Pre-QED: Environment Specifications for Agentic Verification

Tristan Lim*, Haoxin Tu*, Mehtab Zafar*, Zhengxiong Luo, Abhik Roychoudhury†
National University of Singapore, Singapore
{tristanlimyirong, haoxin.tu, mehtab, luozx, abhik}@nus.edu.sg

Abstract

LLM agents are making formal verification practical at scale. Recently, many attempts have raised this as an enticing possibility. However, formal verification at scale is only meaningful if deep formal specifications capturing functionality are available for checking. Unfortunately, recent works seek to bypass this issue by using shallow specifications devised for purposes other than verification. OSS-Verify from Anthropic and Theorem, for example, reuses the fuzz harnesses of open-source libraries as specifications and has an agent prove that each harness never crashes. However, a fuzz harness follows a single sequence of API calls under a single calling context, so the resulting proof covers one usage scenario rather than the library. This is a core principle our team is well aware of, after more than a decade of experience in developing greybox fuzzing technology.

We argue that the bottleneck of agentic verification has shifted from *how* to prove to *what* to prove. Before zooming into verification at scale, we need deep environment specifications to prove, which is why we call our effort *Pre-QED*. We propose *verification harnesses*, executable harnesses that model all legal usages of an API, so that a proof about the harness is a proof about every well-behaved client. To construct them, we use an agentic workflow that extracts the API's usage contract from code / documentation, and uses Linear-time Temporal Logic (LTL) to rule out illegal call orders. We identify the challenges in proving verification harnesses and release our harnesses open-source as challenges for the community. The overarching vision is that verifying an open-source library should become as routine as fuzzing it, with a guarantee that covers every well-behaved client rather than a single scenario like a fuzzing harness.

CCS Concepts

• **Software and its engineering → Formal software verification**.

Keywords

Software Security, Formal Verification, AI Agents

*Equal contribution with alphabetical order. †Corresponding author: Queries about the paper should be sent to the corresponding author abhik@nus.edu.sg.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference acronym 'XX, Woodstock, NY
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Tristan Lim, Haoxin Tu, Mehtab Zafar, Zhengxiong Luo, Abhik Roychoudhury. 2018. Pre-QED: Environment Specifications for Agentic Verification. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

AI is making software vulnerabilities cheaper to find and to exploit, which raises the bar for defense: rather than adding more tests, we need guarantees that entire classes of bugs are absent [1]. Formal verification can provide such guarantees, but it has historically demanded years of expert effort; the seL4 microkernel, for instance, required roughly twenty person-years to verify [5]. Large language model (LLM) agents are now changing this cost model by writing machine-checked proofs whose correctness is guaranteed by an independent proof checker rather than by the agent itself [14]. OSS-Verify [1, 13] takes this idea to open-source software at scale. Instead of asking developers for specifications, it reuses the fuzz harnesses that projects already maintain for OSS-Fuzz [11], mechanically turns each harness into the theorem that the compiled program never crashes, and lets an agent prove that theorem in Rocq. Applied to 239 entry points across 123 libraries, the pipeline produced kernel-checked proofs and uncovered hundreds of bugs along the way.

The price of this convenience is that the fuzz harness *is* the specification, and a fuzz harness was never written to be one. Its purpose is to drive the library efficiently with random bytes, so it typically hard-codes a single sequence of API calls under a single calling context. A proof about the harness is therefore a proof about that one usage scenario, not about the library that other clients will call. Section 3 makes this concrete with a handle-based API whose fuzz harness opens a handle, reads once, and closes it. OSS-Verify can prove that this harness is crash-free for *every* input up to 64 MB, yet the library still contains an out-of-bounds read that any legal client triggers simply by reading twice from the same handle. The proof is correct, but the guarantee it appears to give is not the one it actually gives. As agents make proofs cheap, this gap becomes the dominant risk: a proof is only as trustworthy as the statement it establishes [10], and a "verified" label on a library whose theorem covers one call sequence can create a false sense of assurance.

Our position is that the bottleneck of agentic verification has shifted from *how* to prove to *what* to prove, and that the right object to prove is still a harness, but *not* a fuzz harness. Since our effort concerns what must be settled before a proof can be concluded, we call it *Pre-QED*[1]. We note that our research group has worked in fuzzing for a decade and our group has brought some of the earliest papers on greybox fuzzing, which have subsequently been adopted in popular fuzzers [2, 3].

[1]QED stands for *quod erat demonstrandum* ("which was to be demonstrated"), traditionally written at the end of a proof.

We propose *verification harnesses*: executable harnesses that decode their input into an arbitrary sequence of API operations while admitting only those that respect the API contract, so that every input corresponds to a legal client and a proof about the harness is a proof about every well-behaved client (Section 4). Because a verification harness keeps the familiar entry point and the same "never reaches an error" theorem, it plugs into the existing proof pipeline unchanged. Building one requires meeting two requirements that mirror the gaps of fuzz harnesses: covering all valid API call sequences while excluding invalid orderings (control flow constraints), and generalizing calling contexts while enforcing preconditions across calls (data flow constraints). We further observe that the ordering rules of an API are naturally temporal properties over call sequences, which lets us state them in Linear-time Temporal Logic (LTL) [9] and use them to guide and check an agentic workflow that extracts the usage contract from code and documentation and synthesizes the harness.

