QuickProof: From Executable C Properties to Machine-Checked Proofs

GenProof

https://github.com/genproof/quickproof-public

Abstract. Proving that C code is correct has meant writing both the specification and the proof in a proof assistant, in languages that most C developers do not know. AI agents can now write the proofs. QuickProof lets developers write the specification in C: a small program that calls their library on arbitrary inputs and checks the results with assertions, much like a property-based test. QuickProof turns it into a Rocq theorem, AI agents prove it, and QuickProof accepts a proof only if it proves exactly that theorem, so nothing in the proof needs to be trusted. We show how to use QuickProof and what its proofs guarantee, and demonstrate it on zlib and on WireGuard in Linux, where an unprovable property revealed a kernel bug. Work with QuickProof has also found bugs in lz4, xz, zstd, OpenSSL, curl, Redis and other projects. Some of these bugs have since been fixed upstream.

1 Introduction

A proof establishes a property of a C library for every input, where a test checks only the inputs it tries. The Verified Software Toolchain (VST) [1,10] proves such properties in the Rocq proof assistant, and its program logic is itself proved sound in Rocq with respect to the C semantics of the CompCert compiler [35,9]. Yet few C libraries get such proofs, because the work asks for two things that most developers lack. The specification has to be written in the language of the proof system, as separation-logic formulas over Rocq functions, which a C developer cannot easily write or check. And the proof is long: the proofs of Sect. 5 are about 100 and 40 times longer than their specifications.

The second obstacle is going away. AI agents already write Rocq proofs about C code [41,45,46,37], and Rocq checks every step. The first obstacle remains, and agents make it more important: Rocq guarantees that the stated theorem is proved, not that it is the intended one. An agent that cannot find a proof may weaken the specification or add an assumption, and a developer who cannot read the specification cannot tell.

QuickProof [20] addresses the first obstacle: the developer writes the specification in C. We call such a specification a *spec*. It is a C program that calls the library, like a test, with three kinds of special functions: `nondet_*` gives an arbitrary input, `qp_assume` restricts the inputs, and `qp_assert` checks a result.[1]

[1] Published specs also use the older spelling `gp_assert` and `gp_assume`, the names of the underlying functions in the Clight and the theorems.

Fig. 1 shows a spec for a compression library in pseudocode, and Appendix A gives a proved example for zlib.

```
int main(void) {
    uint8_t data[4096], packed[8192], unpacked[4096];
    size_t n = nondet_size();       // any length ...
    qp_assume(n <= sizeof data);    // ... up to 4 KiB
    nondet_bytes(data, n);          // any contents
    size_t m = compress(data, n, packed);
    size_t k = decompress(packed, m, unpacked);
    // decompressing gives back the original data
    qp_assert(k == n && memcmp(unpacked, data, n) == 0);
    return 0;
}
```

Fig. 1. A spec: for any data of up to 4 KiB, decompressing the compressed data gives the data back.

From this spec, QuickProof generates the Rocq theorem `roundtrip : harness_safe whole_prog` without any input from humans or agents. Here `whole_prog` is the spec linked with the library, and `harness_safe` is the safety property of Sect. 4.1. Proved, the theorem means: for every input of up to 4 KiB, `decompress` gives back exactly the data that `compress` was given, and neither function reads or writes out of bounds (`compress` never writes past the 8 KiB of `packed`) or has any other undefined behaviour. The design of QuickProof rests on four points:

- **Specifications in C.** Humans state properties as C programs and review only C code. They need not read Rocq or separation logic.
- **The same statement for every property.** Every theorem says that the program made of the spec and the library has no undefined behaviour. Only the spec changes from property to property; nothing in the proof is part of the statement.
- **Untrusted proofs.** `qp verify` accepts a proof only if Rocq checks it, its statement is the one generated before the proof work began, and it uses only allowed axioms.
- **A small trusted base.** Besides Rocq, VST, CompCert's semantics of C and `clightgen`, the result depends on the spec and on a small part of QuickProof that does not change from project to project (Table 2).

Sect. 2 shows how to write specs, Sect. 3 how to use QuickProof, Sect. 4 its design, guarantees and trusted base, and Sect. 5 proofs of zlib and WireGuard, where a false property showed a kernel bug.

