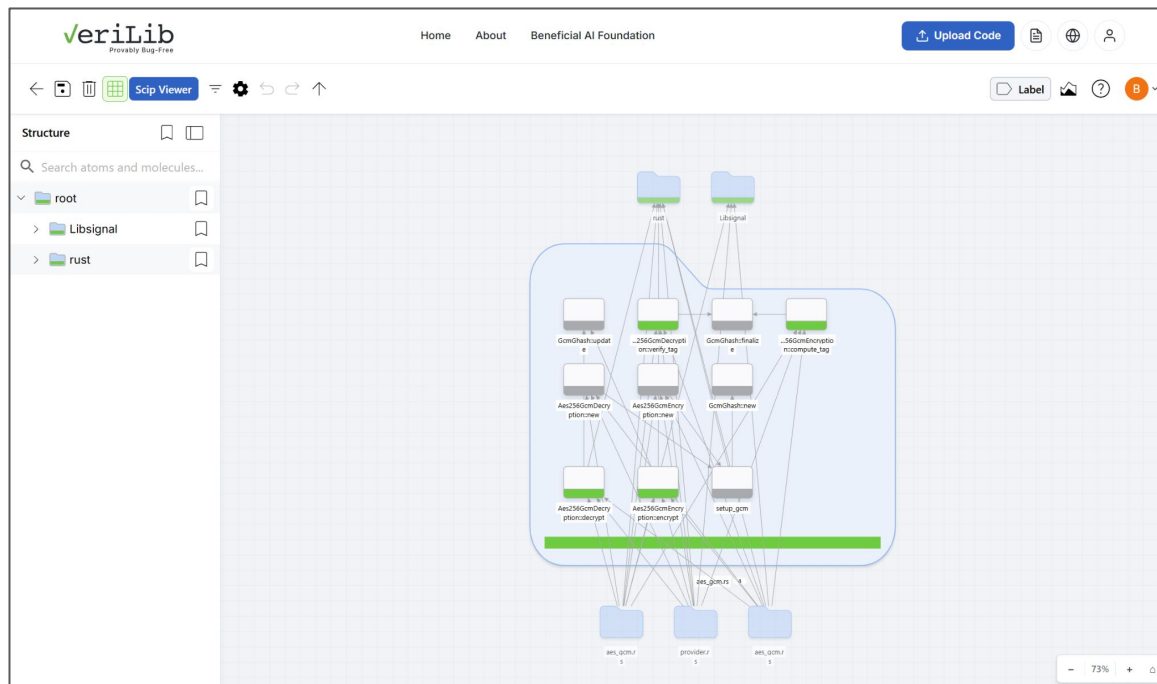
- ▼ 5. The X3DH Protocol
 - 5.1. Key pairs
 - 5.2. Shared secret computation
 - 5.3. Correctness
 - 5.4. Passive Message Secrecy
- ▼ 5.4. Passive Message Secrecy
 - 5.4.1. Random Oracle Model
 - 5.4.2. Passive adversary
 - 5.4.3. Two-game formulation
 - 5.4.4. Advantage
 - 5.4.5. DDH reduction
 - 5.4.6. Distributional equivalences
 - 5.4.7. Security theorem

Special thanks to
Emilio Jesús Gallego Arias!

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

VeriLib - libsignal (dependencies)

How do you display a verification project?