Building on these insights, we construct verification harnesses with an agentic workflow (Section 4). A main agent first studies the library's code and documentation to extract its usage contract, separating ordering rules over the sequence of calls from data rules over the values passed between calls. It then delegates two tracks: a harness subagent writes a candidate executable harness and refines it against code-coverage feedback, while an LTL subagent formalizes the ordering rules as temporal properties and refines them through spot checks and an LLM-as-judge assessment. A final synthesis stage reconciles the two, so that the control flow of the harness implements the validated temporal properties and every guard is traceable to the rule it enforces. On the motivating example, the resulting harness admits the client that reads twice from the same handle, so the proof attempt exposes the out-of-bounds read instead of missing it. Proving such harnesses, however, is harder than proving fuzz harnesses: instead of following one fixed call sequence, the prover must reason about arbitrarily long sequences of API operations and about library state that persists across them (Section 6). We therefore release our verification harnesses as open challenges for agentic verifiers.

Our contributions. Our vision is that verifying an open-source library should become as routine as launching a fuzzing campaign, while the guarantee covers every well-behaved client rather than a single scenario. Toward this vision, this paper makes the following contributions:

- We show that proofs derived from fuzz harnesses cover a single API usage scenario, and can hold while the library remains buggy.
- We introduce *verification harnesses*, which model all legal usages of an API, and an LTL-guided agentic workflow to construct them automatically. This also obviates the need to rely on existing fuzzing harnesses for formal verification. We have an automated procedure for verification harness generation, and then we have checked them manually.
- We identify the challenges in proving verification harnesses and release our harnesses as open challenges, inviting the community to move agentic verification approaches from proving harnesses to proving the entire library. We have thus released the verification harnesses for the 29 libraries as open source at https://pre-qed.github.io/.

2 Background: The Oss-Verify Proof

2.1 Verification Pipeline

OSS-Verify [1], developed jointly by Anthropic's Frontier Red Team and Theorem, demonstrates that proofs written by large language models (LLMs) can make formal verification practical for real-world open-source software at scale. The work is motivated by the observation that AI has substantially lowered the cost of discovering and exploiting software vulnerabilities, which calls for guarantees that entire classes of bugs are absent, rather than for additional testing alone. Formal verification can provide such guarantees, but it has traditionally required years of expert effort; for instance, the seL4 microkernel [5] required approximately two person-years to build and twenty person-years to verify. OSS-Verify reduces this cost in two ways: it reuses artifacts that developers already maintain, and it delegates proof construction to an LLM agent whose output is validated by an independent proof checker.

The pipeline takes as input an open-source library together with its existing fuzz harness, namely the `LLVMFuzzerTestOneInput` entry point already exercised by OSS-Fuzz [11]. No additional annotations or modifications by the library developers are required. The library and the harness are compiled into a single statically linked RISC-V [16] executable. Consistent with OSS-Fuzz practice, C/C++ code is built with AddressSanitizer (ASan) [12] and UndefinedBehaviorSanitizer (UBSan) [7], so that invalid memory accesses and undefined behavior manifest as crashes rather than as silent corruption. Reducing the verification target to one binary is essential for formal reasoning, because the behavior of the program is then fully determined by its machine code and by the formal models of the processor and the operating system on which it executes.

The executable is then combined with Theorem's formal semantics of RISC-V and of Linux processes, together with the property that the program never crashes, to generate a specification mechanically; the specification states, in essence, that the fuzz test always passes, and no part of it is written by hand. A Claude agent subsequently constructs a proof of this specification in Rocq, which the Rocq kernel checks. If the agent encounters a genuine bug, it patches the library, the specification is regenerated for the patched binary, and the agent proves the updated specification. To discourage reward hacking, such as rewriting the program so that it trivially cannot crash, patches are penalized, additional credit is granted for proving which inputs a patch affects, and all patches are logged and audited. Applied to 239 entry points across 123 OSS-Fuzz libraries, the pipeline uncovered hundreds of bugs, including some that fuzzing is unlikely to reach, such as defects triggered only by inputs of tens of megabytes.

2.2 Proof Obligations

The central premise of OSS-Verify is that no separate specification is authored: the fuzz harness itself defines what constitutes correct behavior for the target library. This premise determines the scope of the verified claim. The harness fixes the library API usage, so the proof covers exactly the calls issued by the harness rather than arbitrary client code. The proven property is that, for every input smaller than 64 MB, the program terminates normally, without memory-safety violations or assertion failures, under every modeled operating-system response and memory layout. The

64 MB bound is a parameter of the specification and can be raised by regenerating the specification and repeating the proof. Since many harnesses additionally encode higher-level properties, such as a compress-then-decompress roundtrip, establishing that the harness never fails also establishes those properties.

The trustworthiness of the result rests on a small trusted computing base (TCB) consisting of three components. First, the harness and the exact theorem statements must capture the intended notion of correctness, since the proof establishes only what they state. Second, the environment model, i.e., the formal semantics of RISC-V and of single-threaded Linux processes, must faithfully reflect the behavior of real hardware and operating systems. Third, the Rocq proof checker must be sound, together with the platform on which it runs, including the OCaml compiler, the operating system, and the hardware. Given these components, the proofs themselves, which typically comprise approximately 600,000 lines of Rocq, require essentially no manual review.

The boundaries of this TCB also delineate the directions in which the approach must be extended: reducing the token cost of proof generation, reusing proofs across shared dependencies, and broadening the machine and operating-system models beyond statically linked, single-threaded RISC-V binaries, so that verification may become as routine as fuzzing. However, the work tries to use the fuzz harnesses from OSS-Fuzz as the program environment for verification, which may not be a good fit for verification purposes.

3 Motivation: What the Proofs Leave Out

3.1 Motivating Example

To illustrate why a proof about a fuzz harness falls short of verifying the underlying library, we show a representative library API (Figure 1a) and its conventional fuzz harness (Figure 1b).

