

# ANTHROPIC

## Appendix: LLMs are making formal software verification feasible at a much larger scale

October 22, 2026

|                                                                                    |   |
|------------------------------------------------------------------------------------|---|
| <a href="#">What our OSS-Fuzz proofs prove</a>                                     | 1 |
| <a href="#">Building a Python interpreter in Rust</a>                              | 2 |
| <a href="#">Details on Claude's generated proofs</a>                               | 3 |
| <a href="#">How do Claude's proofs look to a human formal verification expert?</a> | 4 |
| <a href="#">Details on the formal verification tasks we compare LLMs on</a>        | 5 |
| <a href="#">Assertions in real OSS-Fuzz entry points</a>                           | 5 |

### What our OSS-Fuzz proofs prove

Our proofs concern RISC-V binaries running on Linux. This gives us a stronger guarantee than if we had translated the fuzz tests into Rocq (which risks a translation error) or interpreted their source code (which risks our interpreter disagreeing with the actual behavior of the compiled binary). Instead, we interpret the compiled binary directly, using the formal semantics of RISC-V and Linux processes. We then define the notion of a *crash*: For our purposes, a crash means a fault (e.g. an unmapped memory access), an abort (e.g. a failed assert), or any unmodeled behavior (e.g. a syscall that isn't defined in our Linux process semantics). The basic theorem statement that we aim to prove is that no possible input to a given binary results in a crash.

If we can prove this theorem for the binary that OSS-Fuzz executes (i.e. a particular fuzz test, instrumented with ASan), then that particular binary never needs to be fuzzed again. (Great! No need to burn CPUs on more fuzzing!) However, this is a little unsatisfying. What we *really* want is for this crash-freeness guarantee to carry over to the actual binaries that users run. And users don't typically run binaries with ASan enabled. So now we need to

prove a second theorem: That the ASan-instrumented binary has the same behavior as the normal one.

This raises a thorny question: How do we define “the same behavior?” Our answer is to consider only the following:

- The bytes read from stdin
- The bytes written to stdout and stderr
- The program’s exit code, or the signal that killed it
- The final state of the file system

We say that two programs are equivalent if, for every input and every pair of memory layouts, they produce the same *trace* (that is, the same sequence of read/write events) and have the same ending state. This definition clearly falls short of capturing *everything* of interest about a program—for example, it says nothing about performance—but we claim that it’s sufficient to establish the property we care about: namely, that ASan instrumentation doesn’t seriously alter what the program actually does.

An important wrinkle here is that, if a program crashes, its trace does not include the crash itself—the trace simply ends. This is also the case if the program enters a (silent) infinite loop. As a result, two programs can still be considered equivalent even if they crash for different reasons, or if one crashes and one loops forever.

This brings us to our third and final theorem: We must prove that the normal, uninstrumented binary doesn’t crash either. Otherwise, the normal binary could crash in a branch where the ASan binary loops, and the equivalence theorem wouldn’t catch it.

Putting it all together, we prove that on RISC-V Linux, given two binaries, for all inputs under 64 MiB and all memory layouts:

1. Neither binary encounters a fault, an abort, or any unmodeled behavior
2. The binaries produce the same sequence of read/write events, exit in the same way, and leave the file system in the same state

## Building a Python interpreter in Rust

The following section includes more detail on using formal verification methods to write a Python interpreter in Rust.

We asked Claude to develop an interpreter for a realistic fragment of Python in safe Rust, and to prove it using formal semantics, which is shown (in an empirical way) to adhere to the CPython semantics. Since the interpreter was developed in Rust, we chose [Verus](#) for

verification.<sup>1</sup> In Verus, developers write Rust annotated with formal-logic contracts (requires, ensures, loop invariants). Verus proves that the code satisfies the contracts with the Z3 SMT solver.

For this experiment, we chose a fragment of Python whose semantics is based on [ATT](#), a core language into which its authors translate much of Python. We defined a formal semantics based on the paper’s semantics and Claude used it to prove the implementation of its interpreter while developing the code. The connection of our formal semantics with the “real world” (i.e. CPython) was established empirically. We tested the conformity of the formal semantics and CPython by running over 1.7M (mostly random) Python programs.

**Results.** Claude built the full interpreter, with all 17 tiers, and proved that it correctly implements the formal semantics. Verus verifies every tier’s theorem with no errors. The theorem states, “Whenever the interpreter returns a result, that result is the one the formal semantics prescribes.”

To compare what tests and proofs catch, we deliberately injected seven bugs into an early version of the interpreter. Five survived the random tests, while all seven broke the proof.

## Details on Claude’s generated proofs

Claude ended up writing around one billion lines of Rocq proofs in this project in its attempt to prove a few hundred OSS Fuzz entry points ASan-crash-free. About 180 million of these ended up being part of proofs that successfully passed the proof checker to satisfy the spec. Below, we provide a more detailed example.

The library code behind a typical entry point we attempted is small: a few hundred lines of C, compiling to about 2,000 RISC-V instructions. But the proof has to cover the whole process on every input (C start-up, the C library’s allocator and printf, ASan’s own runtime, the fuzz driver), and today each proof must re-prove all of that from scratch rather than importing it. So a finished proof runs to about 600,000 lines of Rocq in some 1,400 files (ranging from about 330,000 to 900,000 lines across libraries), and its size barely tracks the library’s: in cJSON’s proof only about one-tenth of the lines concern cJSON at all. cJSON’s 391-line parser entrypoint got a 630,000-line proof; fftw3, one of the largest codebases Claude was able to prove ASan-crash free, with 51,000 lines linked in, received a 480,000-line proof.