2 Writing Properties in C

A spec is a C file with a `main` function. It includes `quickproof.h`, which declares the functions of its *environment*. `nondet_u8()`, `nondet_u32()` and `nondet_size()` return a value of their type, and `nondet_bytes(buf, n)` fills a buffer. In the theorem they return *any* value, and in test runs pseudo-random ones. `qp_assume(c)` restricts the runs to those where `c` holds, and `qp_assert(c)` states that `c` holds. `malloc` may return `NULL`, as it may in C. A spec reads like a property-based test [13]: `nondet_*` are the generators, `qp_assume` filters the inputs, and `qp_assert` is the check. But a test checks the inputs it tries, and a proof covers every input the spec allows.

Any property that a C program can check about runs of the library can be a spec. The spec of Fig. 1 is a round trip. A spec can also compare a result with an *oracle model*: executable C code that computes the expected result, such as a reference implementation or a protocol model (Sect. 5). Other specs check that a state after a reset equals a fresh one, or that a result does not depend on how the input is split.

A spec may call the library many times, in a loop or in any order the API allows, so it can state properties of state machines and protocols. Memory safety comes with every property, since the theorem rules out undefined behaviour in the library as well as failed assertions.

A proof answers only the question the spec asks. The spec of Fig. 1 says nothing about inputs longer than 4 KiB, and nothing about how well the data compresses: a `compress` that copies its input satisfies it. So humans review every spec, whoever (or whatever) wrote it. The QuickProof documentation gives rules for strong specs, for example: cover every entry point and every argument a client may pass, and make sure that every check is reachable. Developers with fuzz targets can start from them. `qp import-fuzz` turns a libFuzzer target into a spec whose property is that no input of any length makes the target crash. The import is deterministic and uses no LLM. Checks of the results then strengthen the spec.

3 Using QuickProof

Fig. 2 shows the steps of a proof. A project pins the library at an upstream commit. Idioms that CompCert's semantics leaves undefined, or that VST cannot handle, such as `goto`, are patched. Each patch is tested against the original code and, for high assurance, reviewed by a human. Calls to operating-system or kernel interfaces, such as locks or clocks, go to *environment models*, small C versions of these interfaces, which humans review like specs. A project keeps its patches in `patch/` and lists them, with its models, in a file of deviations.

Assemble. `qp assemble` translates the C files to Clight, the C dialect of CompCert and VST, generates one theorem per spec, and commits the code, the configuration and the unproved theorems under the tag `contract`.

Prove. AI agents write the function specifications, loop invariants and proofs, none of which is part of the statement (Sect. 4.1). They may change only proof

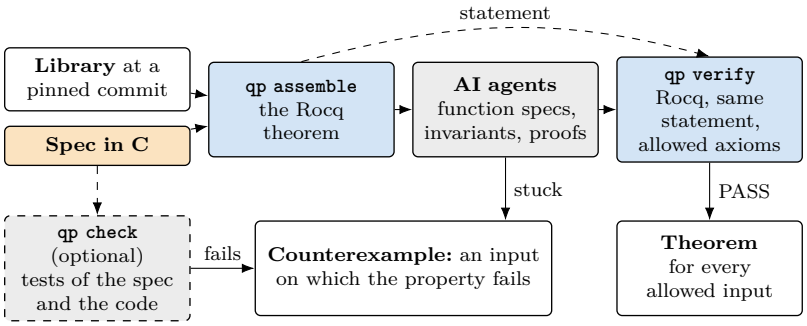


Fig. 2. The steps of a proof with QuickProof. Orange: written or reviewed by humans. Blue: trusted parts of QuickProof, which build on Rocq, CompCert and VST. Grey: untrusted. The statement is fixed before the agents start, and `qp verify` checks that the proved statement is the same. Testing with `qp check` is optional.

files. When a proof gets stuck because the code violates the property, the agent looks for an input that shows it and reports a counterexample: a bug in the library, a gap in its documentation, or a mistake in the spec.

Verify. In a fresh clone, `qp verify` checks that Rocq accepts the proof, that its statement is the one committed under `contract`, and that it uses only allowed axioms (Sect. 4.4).

Check (optional). A spec is a C program, so it can be run before any proof work, to test the spec and the code. `qp check` runs it natively under sanitizers on random inputs and under CompCert's interpreter, which executes C according to the semantics the theorem refers to. It also runs it against *mutants* of the library, in which a comparison or an operator is changed or a constant moved by one. This is cheap: many false properties fail in seconds, and a spec that notices no mutant is too weak to be worth a proof.