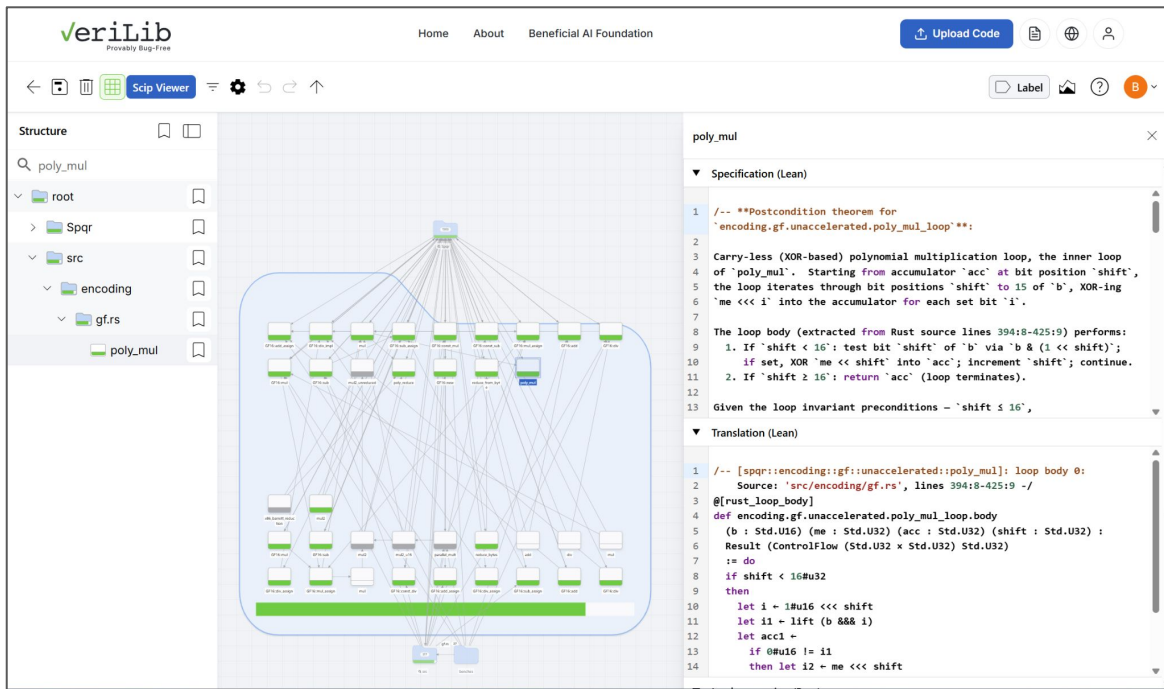


<https://verilib.org/repobrowser?id=5172>

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



The screenshot displays the VeriLib web interface. At the top, the logo 'veriLib' is visible with the tagline 'Privately Bug-Free'. Navigation links for 'Home', 'About', and 'Beneficial AI Foundation' are present, along with an 'Upload Code' button. The interface is divided into three main sections:

- Structure:** A sidebar on the left shows a tree view of the project files: root, Spqr, src, encoding, gf.rs, and poly_mul.
- Graph:** The central area shows a complex dependency graph with numerous nodes and edges, representing the relationships between different parts of the code.
- Code Editor:** The right side shows the code for 'poly_mul'. It includes a Lean specification and its translation to Rust. The specification describes a carry-less (XOR-based) polynomial multiplication loop. The translation shows the corresponding Rust code with comments explaining the loop's operation.

<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)

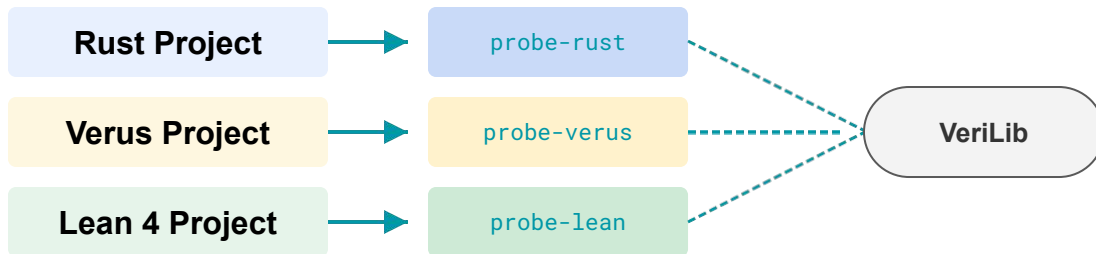
The Interoperability Challenge

- **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

The screenshot displays the SCIP Call Graph Viewer interface. At the top, there's a header with the title "SCIP Call Graph Viewer" and navigation tabs for "Call Graph", "File Map", and "Namespace Map". A "Load Graph JSON" button is on the right. Below the header, a search bar contains the query "all (no traversal)".