---

<sup>1</sup> Choosing Verus over Lean was a preference, not a requirement. It was rather a practical choice (Verus fits well with the Rust ecosystem) and to experiment with Claude’s ability with different formalisms. For example using Verus, Claude has to show its ability in making proofs at the source code level with Hoare Logic (a rather different setting than proving code at the binary level).

Over half of each proof is machine-emitted by tools Claude wrote for itself (instruction encodings, lookup tables), roughly 40% is the proof text itself (definitions, models of the machine, lemmas, and proof steps) rather than generated tables, though part of that, too, was produced by scripts rather than explicitly typed out by Claude, and running the proof checker to see if it passes the spec on one proof can take from 30 minutes to several hours of CPU time.

## **How do Claude’s proofs look to a human formal verification expert?**

One interesting aspect of the proofs produced by agents in this project is that the agents proved shorter, more interesting, and more important properties than were strictly required. Early on, we tasked the models with proving equivalence between two versions of each program (fuzzing entry point)—one that linked ASan and one that did not. Almost universally, agents first formulated a short specification describing what outputs a program (fuzzing entry point) can produce, and then proved that both programs match that specification. These specifications were far shorter than the programs being fuzzed: for example, for sanitizer proofs, the median specification size was about 110 lines long, in contrast to the median program having 86 KBytes of code (and the proofs cover around 20–50% of that code, as exercised by the fuzz driver). This is promising in the sense that agents are capable of synthesizing short specifications for more complex programs.

At the next level of detail, the proofs look quite different from those written by human proof engineers. The most striking feature of agent proofs is that they are verbose, repeatedly spelling out details about the program’s execution: specifying the contents of CPU registers, explicitly writing out integer conversions, explicitly spelling out that many different addresses are all distinct from one another, writing out the exact contents of stack addresses, explicitly considering every possible branch in a function, etc. To be sure, these details are important to precisely reason about the execution of code. However, human proof engineers would have immediately introduced abstractions to factor out these types of repetitive specifications and proofs, so as to both capture the precise details and be succinct and general at the same time, such as a notion of a function-level calling convention, a notion of a set of disjoint pointers, an abstraction for the behavior of a memory allocator or stack contents, unifying reasoning about many branches, etc.

Agents did introduce a number of familiar-looking proof constructs in the process of writing their proofs, such as a framing rule resembling that from separation logic, reasoning about code in a way that’s independent of the exact layout / position of the code, and some abstraction of the memory allocator’s behavior, which was critical to reasoning about address sanitizer behaviors. However, not every agent used these

techniques, and the techniques as used by agents correlated with only a modest reduction in proof size; the proofs still remained verbose.

An interesting question is how much these kinds of abstractions matter. The proofs written by agents are just as “good” as any other proof of the same theorem, in that they prove the equivalence theorem. However, it remains to be seen whether agents can scale this verbose proof style directly to more complex programs. Experience from other agent-based verification projects, such as [MachCSL](#), suggests that the answer might be no: the agents hit a complexity wall with the verbose proof style, and scaling up requires the proofs to be structured around better modular abstractions and reasoning principles. If that is the case, how can we help agents discover and introduce the right abstractions to reduce this complexity? Does this require input from expert proof engineers, or can we extract common principles that can guide agents to discovering good abstractions that will enable scalable proofs?

## **Details on the formal verification tasks we compare LLMs on**

We built the tasks by shrinking real OSS-Fuzz entry points while keeping the core of each program. Some run the project’s own fuzz harness unchanged, such as OpenVPN’s base64 harness, which encodes and decodes its input. Others call one library function from a short custom driver, such as libidn’s punycode decoder, LZO’s safe decompressor, miniz’s inflate, zstd’s frame-header decoder and curl’s fnmatch.

To make them small, we replaced everything except the library’s own code. We replaced libc with a few memory and string functions, limited inputs to 64 bytes, and swapped AddressSanitizer for a “mini-ASan” that checks every memory access in about 100 instructions (where the real runtime takes over 100,000). The resulting programs are far smaller (283–2,491 instructions).

The 36 tasks come in three groups of size 12. The first group tasks the agent with proving that a mini-ASan build never errors, the second introduces larger programs, and the third swaps the theorem statement for a more complex statement on the first group programs. The theorem statement is to prove that the plain build and the mini-ASan build of the same program behave identically and neither errors. Each agent ran with a similar harness and proof checker as the full OSS-Fuzz work, but alone.

## Assertions in real OSS-Fuzz entry points

Real OSS-Fuzz entry points already carry checks written by their maintainers, and the proofs in the main post cover those too, because a failed assertion aborts the program and an abort counts as a crash.

LZO's compression fuzz target is one example. It compresses its input, decompresses the result, and asserts that both steps succeed and that it gets back exactly as many bytes as it started with:

```
C/C++
r = lzo1x_1_compress(data, size, compressed, &compressed_len, wrkmem);
assert(r == LZO_E_OK);
if (compressed_len >= size) goto join;
decompressed_len = size + 1;
r = lzo1x_decompress(compressed, compressed_len, decompressed, &decompressed_len,
NULL);
assert(r == LZO_E_OK && decompressed_len == size);
```

To show that neither assertion can fire on any input, Claude had to prove that LZO's compressor and decompressor round-trip correctly, a statement about what the library computes rather than just where it writes. In the binary, each assertion is a branch that jumps to the abort handler; the proof steps through those branches at their exact addresses and shows that they are never taken.

A second example comes from p11-kit, which manages access to smart cards and cryptographic hardware. Its configuration lexer asserts that when it finds a PEM block, the block's length never runs past the end of the input buffer. Claude's proof shows that the check never fails either.