4 Design and Guarantees

4.1 The Theorem

`qp assemble` translates every C file with CompCert's `clightgen` and links the spec with the library and with QuickProof's own C versions of `memcpy`, `memset` and a few other C library functions, which are verified together with the library. The result is one Clight program, `whole_prog`. For each spec, it generates

```
Theorem law : harness_safe whole_prog.
```

`harness_safe`, defined in QuickProof's `Contract.v`, is VST's whole-program judgement `semax_prog` for `whole_prog`, with the specifications of the environment (Table 1) for the external functions. `semax_prog` fixes the precondition of `main`, which describes the initial values of the global variables,

Function	Precondition	Postcondition	Met by
`nondet_u32()`	–	any `uint32_t`	any function
`nondet_bytes(b, n)`	`b[0..n)` writable	`b[0..n)` arbitrary	any function that writes `b[0..n)`
`qp_assume(c)`	–	$c \neq 0$	`if (!c) for (;;);`
`qp_assert(c)`	$c \neq 0$	–	`if (!c) *(int *)0 = 0;`
`malloc`, `free`	VST's standard specifications; `malloc` may return `NULL`		any allocator

Table 1. The environment of a spec (`quickproof.h`, `Env.v`): the VST specification of each function, and a C function that meets it (simplified from the driver for CompCert's interpreter).

and leaves its postcondition free. The specifications of the library's functions appear only under an existential quantifier. In Hoare-logic terms, the theorem is {init} `main` {⊤}. It is proved in VST's separation logic from function specifications that the agents choose, and the proofs of the functions are composed with VST's verified software units [7]. VST's judgements rule out undefined behaviour, so the theorem says that no run of `whole_prog` has undefined behaviour (with the caveat of Sect. 4.2), and its statement does not depend on anything in the proof.

The environment turns every property into this one statement. A `nondet_*` function has a specification that says nothing about its result but its type, so the proof must hold for every value it may return: this is nondeterministic choice. The precondition of `qp_assert(c)` is $c \neq 0$, so each call is an obligation that the proof must discharge. The postcondition of `qp_assume(c)` is $c \neq 0$, so after the call the proof may use it.

Each of these specifications is met by a C function (Table 1). `qp_assert` stores through a null pointer when the condition is false, so a failed assertion *is* undefined behaviour, and a property written as assertions reduces to the absence of undefined behaviour. `qp_assume` loops forever when the condition is false. VST proves partial correctness, in which a function that never returns meets every postcondition, so runs in which an assumption fails need no proof. Under CompCert's interpreter, `qp check` runs the spec with exactly these functions, so a failed assertion there is a counterexample to the theorem itself. Native runs discard an input that fails an assumption rather than loop. The environment's definitions contain no axioms.

4.2 Guarantees

The Rocq theorem is a VST judgement. VST's adequacy theorem turns such a judgement into a statement about runs, but its hypotheses about the external functions are not yet proved in Rocq for the QuickProof environment (Sect. 7). With this caveat, the theorem states one thing: no run of `main`, the spec linked

with the library, has undefined behaviour. Since `nondet_*` may return any value, and the only runs cut off are those in which a `qp_assume` fails, this holds for every input the spec allows. In particular,

- every `qp_assert` of the spec holds, since a failed one is undefined behaviour (Sect. 4.1);
- the spec and the library are memory safe: no access outside an object, no use after `free`, no double `free` and no null-pointer dereference, each of which makes a run undefined in CompCert's semantics, and no read of an uninitialized value, which VST's checks rule out;
- there is no division by zero and no shift by a negative amount or by the width of the type or more, both undefined in CompCert, and no signed overflow in `+`, `-` and `*`, which CompCert defines (the result wraps around) but VST's checks rule out.

The guarantee is about the C source as CompCert defines C on 64-bit x86 Linux, with VST's checks, which is stricter than ISO C in some places and more permissive in others. For example, CompCert defines computing a pointer outside an object (but not using it) and reading an object through a pointer of another type, and the theorem does not rule these out. The theorem is about partial correctness of one thread. It does not cover termination, running out of stack, the compiled binary or the system's C library, and it rules out memory leaks only if the spec counts allocations. Assembly, SIMD intrinsics and most floating-point code are outside what QuickProof can handle today.

4.3 Trusted Base

Table 2 lists the parts of a proof. Nothing in the proof is trusted. The trusted parts fall into three groups. The first is the property: the spec, the models and the patches, which humans read and `qp check` tests. This is the only part that changes from project to project, and the part that QuickProof makes readable. The second is a fixed part of QuickProof: the environment of Table 1, the translation and `qp verify`. The translation runs `clightgen` on each C file with fixed options. Around that call, it checks that the build configuration is plain data, renames clashing static names and generates `whole_prog` and the theorems. The third is the foundations. Rocq's kernel checks the proofs, including VST's derived rules and its soundness proof with respect to CompCert's Clight [1,9], so these need not be trusted. Trusted are VST's definitions of the judgement and of linking, which must express the property of Sect. 4.1, and the axioms listed in Sect. 4.4. CompCert's semantics defines what undefined behaviour is, and `clightgen` parses and elaborates C without a formal proof.

