AlphaProof Code explained: how AI agents formally verified Signal's machine code

21 November 2026 · 9 min read · blogs.iictp.org/alphaproof-code-signal-formal-verification-explained

Short answer: AlphaProof Code is a multi-agent framework from Google DeepMind, described in a technical report dated 10 November 2026, that writes and checks formal proofs about compiled software in the Lean proof assistant. The team used it to prove security properties of libsignal, the Signal protocol's reference implementation, at the level of AArch64 machine code built with the standard Rust compiler. What is new is the scale and the target: about 6.5 million lines of agent-written proof about unmodified production code, with every proof checked by Lean's kernel rather than trusted because an agent said so.

TL;DR The short version

- AlphaProof Code is a Google DeepMind multi-agent framework, described in a 10 November 2026 report, that writes and checks formal proofs in Lean.
- The team used it to prove security properties of libsignal at the level of AArch64 machine code, with every proof checked by Lean's kernel rather than trusted.
- The guarantees cover a narrow scenario and assume perfect cryptography; the lesson for engineers is to invest in precise specifications and trust the checker, not the agent.

Read online, listen to it, or change the text size: https://blogs.iictp.org/alphaproof-code-signal-formal-verification-explained

What is AlphaProof Code?

Formal verification means writing a precise mathematical statement of what a program must do (the specification) and then proving that the program meets it. A proof assistant such as Lean checks each proof step mechanically, so a finished proof rules out every bug that would break the stated property. The catch has always been cost: the report cites the seL4 microkernel at twenty person-years of verification effort and the CompCert compiler at six.

AlphaProof Code splits the work so that humans decide what to prove and AI agents do the proving. The authors' central claim is that once proofs become cheap, the hard part is the specification.

How it works

- **Specification.** A specification-writer agent works with a human to turn security models from published papers into Lean statements. Reviewer agents from a different model family critique each draft for faithfulness and strength, and the human approves the final version, which is pinned at a git commit.
- **Code.** The unmodified Rust code of libsignal is linked into a small command-line harness and compiled with stock rustc to AArch64 Linux machine code. The executable is loaded into Lean together with a formal

semantics of AArch64 (the report gives 467 instructions and 31 system calls in 7,532 lines of Lean). Lean checks the loaded code against the executable's bytes.

- **Proof.** A team of prover agents, with an orchestrator, workers and support roles, proves the specification about the executable. The report says the proofs were produced entirely by agents powered by Gemini 4 Carbon and that none of the authors wrote proof code; specifications were developed with agents powered by Gemini 4 Carbon and Claude Opus 5.5.
- **Gatekeeper.** A gatekeeper agent accepts a proof only if Lean's kernel checks it and a comparator confirms it proves exactly a held-out copy of the approved specification. This exists because agents under pressure sometimes weaken a goal, for example by adding a hypothesis, instead of proving it.

What was proved about Signal

The proofs cover Signal's post-quantum PQXDH handshake and the Triple Ratchet, the parts that agree keys and then rotate them per message. They are stated in two standard models. In the symbolic (Dolev–Yao) model, cryptographic primitives are treated as perfect black boxes; in the computational model, an attacker's success probability is bounded by the hardness of the underlying problems. The report lists forward secrecy (old messages stay safe if a key leaks later), post-compromise security (the session heals after a compromise), authentication, and handshake secrecy, including against a future quantum attacker in the symbolic model.

To check that the specifications can tell good code from bad, the team re-applied two previously fixed libsignal bugs to libsignal v0.105.4, recompiled, and proved in Lean that the resulting binaries violate the symbolic security theorem.

Results reported by the authors

- About 6.5 million lines of Lean proof, of nearly 7 million lines in the repository, according to the report. Google DeepMind's blog post rounds this to over six million.
- About 18,000 lines of trusted statements and definitions, plus about 8,000 lines of AArch64 semantics and the symbol names of the machine-code representation, that a reader must audit to trust the result.
- The verified libsignal code is about 560 KB, about 140,000 AArch64 instructions, per the report.
- The proof work was done by a team of five over four weeks, working in parallel on the symbolic and computational proofs.
- Writing specifications against real code exposed errors in published analyses: the report lists fifteen gaps or bugs in a computational analysis of the Double Ratchet and errors in printed ProVerif queries from earlier work, which the team repaired before proving.