The library exposes three functions: `h_open()` initializes a handle `h` over an input buffer `in` of length `n` with capacity `h->cap = n` and cursor `h->pos = 0` (lines 3–9); `h_read()` copies up to `len` bytes from `h->data + h->pos` into a caller buffer `buf`, clamping the read at stream end and advancing `h->pos` (lines 10–18); and `h_close()` frees `h` (line 19). Following standard OSS-Fuzz practice, the developer-written `FuzzHarness` (lines 18–26) opens a handle `h` on the fuzzer input, performs a single `h_read()` into a 64-byte buffer `buf` with `len = sizeof(buf)`, and closes `h`.

What the Proof Establishes. When `FuzzHarness` is fed into the OSS-Verify pipeline (Section 2), verification passes: the prover successfully establishes a machine-checked proof that the compiled binary is crash-free (and trace-equivalent across builds) for *all* inputs up to 64 MB. Indeed, along the single execution path fixed by lines 20–24, `h_read()` is invoked at most once per handle with `h->pos == 0` and `len == 64`. When `h->pos == 0`, clamping `len` to `h->cap` at line 14 is exact: whether `n < 64` or `n >= 64`, the `memcpy` at line 15 copies min(64, n) bytes from `h->data + 0` and stays within bounds, so the verification goal holds unconditionally.

What the Proof Misses. Although formal verification passes on `FuzzHarness`, the library still contains a genuine stateful out-of-bounds read at line 14. Like standard stream APIs (e.g., `fread` or `gzread`), `h_read()` is designed to be called repeatedly in chunks until it returns `0` at EOF (`h->pos >= h->cap`, line 12); clamping

```
1  typedef struct { const uint8_t *data; size_t cap, pos; } handle_t;
2
3  handle_t *h_open(const uint8_t *in, size_t n) {
4    if (n < 4) return NULL;
5    handle_t *h = malloc(sizeof(handle_t));
6    if (!h) return NULL;
7    h->data = in; h->cap = n; h->pos = 0;
8    return h;
9  }
10 int h_read(handle_t *h, uint8_t *buf, size_t len) {
11   if (h->pos >= h->cap)
12     return 0;     /* EOF: no remaining bytes */
13   if (len > h->cap)
14     len = h->cap; /* BUG: clamps to h->cap, not h->cap-h->pos */
15   memcpy(buf,h->data+h->pos,len);/*OOB read on a second h_read()*/
16   h->pos += len;
17   return (int)len;
18 }
19 void h_close(handle_t *h) { free(h); }
```

(a) Target library code (`h_open()`, `h_read()`, `h_close()`).

```
18 int FuzzHarness(const uint8_t *in, size_t n) {
19   uint8_t buf[64];
20   handle_t *h = h_open(in, n);
21   if (h != NULL) {
22     h_read(h, buf, sizeof(buf));
23     h_close(h);
24   }
25   return 0;
26 }
```

(b) Conventional fuzz harness exercising a single call sequence.

Figure 1: A representative library API with a stateful out-of-bounds read across multiple `h_read()` calls (a), and its conventional fuzz harness (b).

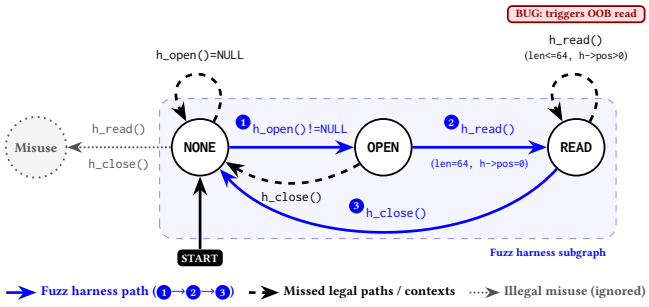


Figure 2: Finite automaton (over finite sequences of API calls) of the library API. `NONE` is the initial state, and all states except Misuse are accepting, so a call sequence is legal iff it never reaches Misuse. The conventional fuzz harness (Figure 1b) covers only the shaded blue subgraph (1→2→3) under a single context (e.g., `len = 64`, `h->pos = 0`), missing the black dashed self-loop on `READ` for repeated `h_read()` calls that triggers the OOB read.

a caller's request to the remaining bytes is the library's own responsibility. However, line 14 clamps `len` against the total capacity `h->cap` rather than the remaining bytes `h->cap - h->pos`. Consequently, any legal client that reads in two chunks—for instance, calling `h_read()` with `len = 40` twice on a stream of `h->cap = 64` bytes—sees `h->pos = 40 < h->cap` (not EOF) and `len = 40 <= h->cap` on the second call, causing `memcpy` at line 15 to read 16

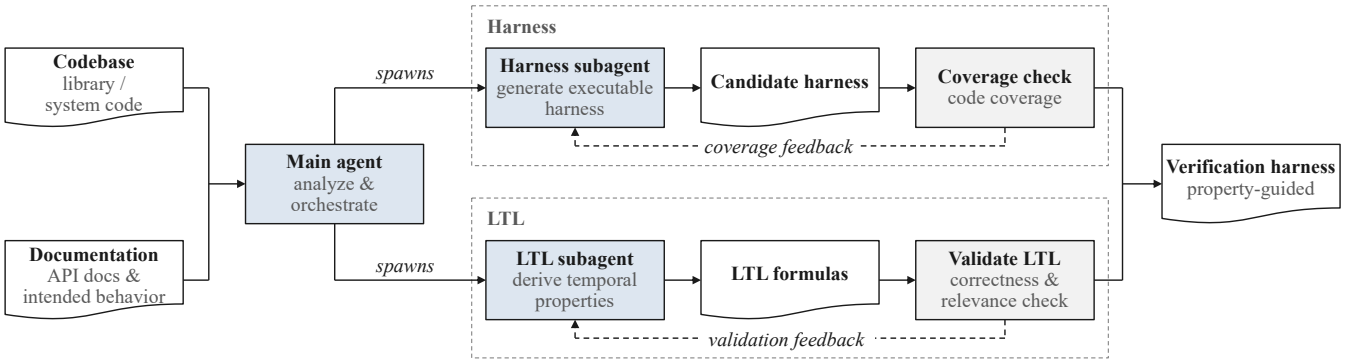


Figure 3: Agentic verification-harness construction workflow.

bytes past the end of `h->data`. Because `FuzzHarness` hard-codes a single one-shot sequence (`h_open()` → `h_read()` → `h_close()`), the verified theorem is completely silent on chunked reads.