The left sidebar contains several filter sections:

- Filters:**
 - Source Type: Libsignal, External
 - Declaration Kind: Definitions (def, abbrev, class, ...), Theorems, Axioms
 - Verification Status: Verified, Failed, Unverified / Unknown
 - Exclude by Name: Input field with "e.g., *_comm*, lemma_mul_*". A note says "Matches function names. *_comm* excludes lemma_commutativity".
 - Exclude by Path: "+ Add preset..." dropdown. Input field with "e.g., */specs/*, */common_lemmas/*". A note says "Matches node paths. */specs/* excludes all spec functions".
 - Include Files: (empty)

The main area shows a call graph with nodes and edges. A "Namespace Map" tooltip is visible, explaining that blue boxes represent namespaces, lines represent cross-namespace calls, and line width indicates call count. It also lists interactions: clicking an edge expands functions, clicking a namespace shows info, and double-clicking a namespace opens its call graph.

At the bottom, there are instructions: "Zoom: Scroll or pinch", "Pan: Click and drag background", "Move Node: Drag a node", "Select: Click a node to see details", "Hide: Shift+click a node", and "Source/Sink: Enter function names to explore call paths".

The footer indicates "Built with D3.js | GitHub".

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

GitHub Issues

How do you coordinate a verification project?

Beneficial-AI-Foundation / curve25519-dalek-lean-verify

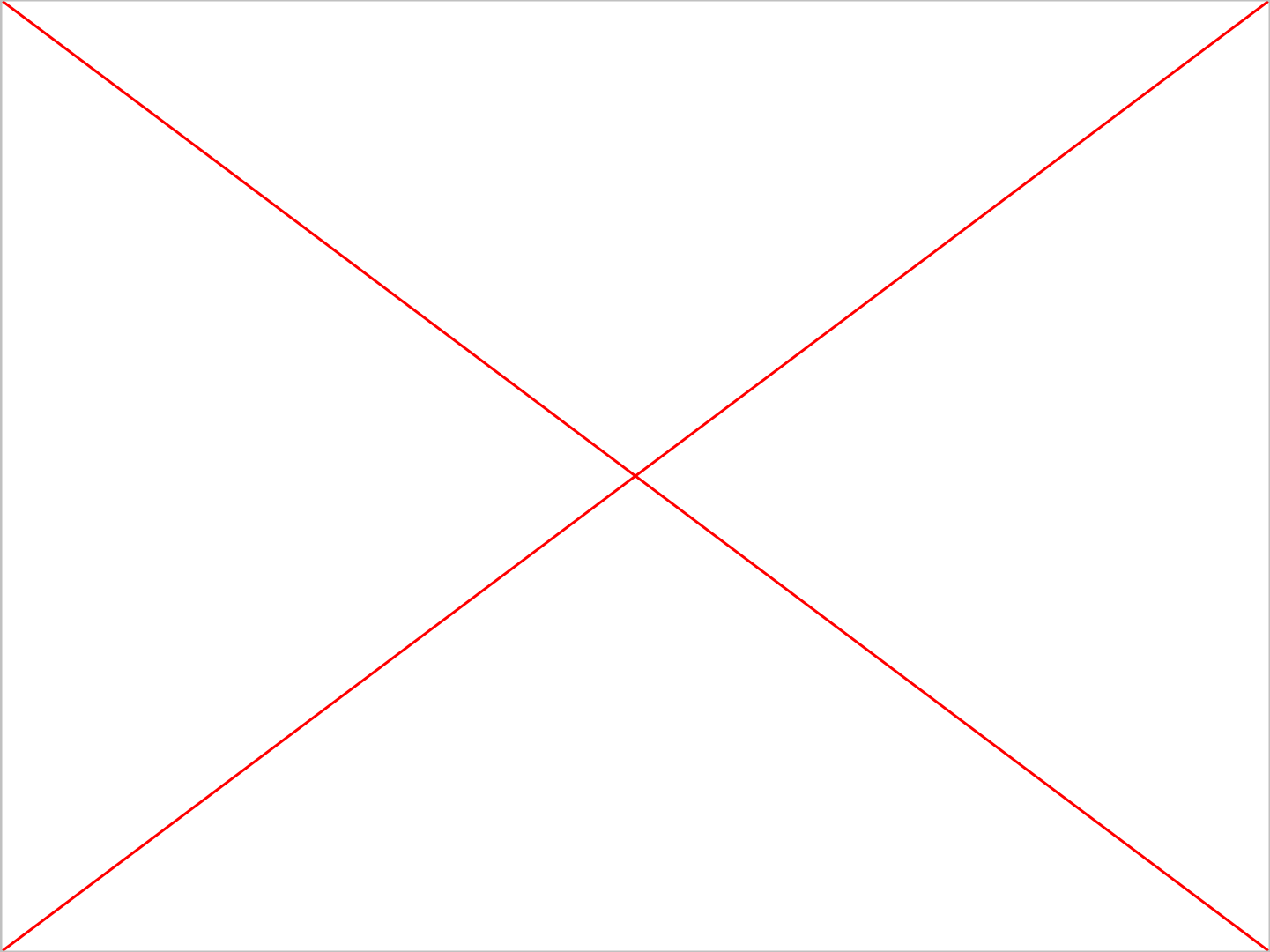
Code Issues 24 Pull requests 4 Agents Discussions Actions Projects Models Security and quality Insights Settings