These are the authors' own figures. The report is a Google DeepMind technical report, not a peer-reviewed publication, and the Lean development is published for others to check.

Limitations and open questions

- **Narrow scenario.** The machine-code theorems cover the team's build of a libsignal command-line harness, run as a fresh process, not Signal's own apps. Key generation is not verified, one-time Kyber prekeys are not covered, multi-device state is out of scope, and input sizes and the number of sends are bounded.
- **Symbolic means perfect crypto.** The symbolic guarantees assume perfect cryptography and say nothing about the computational secrecy of bytes. The computational bounds give no guarantee against a quantum

adversary and restrict the attacker in ways the literature does not.

- **Side channels are out of scope.** Leaks in the model reveal only certain keys; timing and other memory secrets are not modelled.
- **Trusted base.** Readers still have to trust the AArch64 semantics, the Linux process model, the contracts of routines the binary calls, the assumed properties of cryptographic primitives, and Lean itself. The Rust compiler is not trusted: a compiler bug that broke a property would make the proof fail.
- **Our reading:** the hard, human part moves to specification review. A wrong but approved specification would be proved just as confidently, which is why the report spends most of its effort on how specifications are drafted, critiqued and pinned.

How to try it or use it

The report says the Lean development, with all specifications and proofs, is published at google-deepmind.github.io/code-verification. We could not load that page when writing, so check its licence and build instructions there. AlphaProof Code itself is described as a research framework; neither the blog post nor the report describes a product or API for running it on your own code.

What it means for engineers and technical teams

- Specification is the skill to invest in. If agents can produce proofs, the people who can state precisely what “secure” or “correct” means for a component become the bottleneck.
- Trust the checker, not the agent. The gatekeeper pattern (a mechanical check against a held-out copy of the goal) applies to any agent workflow: tests, policy checks and migrations included.
- Verify what ships. Proving properties of compiled machine code closes the gap between the design on paper and the binary on devices, where real vulnerabilities often sit.
- Formal methods complement, not replace, testing and threat modelling. See what is threat modeling.

Frequently asked questions

- **What is AlphaProof Code?** A Google DeepMind multi-agent framework in which humans approve Lean specifications and AI agents write proofs that Lean’s kernel checks.
- **Did AlphaProof Code verify the Signal app?** No. It verified security properties of libsignal compiled into a small command-line harness for AArch64 Linux, under stated assumptions and a bounded scenario.
- **Is the paper peer reviewed?** It is a technical report published by Google DeepMind on 10 November 2026, not a peer-reviewed paper at the time of reading.
- **Did humans write any proofs?** The report says no proof code was written by the authors; humans worked on specifications and on running the agent teams.

Related reading

- AI agents explained
- Gemini 4 Carbon: vendor claims and your own evals
- What is DevSecOps?
- Applied AI in the TPBOK

Sources were read on 21 November 2026. Preview APIs, model IDs, prices and access rules change often, so check the linked documentation before you build on them.

Related credential: AI Practitioner. Explore the exam facts and syllabus.

Sources and further reading

1. Chaudhuri, S., Ferraiuolo, A., Kovsharov, A., Chajed, T., Schlesinger, S., Seth, K., Katz, J., Sutton, C., Kohli, P. AlphaProof Code: Verifying Signal's Security at the Machine-Code Level (technical report). 2026-11-10. deepmind.google/documents/36/Formal_Verification_of_Signal_4.pdf
2. Google DeepMind (Kohli, P., Chaudhuri, S.). Introducing AlphaProof Code: a breakthrough in verifying complex software at depth. 2026-11-10. deepmind.google/blog/introducing-alphaproof-code

Published by IICTP, an independent technical credential body, on 21 November 2026. The current version of this post is at https://blogs.iictp.org/alphaproof-code-signal-formal-verification-explained. More guides: blogs.iictp.org.