3.2 Gap Analysis

To understand why this gap is structural rather than incidental to a single harness, we model the legal usages of a library API as an *automaton* (Figure 2). It is a finite automaton over finite sequences of API calls, in which every state except Misuse is accepting: a call sequence is legal if and only if it never reaches Misuse, so every prefix of a legal sequence is itself legal. This choice follows from the fact that all ordering rules we consider are safety properties: every violation is witnessed by a finite *bad prefix* [6], which here is a sequence of calls that reaches Misuse, and a harness enforces these rules by never taking a transition into Misuse. Liveness rules, such as requiring that every opened handle is eventually closed, cannot be witnessed by a finite prefix and would instead require a Büchi automaton [15] over infinite sequences (Section 6).

In this automaton, states represent the lifecycle of library objects—uninitialized or closed (`NONE`), freshly opened at `h->pos = 0` (`OPEN`), and actively being read (`READ`)—transitions correspond to API calls, and each transition is parameterized by a *calling context* (the arguments passed to the call and the internal state accumulated along the path, such as `len`, `h->cap`, and `h->pos`). Viewed through this lens, repurposing a conventional fuzz harness as a formal specification suffers from two structural limitations:

(1) Sequence (Single Path vs. Full Automaton). In Figure 1b, `FuzzHarness` traverses only the three-step subgraph ❶ `h_open()` → ❷ `h_read()` → ❸ `h_close()` (`NONE` → `OPEN` → `READ` → `NONE`) and terminates. Every other valid cycle in Figure 2, including the black dashed self-loop on `READ` that invokes `h_read()` repeatedly and the reopen cycle across handles, is syntactically absent from the binary and therefore invisible to the theorem. A natural attempt to cover more of the automaton is to invoke APIs in random order; however, unconstrained random sequences immediately take illegal transitions—such as calling `h_read()` or `h_close()` in state `NONE` (the gray dotted edge to Misuse in Figure 2)—and produce client-side misuse crashes that falsify the theorem without implicating the library. To verify the full automaton without admitting such misuse, the harness must explicitly encode the boundary between legal and illegal call sequences.

(2) API Call Context (Single Scenario vs. General Preconditions). Even along the transitions it does take, a fuzz harness hard-codes a single calling scenario to satisfy API preconditions, rather than ranging over all valid contexts. In Figure 1b, the transition from `OPEN` to `READ` is always taken at `h->pos == 0` with a fixed request size `len = 64` on a single handle `h`. In reality, the precondition of `h_read()` permits *any* `len <= sizeof(buf)` at *any* accumulated cursor offset `h->pos` produced by prior reads in state `READ`, as well as multiple handles coexisting in memory. By collapsing this space of legal contexts to a single point (`len = 64`, `h->pos = 0`), the proof in Section 3 passes for that one scenario while leaving genuine library defects triggered under general contexts (Figure 1a, line 14) unverified.

4 Construction of Verification Harnesses

4.1 Definition and Requirements

The proofs discussed in Section 2 are only as general as the fuzz harness from which their specification is derived (Section 3). Because that harness (Figure 1b) fixes a single API call sequence and a single calling context, the resulting guarantee covers one usage scenario rather than the library as a whole. This is why the proof in Section 3 is silent on the stateful out-of-bounds read of Figure 1a, which manifests only when `h_read()` is invoked repeatedly on the same handle. We therefore introduce the notion of a *verification harness*: an executable harness that models *all* legal usages of the target library's API, so that a proof about the harness is a proof about every well-behaved client. Figure 4 illustrates the idea for the target library in Figure 1a. Instead of issuing a fixed sequence of calls, the harness decodes its input bytes into an arbitrary sequence of API operations, and it admits only those operations that respect the API contract, so that every decoded input corresponds to a legal client. The theorem to be proved is then that, for all inputs, the program never reaches an error state.

Two requirements follow directly from the gaps identified in Section 3.

(1) **Control flow (API call sequence):** The harness must cover every valid sequence of API calls while excluding invalid call orders. In Figure 4, this is the role of the explicit state variable that records whether a handle is open and thereby encodes the ordering rules (O1, O2).

(2) **Data flow (API call context):** The harness must generalize the calling contexts, rather than fixing a single scenario, while enforcing the API preconditions across calls. In Figure 4, the handle, the buffer, and its length are derived only from values that the harness owns, which satisfies the data rules (D1–D3), whereas the remaining inputs, such as the raw bytes passed to the open operation, remain unconstrained.

A harness that violates the first requirement either misses reachable behavior or exercises illegal usages whose failures are attributable to the client rather than to the library; a harness that violates the second requirement either proves too little, by remaining tied to one calling scenario, or too much, by admitting contexts that violate the preconditions.