is:issue state:closed

Open 24 Closed 427

Author	Labels	Projects	Milestones	Assignees	Types	Newest
MarkusFerdinandDablander					1	
MarkusFerdinandDablander					1	
truonghoangle	chore				1	
oliver-butterley						
truonghoangle	verify				1	
truonghoangle	specify				1	
truonghoangle	verify				1	
truonghoangle	specify				1	

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



VeriLib Toolkit

Alessandro D'Angelo

Collab

Blueprint
Issues

Analysis

Dependencies

AI

Autoformalize
Autoverify

Trust

Reproducible
Composable

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}	<code>sqrt_ratio_i_spec</code> needs $\leq 2^{52}-1$	✗	

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 $\leq 2^{52} - 1$ but its internal operations only need $< 2^{54}$.

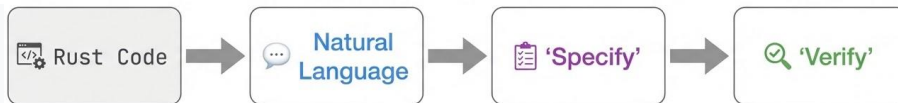
Recommended pre-requisite fix

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
1 <purpose>
2 Orchestrate specification generation for a single Lean function using two-phase
3 subagent dispatch (research -> execute).
4
5 Takes a verification target (function name), dispatches fvs-researcher to gather
6 context (Funs.lean, Types.lean, Rust source, existing stubs, similar specs), then
7 dispatches fvs-executor to write the spec file.
8
9 Output: Specs/{path}/{FunctionName}.lean with @[step] theorem and sorry placeholder.
10 </purpose>
```

Skill Specification & Generation