Finally, QuickProof links each spec with only the C files it needs. A file left out by mistake cannot make a false theorem provable, since its functions become external functions that a proved program never calls. That the smaller program has undefined behaviour exactly when the full one has is argued on paper, not yet in Rocq.

Part	Checked by
Trusted: the property, written or reviewed by humans	
The specs, their assumptions, the build configuration	humans; tests
Models of operating-system and kernel interfaces	humans; tests
Patches of idioms that CompCert or VST cannot take	humans; differential tests
Trusted: QuickProof	
The environment: `quickproof.h`, `Env.v`, `Contract.v` (194 lines)	the authors
The translation scripts around `clightgen` (about 900 lines)	the authors; reruns
The acceptance check, `qp verify`	the authors; tests
Trusted: foundations	
Rocq's kernel; CompCert's semantics of Clight; `clightgen`	their authors
VST's judgement, linking and adequacy statement	VST's authors
Allowed axioms, including VST's allocation assumptions	their authors
Not trusted	
VST's derived rules, tactics and soundness proofs	Rocq
Function specifications, loop invariants, lemmas, proofs (agents)	Rocq
QuickProof's C library functions (`memcpy`, `memset`, ...)	Rocq

Table 2. The parts of a QuickProof proof

4.4 Proof Acceptance

`qp verify` treats the proof repository as untrusted. In a fresh clone it checks that only proof files changed since the commit tagged `contract`, that the Clight generated again equals the committed Clight, and that the framework files that define the theorems match its own copy. It builds the proof and compares two things with what the trusted generator produces from the spec: each statement, as Rocq elaborates it, without notations or implicit arguments, and the hash of its `whole_prog`. `Print Assumptions` must list only entries in `ALLOWED_AXIOMS`, a fixed list in the trusted verifier (`tools/qp`) that proof repositories cannot extend. It contains standard logical axioms such as excluded middle, extensionality and proof irrelevance, two axioms of Rocq's classical reals, and the declarations and axioms of VST's abstract allocation library.

The proof's own sources may contain no `Admitted`, `admit`, `Axiom`, `Parameter` or `Conjecture`, even in a lemma that no theorem uses; in a vendored third-party library such places are only listed, and `Print Assumptions` decides whether a theorem depends on them. Without a contract, the result is UNVERIFIED, never a pass.

Threat model. `qp verify` runs no script from the repository. It reads the build configuration `quickproof.conf` only after checking that it holds nothing but variable assignments, since bash sources it and a command in it could modify framework files after they have been compared. It refuses Rocq commands and

Library	Properties	Spec and model lines	Rocq lines
zlib 1.3.2, core library	62	5,500	590,000
WireGuard in Linux	38	7,500	300,000

Table 3. Published proofs (approximate). Spec and model lines include the specs, their oracle models (zlib's reference encoder, WireGuard's protocol model and reference cryptography) and the environment models of kernel interfaces, but not patches or configuration.

options that load OCaml plugins (`Declare ML Module`) or turn off checks of the kernel (`Unset Guard Checking`, `-type-in-type`). Since a repository could define its own `harness_safe` that prints like the real one, it checks the type of each theorem with an explicit ascription to the definition in its own copy of the framework. Its end-to-end tests check each of these refusals, an unused `Admitted`, and a repository without a contract.

5 Case Studies

Table 3 summarizes the two case studies of this report. Two more proofs, of OpenSSL's punycode decoder [19] and of zstd's decoder [23], are published in the same form.

For zlib 1.3.2, at upstream commit `da607da7`[2], we proved 62 specs of the core library: compression, decompression and the checksums [22]. Forty-five of them are short, 14 to 204 lines of C, and most bound neither the data nor the number of calls. The short ones fall into six groups by what they promise:

- **Round trips (7 specs).** What goes in comes back: any data at any level, through the one-shot API (Appendix A) or through streams with the data in any pieces and any flush up to `Z_BLOCK`, with a preset dictionary or a gzip header.
- **Rejecting bad input (12).** Bad parameters and bad streams are refused, and any input is safe: `inflate` on any bytes, in any pieces, returns only documented codes.
- **The standard on valid input (4).** A stream of stored blocks built by hand from the format specifications, RFC 1950 (zlib), RFC 1951 (DEFLATE) and RFC 1952 (gzip), decompresses to its data, and compression at level 0, which stores the data uncompressed, writes exactly the stored blocks that RFC 1951 defines.
- **The stream interface (10).** The pointers and counters of a `z_stream` (`next_*`, `avail_*`, `total_*`) move by exactly what a call read and wrote, unwritten output stays as it was, and the same calls give the same results.