4.2 Verification Harness Generation

Writing such a harness by hand requires a thorough understanding of the library's API contract, and the contract is typically scattered across the source code and the documentation. We therefore construct verification harnesses with an agentic workflow (Figure 3) and model the required control-flow constraints with Linear-time Temporal Logic (LTL) [9]. LTL is a natural fit because the ordering rules of an API, such as "a handle is read only after it has been opened and before it is closed", are properties over sequences of calls. Expressing them as LTL properties makes the intended usage protocol explicit and checkable, and it provides a precise reference against which the generated harness can be assessed.

The workflow proceeds in four stages. First, a main agent *analyzes and orchestrates*: it studies the codebase together with the API documentation to understand the library and its intended behavior, and then delegates two focused tracks. Second, a *harness subagent* generates a candidate executable harness; the candidate is validated through a code-coverage check, and the resulting feedback is returned to the subagent for refinement. Third, an *LTL subagent* derives temporal properties from the code and the documentation; these formulas are validated through spot checks and an LLM-as-judge assessment of their correctness and relevance, and the feedback is again used to refine them. Finally, the *synthesis* stage combines the validated outputs into the final verification harness, which is both executable and guided by the validated properties.

4.3 An Example of Verification Harness

We use the harness in Figure 4 to illustrate how the workflow in Figure 3 produces a verification harness for the target library in Figure 1a (contrasting with its conventional fuzz harness in Figure 1b). The target is a small library with a handle-based API consisting of `h_open()`, `h_read()`, and `h_close()`. Throughout the example, O1 and O2 denote the API's ordering rules, and D1–D3 denote its data rules. The workflow extracts these rules as follows.

Analyze & Orchestrate. The main agent reads the library's code and documentation, and it extracts the usage contract of the API. Ordering rules: (O1) a read is legal only while a handle is open; (O2) at most one handle is live at a time, and a close is legal only for a handle that is currently open. Data rules: (D1) a handle is usable only if `h_open()` returned it as non-NULL; (e.g., `requires \valid(h)`) (D2) after `h_close()`, the handle must not be used again; (D3) the length passed to `h_read()` must not exceed the size

```c
/* Verification harness: every decoded input is a legal
   client of the API. Theorem to prove: for all inputs,
   this program never reaches an error. */

/* Call-order state: is a handle open? (O1, O2) */
enum { NONE, OPEN } st = NONE;
/* The open handle, or NULL if none (D1, D2) */
handle_t *h  = NULL;
/* Read buffer of known size, used to bound len (D3) */
uint8_t  buf[64];

int VerificationHarness(const uint8_t *in, size_t n) {
  size_t i = 0;
  while (n - i >= 2) {
    uint8_t kind = in[i], arg = in[i + 1];
    i += 2;
    switch (kind % 3) {

    case 0: /* OPEN */
      /* O2: at most one live handle, so close it first */
      if (st == OPEN) { h_close(h); st = NONE; }
      if (arg > n - i) break;
      /* Data left unconstrained: the library must check it */
      h  = h_open(in + i, arg);
      i += arg;
      /* D1: the handle is usable only if it is non-NULL */
      st = (h != NULL) ? OPEN : NONE;
      break;

    case 1: /* READ */
      /* O1, D1: read only an open handle; else skip */
      if (st != OPEN) break;
      /* D3: len < sizeof(buf), so the read fits in buf */
      h_read(h, buf, arg % sizeof(buf));
      break;

    case 2: /* CLOSE */
      /* O2: close only an open handle; else skip */
      if (st != OPEN) break;
      h_close(h);
      /* D2: drop the closed handle so it is never reused */
      h  = NULL;
      st = NONE;
      break;
    }
  }
  /* Close a handle left open at the end, so nothing leaks */
  if (st == OPEN) h_close(h);
  return 0;
}
```

Figure 4: An example verification harness for the library in Figure 1a, generalizing the fuzz harness in Figure 1b.

of the buffer passed alongside it. (e.g., `requires \valid(buf + (0 .. len-1))`) The main agent then delegates the ordering rules to the LTL track and the construction of an executable harness to the harness track.

LTL Subagent. From the ordering rules, the LTL subagent derives temporal properties over the sequence of API calls, for example

$$\varphi_{O1} = \neg\, \mathit{read}\ \mathbf{W}\ \mathit{open} \ \wedge\ \mathbf{G}\big(\mathit{close} \rightarrow \mathbf{X}(\neg\, \mathit{read}\ \mathbf{W}\ \mathit{open})\big),$$

$$\varphi_{O2} = \mathbf{G}\big(\mathit{open} \rightarrow \mathbf{X}(\neg\, \mathit{open}\ \mathbf{W}\ \mathit{close})\big) \ \wedge\ \big(\neg\, \mathit{close}\ \mathbf{W}\ \mathit{open}\big),$$

where **G**, **X**, and **W** denote "always", "next", and "weak until". In words, a read never occurs before the first open or after a close until the next open, and a handle is never opened twice without an intervening close, nor closed unless it is open. Spot checks and the LLM-as-judge assessment confirm that these formulas agree with the documentation and that none is vacuous or irrelevant to the

API. The data rules D1–D3 are constraints on values rather than on the order of calls, so they are not expressed in LTL; they are handled by the harness track.