```
.claude > fv-skills > workflows > ↓ lean-verify.md
1 <purpose>
2 Orchestrate interactive proof development for a Lean specification using two-phase
3 subagent dispatch (research -> iterative execute).
4
5 Dispatches fvs-researcher to analyze sorry locations and recommend proof strategies,
6 then iteratively dispatches fvs-executor to replace each sorry ONE AT A TIME with
7 small tactic blocks. The user checks Lean compiles between each step.
8
9 This is the most interactive workflow -- it feels like pair programming. Small changes,
10 user approval, Lean compile check, repeat.
11
12 Output: Spec file with sorry replaced by complete proof, or clear report of stuck goals.
13 </purpose>
```

Interactive Verification & Analysis

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

Next Steps

Shaowei Lin

Libsignal Protocols Code Map

Security Protocols

All protocols 538

PQXDH Hybrid Key Exchange 110	Double / Triple Ratchet Message Encryption 331	SPQR Post-Quantum Ratchet 190
Group (SenderKey) Group Messaging 71	Sealed Sender Anonymous Delivery 63	Key Transparency Transparency Verification 75

Cryptographic Primitives

All primitives

KEM 3 protocols 3	DH 4 protocols 4	AEAD 2 protocols 2
KDF 5 protocols 5	Hash 6 protocols 6	MAC / PRF 4 protocols 4
CKA / SCKA 2 protocols 2	Symmetric Enc 3 protocols 3	Signature 4 protocols 4

Search functions...

Function Browser

Showing 580 of 580 rows across 17 crates

- ▶ libsignal-core
- ▶ libsignal-keytrans
- ▶ libsignal-protocol
- ▶ signal-crypto
- ▶ spqr
- ▶ External Crypto Crates

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

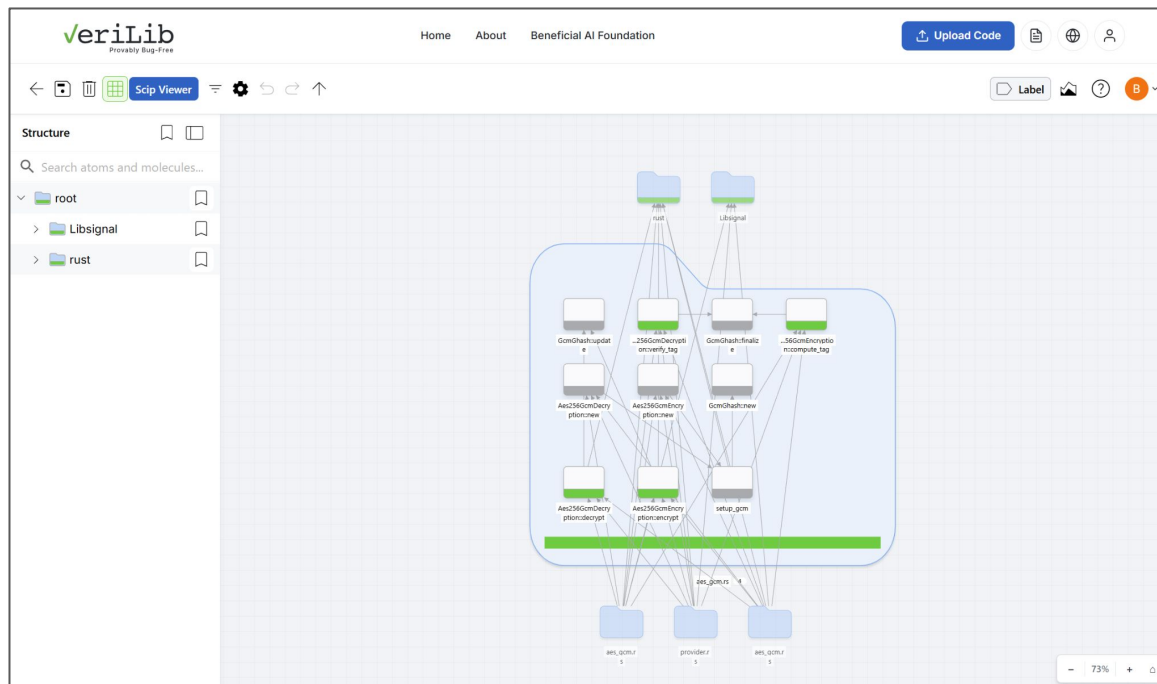
Stage 2 - Bridging Protocols and Functions

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

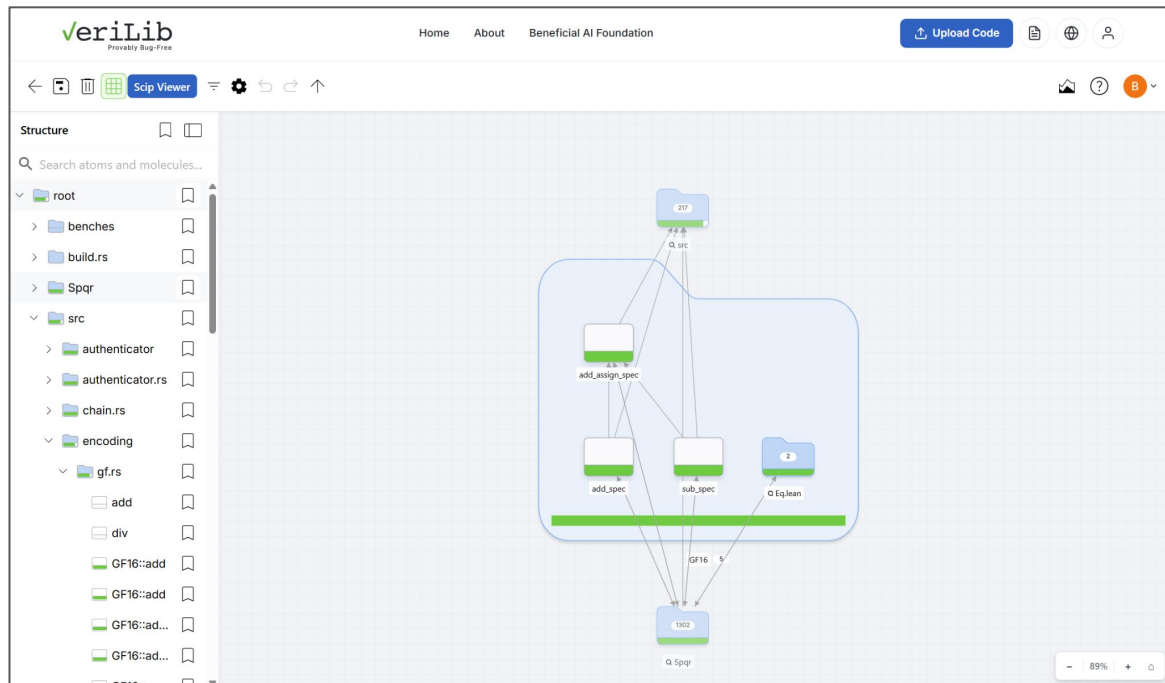
The screenshot displays the VeriLib web interface. The top navigation bar includes 'Home', 'About', and 'Beneficial AI Foundation', along with an 'Upload Code' button and user icons. The left sidebar shows a project structure with folders for 'root', 'Libsignal', 'rust', 'crypto', 'benches', 'src', and 'tests'. The main area features a 'Scip Viewer' and a diagram of the project's structure, with a blue box highlighting a specific section. The right panel shows a theorem proof in a code editor:

```
from_spec
1 @[step]
2 theorem from_spec (value : libsignal_core.address.ServiceIdKind) :
3   «from» value { result =>
4     (value = .Aci → result = 0#u8) ∧
5     (value = .Pni → result = 1#u8) ] := by
6   unfold «from»
7   step*
8   cases value <;> simp_all <;> native_decide
9
```

<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

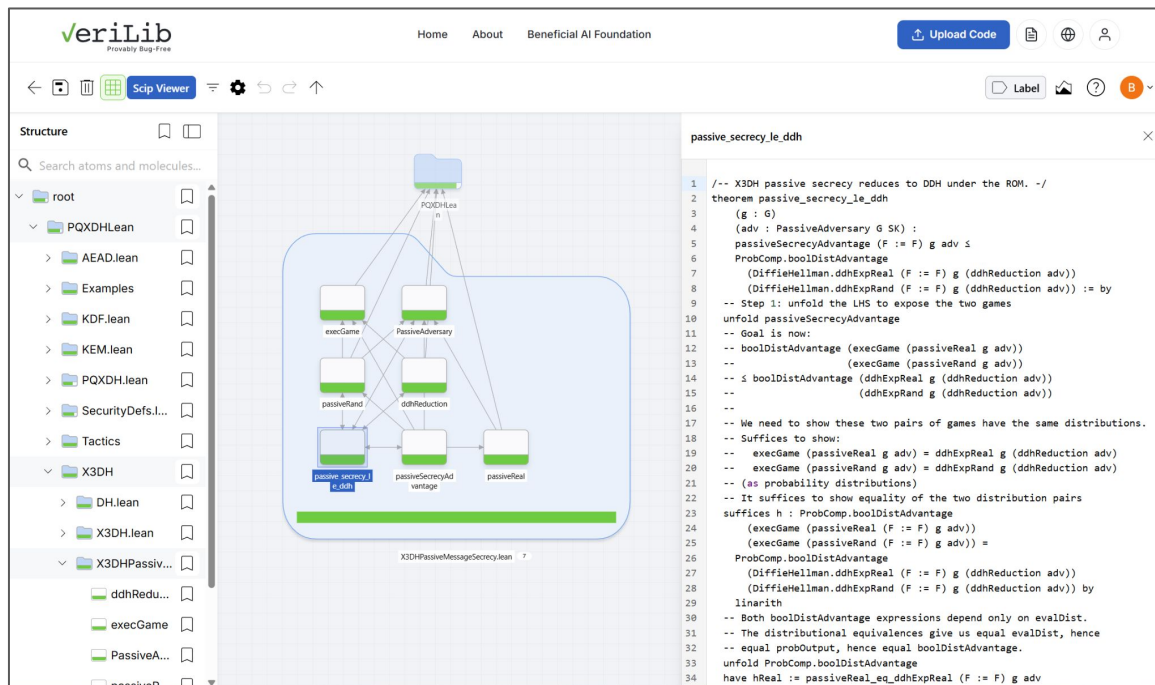
The screenshot shows the VeriLib web interface. At the top, there's a navigation bar with 'Home', 'About', and 'Beneficial AI Foundation'. A 'Upload Code' button is on the right. Below the navigation bar, there's a 'Scip Viewer' toolbar. The main content area is divided into three panels:

- Structure:** A file tree on the left showing the project structure: root > Spqr > src > encoding > gf.rs > poly_mul.
- Graph:** A central dependency graph showing nodes for various files and their relationships. A blue box highlights a specific part of the graph.
- Code Editor:** On the right, the code for 'poly_mul' is displayed. It includes a Lean specification and a Rust translation. The specification describes a carry-less (XOR-based) polynomial multiplication loop. The translation shows the corresponding Rust code with comments.

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

VeriLib - PQXDH (protocol security)

How do you display a verification project?



The screenshot displays the VeriLib web interface. On the left, a 'Structure' sidebar shows a tree view of the project files, including folders for 'PQXDHLean', 'AEAD.lean', 'Examples', 'KDF.lean', 'KEM.lean', 'PQXDH.lean', 'SecurityDefs.l...', 'Tactics', and 'X3DH'. Under 'X3DH', there are sub-folders for 'DH.lean', 'X3DH.lean', and 'X3DHPassiv...'. The main area shows a dependency graph for the 'X3DHPassiveMessageSecurity.lean' file, with nodes for 'execGame', 'PassiveAdversary', 'passiveRand', 'ddhReduction', 'passiveSecretyAdvantage', and 'passiveReal'. A blue box highlights the 'passiveSecretyAdvantage' node and its dependencies. On the right, a code editor shows the 'passive_securety_le_ddh' theorem, which is a proof that X3DH passive security reduces to DDH under the ROM. The code includes imports, type signatures, and a proof strategy using 'unfold' and 'suffices'.