Harness Subagent. The harness subagent turns the contract into an executable harness (`VerificationHarness` in Figure 4). To cover *all* valid call sequences, it replaces the fixed sequence of calls in `FuzzHarness` (Figure 1b) with a decoder: each pair of input bytes selects one operation (`kind % 3`) and its argument, and the loop continues until the input is exhausted. Every sequence of operations is thereby reachable from some input, including the repeated reads that the fixed sequence of Figure 1b never issues. To restrict the sequences to legal ones, it realizes the automaton of φ_{O1} and φ_{O2} as the state variable `st`. A read or a close whose guard does not hold (`st != OPEN`) is skipped, so that the input remains legal and no input is rejected, and an open in the `OPEN` state first closes the live handle, which preserves O2. For the data rules, the harness owns the only handle `h` and the only buffer `buf`, updates `st` from the return value of `h_open()` (D1), resets `h` on close (D2), and derives the read length as `arg % sizeof(buf)` (D3). Inputs that the contract leaves open remain unconstrained: the raw bytes passed to `h_open()` are forwarded without validation, because rejecting malformed inputs is the library's responsibility, and constraining them would exclude exactly the behavior that the proof is meant to cover. The coverage check then confirms that every API function and every guard branch of the harness is reached; unreached branches are returned to the subagent as feedback, for instance to add a missing reopen case after a close.

Synthesis. Finally, the synthesis stage checks that the candidate harness and the LTL formulas agree, and it emits the final verification harness. The check is traceable by construction, since each guard and state update in the harness is annotated with the rule (O1, O2, D1–D3) that it enforces. The synthesized harness additionally closes any handle that is still open at the end of the input, so that every run ends in a state permitted by the rules and the run does not leak. The resulting harness is executable, as a fuzz harness is, and property-guided, because its control flow implements the validated temporal properties. The theorem to be proved is that, for all inputs, the program never reaches an error, which by construction is a statement about all legal clients of the API. Unlike the proof for the fuzz harness in Section 3, this theorem cannot be established for the library of Figure 1a: an input that decodes to `h_open()` on a 64-byte stream followed by two `h_read()` operations of 40 bytes each leaves `len` unclamped at line 14 on the second call (`40 <= 64`) and reads 16 bytes past the end of the buffer at line 15, so the proof attempt exposes the defect rather than missing it.

5 Statistics of Verification Harnesses

Subjects. Of the libraries verified by OSS-Verify (Section 2), 32 have publicly released proofs[2], and we take these 32 as our candidate subjects. We construct verification harnesses for 29 of them and exclude the remaining three for the following reasons. One subject `reac-pw` is an application rather than a library: it has no API that clients call in sequence, so the legal call sequences that a verification harness should model are not well defined. Two subjects, `openssl` and `mbedtls`, each expose more than 2,000 API functions; a harness that covers such an API would be too large to be reviewed properly, and an unreviewed harness may not serve as a trusted specification.

[2]https://github.com/theorem-labs/oss-verify-proofs/tree/main#what-is-proved

Table 1: Comparison between fuzzing and verification harnesses in terms of API usage coverage (refs. API Cov(%)) and the number of LTL constraints (refs. # LTL), ordered by increasing size (LoC) of the core library code.

		Fuzz harness	Verification harness	
Subject	Size	Cov. (%)	Cov. (%)	# LTL
reac-repacer	290	100.0	100.0	0
libdiginet	1,659	85.1	100.0	5
cgif	1,760	100.0	100.0	2
yajl-ruby	3,087	11.5	100.0	3
double-conversion	3,921	6.2	84.4	3
fast_float	3,945	8.3	100.0	0
cjson	4,002	6.5	100.0	5
bzip2	5,871	33.3	100.0	5
json-c	6,198	31.5	78.5	8
skcms	7,095	10.5	100.0	2
plugin-hostd	9,726	0.0	100.0	1
coturn	11,938	37.6	97.3	6
tinyobjloader	13,136	5.5	96.4	4
libreac	13,202	17.9	100.0	33
zlib	17,445	58.0	93.2	6
speex	18,863	35.8	100.0	6
omx-clap-core	20,378	0.0	76.5	7
boost-json	26,496	5.1	63.3	4
pupnp	34,463	0.8	99.6	20
simdjson	36,709	13.1	90.3	15
boost-beast	39,776	2.6	62.3	16
libpcap	46,453	9.4	88.7	4
zstd	50,891	30.3	90.4	13
msgpack-c	76,964	2.1	82.4	11
glaze	85,456	7.9	92.5	5
jsoncons	89,201	4.2	61.0	10
spirv-tools	109,870	6.0	62.9	18
varnish	120,139	0.7	71.7	20
leptonica	144,883	10.2	97.5	2

Comparison with Fuzzing Harnesses. Table 1 compares, for each of the 29 subjects, the existing fuzz harness used by OSS-Verify with our verification harness. The subjects are ordered by increasing size (LoC) of the core library code, excluding the tests, fuzzers, and examples. For both harnesses, we report the API coverage (Cov. %), i.e., the number of APIs that appear in the harness divided by the total number of APIs in the library, and for the verification harness we additionally report the number of LTL properties (# LTL) extracted by the LTL subagent (Section 4).

On average, the verification harnesses reach 89.3% coverage, compared with 22.1% for the fuzz harnesses, an improvement of 67.2 percentage points. Two subjects, `reac-repacer` and `cgif`, are already fully covered by their fuzz harnesses and remain at 100%, and the verification harnesses reach 100% coverage on 11 subjects in total, compared with two for the fuzz harnesses. The largest gains occur for `plugin-hostd` (0.0% to 100.0%), `pupnp` (0.8% to 99.6%), `cjson` (6.5% to 100.0%), and `tinyobjloader` (5.5% to 96.4%), whose fuzz harnesses cover less than 10% of the API, whereas the verification harnesses invoke nearly all API functions in arbitrary legal

orders. In total, the workflow extracts 234 LTL properties, 8.1 per subject on average (ranging from 0 to 33, the latter for `libreac`), and each of them is enforced by the control flow of the corresponding harness. Two subjects, `reac-repacer` and `fast_float`, have no LTL properties because their APIs are stateless: API calls do not carry anything between their calls, so there is no ordering constraint on the call sequence to express. Cov. measures the breadth of the API that a harness exercises; it does not capture call sequences or calling contexts, whose generality follows instead from the construction of the verification harness (the input decoder together with the LTL-guided guards in Section 4). Accordingly, a fuzz harness with full API coverage, such as those of `reac-repacer` and `cgif`, may still fix a single call sequence and calling context. Even by this breadth measure alone, however, the numbers indicate that the theorems obtained from fuzz harnesses leave most of the API of the libraries unexercised, and the verification harnesses close much of this gap. Where the verification harness does not reach every API, we check the uncovered APIs, and the gaps fall into a few categories: APIs that only exist in other build configurations (e.g., `libpcap`, `pupnp`), and APIs that spawn processes or touch the filesystem (e.g., `leptonica`), code that is only used by a daemon or that depends on private libraries (e.g., `varnish`, `omx-clap-core`).

Manual Efforts Involved. Our workflow produces the verification harnesses automatically, and we spend manual effort mainly to further increase our confidence in them before we use them as specifications. Because each harness is short and structured around the extracted API usage protocol, we can review it easily: we mostly look at code quality (e.g., unnecessary complexity added to increase code coverage, unexplained use of constants) and at whether the harness has the desired shape, i.e., that it drives the API through the legal call sequences and calling contexts described in Section 3.2 instead of a single hard-coded scenario. We also carefully check the library documentation to make sure that the API call sequences encoded in each harness align with the sequences that the documentation expects. In all cases, we need only light-weight inspection and do not have to rewrite the harnesses.

6 Towards Proving Verification Harnesses

Proving Lemmas over Complex API Call Sequences. The existing entry-point proofs succeed largely because a fuzz harness follows a single, fixed sequence of API calls, each starting from freshly initialized library state. A verification harness removes this structure: its decoding loop admits arbitrarily long sequences of operations, and the library state, such as the cursor `h->pos` in Figure 1a, persists and evolves across them. Following each sequence symbolically is no longer possible, so the prover must instead establish a loop invariant that characterizes every library state reachable by a legal client, together with per-API lemmas showing that each operation, started in a state satisfying the invariant and called with arguments that satisfy its preconditions, terminates without error and preserves the invariant. For the example library, the invariant must state that `h->pos` never exceeds `h->cap`, which is exactly the fact that the faulty bounds check fails to maintain. For real libraries, these invariants involve heap-allocated structures, ownership of buffers across calls, and internal state machines, and they must be proved at the level of the compiled binary. A verifier for verification harnesses therefore needs the agent to discover such invariants and API-level lemmas, to prove them once, and to compose them across calls, rather than to re-establish safety for every call sequence. For this reason, it is our position that such realistic agentic verification of a library using verification harnesses will need significant *specification inference* (see a recent work from our group [17]). In particular, determining the appropriate level of abstraction at which to formulate these invariants and API-level lemmas requires careful design. They should be concrete enough to be provable at the machine code level, yet general enough to be composed and reused. Furthermore, proving these lemmas at the machine code level may be complicated by low-level implementation details. Therefore, recovering and incorporating the abstractions present in the higher-level source program may be necessary to guide the verification process and make machine code verification more scalable. This was most likely not needed for pushing through the proofs that Anthropic and Theorem presented using fuzzing harnesses.

Proving Under LTL Constraints. Our verification harness admits only legal clients because its control flow implements the LTL properties of the API, so the proof must also reason about the LTL properties. All the ordering rules we extract, such as φ_{O1} and φ_{O2}, are safety properties, and the harness realizes them through an explicit automaton state such as `st`. The loop invariant must therefore relate the automaton state to the library state, for instance, that `st == OPEN` implies that `h` points to a live, well-formed handle, and that `st == NONE` implies that no handle is used. Beyond crash-freedom, the verifier should also establish that every trace of the harness satisfies the extracted LTL properties, which turns the agreement between the harness and its properties from a checked artifact into a proven fact and removes it from the trusted computing base. Supporting richer protocols, such as properties that require a resource to be eventually released, would further require reasoning about liveness, for example by reducing it to safety or termination [4, 8]. These capabilities, namely invariant discovery over persistent state, compositional API-level lemmas, and protocol-aware reasoning, are what the released verification harnesses are designed to challenge.

7 Related Work

Formal verification has long delivered strong guarantees for critical systems, but at the price of substantial expert effort. The seL4 microkernel [5] is the canonical example. Recent work reduces this effort with LLM agents that construct machine-checked proofs. AutoRocq [14] applies an agentic approach to generate Rocq proofs for software systems, and OSS-Verify [1, 13] scales the idea to open-source libraries by deriving specifications mechanically from existing fuzz harnesses. These efforts focus on *how* to prove a given specification. In OSS-Verify, the specification inherits the single call sequence and calling context of the fuzz harness (Section 3). This direction fits the broader vision of agentic software engineering, in which the focus of programming shifts from scale to trust in what agents produce [10]. A proof is only as trustworthy as the specification it establishes. Our work addresses the complementary problem of *what* to prove, namely a verification harness that covers all legal usages of an API. Our use of temporal logic for this purpose relates to verifying temporal properties of programs [9]. Prior work on liveness and termination [4, 8] studies how to *establish* such

properties for a given program, e.g., by reducing liveness to safety or to termination. In contrast, we use LTL properties to *specify* the legal API-usage protocol that constrains the harness, and we leave the proof of the resulting theorem to the entry-point proof pipeline.