```
1  /-- X3DH passive security reduces to DDH under the ROM. -/  
2  theorem passive_securety_le_ddh  
3    (g : G)  
4    (adv : PassiveAdversary G SK) :  
5    passiveSecretyAdvantage (F := F) g adv ≤  
6    ProbComp.boolDistAdvantage  
7      (DiffieHellman.ddhExpReal (F := F) g (ddhReduction adv))  
8      (DiffieHellman.ddhExpRand (F := F) g (ddhReduction adv)) := by  
9    -- Step 1: unfold the LHS to expose the two games  
10   unfold passiveSecretyAdvantage  
11   -- Goal is now:  
12   -- boolDistAdvantage (execGame (passiveReal g adv))  
13   --   (execGame (passiveRand g adv))  
14   -- ≤ boolDistAdvantage (ddhExpReal g (ddhReduction adv))  
15   --   (ddhExpRand g (ddhReduction adv))  
16   --  
17   -- We need to show these two pairs of games have the same distributions.  
18   -- Suffices to show:  
19   -- execGame (passiveReal g adv) = ddhExpReal g (ddhReduction adv)  
20   -- execGame (passiveRand g adv) = ddhExpRand g (ddhReduction adv)  
21   -- (as probability distributions)  
22   -- It suffices to show equality of the two distribution pairs  
23   suffices h : ProbComp.boolDistAdvantage  
24     (execGame (passiveReal (F := F) g adv))  
25     (execGame (passiveRand (F := F) g adv)) =  
26     ProbComp.boolDistAdvantage  
27       (DiffieHellman.ddhExpReal (F := F) g (ddhReduction adv))  
28       (DiffieHellman.ddhExpRand (F := F) g (ddhReduction adv)) by  
29     linarith  
30   -- Both boolDistAdvantage expressions depend only on evalDist.  
31   -- The distributional equivalences give us equal evalDist, hence  
32   -- equal probOutput, hence equal boolDistAdvantage.  
33   unfold ProbComp.boolDistAdvantage  
34   have hReal := passiveReal_eq_ddhExpReal (F := F) g adv
```

<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

A composite image of space. In the foreground, a satellite with a large blue parabolic dish and yellow instruments is shown. To the right, a large yellow planet with a prominent ring system is visible. In the background, a diagram of a solar system with a central yellow star and several planets on elliptical orbits is shown against a starry black background.

Make it so!

beneficialaifoundation.org/signal-shot