In summary, existing verification efforts either demand expert-written specifications or reuse the fuzz harnesses that projects already maintain for continuous fuzzing [11]. Our vision is to provide verification harnesses that model all legal usages of an API and are constructed automatically from the code and documentation. With such harnesses, verifying an open-source library becomes as easy as launching a fuzzing campaign, while the guarantee extends from one scenario to every well-behaved client.

8 Perspectives

Formal verification at scale using AI is a topic of high interest. As AI-based formal verification takes centre-stage - the focus shifts to two key aspects.

- how to verify real software systems using AI agents instead of protocols.
- more importantly, how to find specifications to capture the operating environments of software being checked, and the properties to check.

In this work, we have taken forward our recent past experience in agentic formal verification of software [14] to focus on the environment specifications of a software library. We posit that fuzzing harnesses should not be used as environment specifications for formal verification.

We present verification harnesses, which shift a proof about a single usage scenario into a guarantee for every legal client, and an agentic workflow that constructs them automatically. The harnesses we construct form a set of ready-to-use, library-level verification targets, which we release as open challenges so that future work can focus on *how* to prove them, as well as on richer properties than crash-freedom, proofs that compose across library dependencies, and support for callbacks, global state, and concurrency.

We also seek to make another observation related to fuzz testing while presenting our findings. Even though the verification harnesses we presented are meant for formal verification - they can be used equally well, for *more extensive fuzzing of a software library as well.* Thus, the fuzzing community could also benefit from our efforts. Indeed in our experiments, we observed new bugs surface when we subjected the open source libraries to testing using the verification harnesses.

We envision a future in which verifying an open-source library is as routine as fuzzing it. We recall how OSS-Fuzz made continuous fuzzing standard practice by asking projects only for a harness. Similarly, moving forward, a software project would contribute a verification harness, agents would prove it, the agent-constructed proof would be validated by a theorem prover. Furthermore, the proof would be re-checked and repaired as the library evolves. In this setting, the verification harness and its temporal properties become the central, trust-critical artifact, maintained and reviewed alongside the code as the library's API contract. We invite the verification and AI communities to build on our harnesses and advance agentic verification from proving based on fuzz harnesses, to proving *actual* software libraries.

Data Availability

We release the verification harnesses as open-source challenges for the verification community. More information can be found at:

https://pre-qed.github.io/

Acknowledgments

This work was partially supported by a research project "AI for Program Reasoning", project AI4SCH-2025-0084 under AI for Science program of National Research Foundation (NRF) Singapore. Any opinions, findings and conclusions, or recommendations expressed in this material are those of theauthor(s) and do not reflect the views of National Research Foundation, Singapore. Thanks to Anthropic's Frontier Red Team and Theorem for making the OSS-Verify proofs publicly available.

References

[1] Anthropic. 2026. LLMs are making formal software verification feasible at a much larger scale. https://www.anthropic.com/research/formal-verification. Blog post, Oct 22, 2026; accessed 2026-10-29.

[2] Marcel Boehme, Van-Thuan Pham, MD Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of ACM Conference on Computer and Communications Security (CCS).* https://abhikrc.com/pdf/CCS17.pdf.

[3] Marcel Boehme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transcations on Software Engineering, Extended paper of conference version appearing in CCS 2016,* 45, 5 (May 2019). https://abhikrc.com/pdf/TSE19.pdf.

[4] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. 2007. Proving that programs eventually do something good. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07).* Association for Computing Machinery, New York, NY, USA, 265–276. doi:10.1145/1190216.1190257

[5] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (June 2010), 107–115. doi:10.1145/1743546.1743574

[6] Orna Kupferman and Moshe Y Vardi. 2001. Model checking of safety properties. *Formal methods in system design* 19, 3 (2001), 291–314.

[7] LLVM Project. [n. d.]. UndefinedBehaviorSanitizer – Clang documentation. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. Accessed 2026-11-02.

[8] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2017. Reducing liveness to safety in first-order logic. *Proc. ACM Program. Lang.* 2, POPL, Article 26 (Dec. 2017), 33 pages. doi:10.1145/3158114

[9] Amir Pnueli. 1977. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS '77).* IEEE, 46–57.

[10] Abhik Roychoudhury, Corina Păsăreanu, Michael Pradel, and Baishakhi Ray. 2026. Agentic AI Software Engineers: Programming with Trust. *Commun. ACM* 69, 5 (April 2026), 56–58. doi:10.1145/3769314

[11] Kostya Serebryany. 2017. OSS-Fuzz – Google's continuous fuzzing service for open source software. USENIX Security Symposium (invited talk); project at https://github.com/google/oss-fuzz.

[12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12).* USENIX Association, USA, 28.

[13] Theorem. 2026. OSS Verify: proofs about OSS-Fuzz fuzz targets. GitHub repository, https://github.com/theorem-labs/oss-verify-proofs. Accessed 2026-10-29.

[14] Haoxin Tu, Huan Zhao, Yahui Song, Mehtab Zafar, Ruijie Meng, and Abhik Roychoudhury. 2026. Agentic Verification of Software Systems. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE157 (June 2026), 24 pages. doi:10.1145/3808164

[15] Moshe Y Vardi and Pierre Wolper. 1986. An automata-theoretic approach to automatic program verification. In *1st Symposium in Logic in Computer Science (LICS).* IEEE Computer Society.

[16] Andrew Waterman and Krste Asanović. 2019. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA. https://riscv.org/technical/specifications/.

[17] Huan Zhao, Haoxin Tu, Zhengyao Liu, Martin Rinard, and Abhik Roychoudhury. 2026. Automated Lemma Discovery in Agentic Program Verification. In *41st IEEE/ACM International Conference on Automated Software Engineering (ASE).*