[2] With nine bug fixes and the idiom patches that `patch/PATCH.md` of the proof repository describes.

- **Sizes and memory (6).** The documented size bounds hold, the memory that compression and decompression allocate stays within exact bounds, and `deflateEnd` and `inflateEnd` free all of it.
- **Checksums (6).** Adler-32 and CRC-32 equal the definitions of RFC 1950 and RFC 1952, written out in the spec, and combining them works for any cut.

The other 17 specs come from an earlier proof of zlib's core, with their statements unchanged. They are bounded, most to 1 MiB of data or 16 MiB of input, and one of them checks `inflate` against a reference encoder written from RFC 1951 for all three kinds of block. The `gz*` file API and `infback.c` are not covered.

We also proved 38 specs of WireGuard in Linux at commit `551c722f`[3]: the replay window and the handshake checks of `receive.c`, the Noise_IKpsk2 handshake with its cookies (`noise.c`, `cookie.c`), and the generic C of the primitives these call, BLAKE2s, ChaCha20, Poly1305, ChaCha20-Poly1305, XChaCha20-Poly1305 and Curve25519 [21]. Nineteen short specs each state one property in a few dozen lines of C, with no model of the protocol:

- **The replay window (5 specs).** A nonce is a 64-bit counter, and `REJECT_AFTER_MESSAGES` is its limit. `counter_validate` accepts a new nonce that is below the limit and at most the window size less than the newest accepted one, whatever nonces came before. It rejects a nonce it has accepted, a nonce more than the window size less than the newest and every nonce from the limit on.
- **Authenticated encryption (6).** ChaCha20-Poly1305 and XChaCha20-Poly1305 give back the message when they decrypt what they encrypted, for any key and nonce, any message of up to `PTRDIFF_MAX - 16` bytes and any associated data, and a decryption that fails leaves its output buffer as it was. BLAKE2s, ChaCha20 and Poly1305 give the same result over a message in one call or in pieces.
- **The handshake (8).** Between two devices with any keys, clocks and randomness, when every step succeeds on both sides, each side's sending key is the other's receiving key. A session that begins erases the handshake's ephemeral keys and chaining key, and a session does not begin unless the handshake is in a state that allows it. A replayed initiation is rejected, and a rejected initiation or response leaves the receiver's state for the sender as it was.

The other 19 specs, prefixed `x_`, are longer. They compare the handshake code with a model of the handshake of the WireGuard paper, written as plain C over the kernel's primitives, and the primitives with reference implementations written from RFC 7693, RFC 8439 and the XChaCha draft. They cover the handshake

[3] With our two patches to `lib/crypto` (*Bugs found* below), idioms rewritten for VST and a shim of plain C for the kernel around the code, all listed in the repository's `DEVIATIONS.md`.

functions from any prior state, a full handshake, replayed initiations, keypair rotation and reference counts, the configuration functions, the MACs and cookie replies, the handshake checks of `receive.c`, the replay window, the length check of data packets, the data path's order of tag check and replay window (assumed), the sending nonce (a model of `send.c`), and the primitives for any length up to `PTRDIFF_MAX`. Several proofs use the Diffie–Hellman agreement of X25519, which is proved from the verified Montgomery ladder of Fiat-Crypto [17].

The complete receive and send paths, the rest of the driver, runs of more than one thread, and the variants of the primitives that x86-64 kernels build instead of the generic C are not covered. WireGuard's protocol itself has a mechanized cryptographic proof [36], and Linux's generic Curve25519 comes from verified HACL* [47] and Fiat-Crypto [17] code. Our properties say what the kernel's C code of the handshake and of the primitives above computes, not whether the protocol is secure.

Agents based on Claude Opus 5.5 in Claude Code wrote the proofs. Human effort went mostly into choosing the properties and reviewing the specs. The specs and models, the part that humans read, are about 1% of the proof's size for zlib and 2.5% for WireGuard. For comparison, the VST proof of OpenSSL's SHA-256, 235 lines of C, is 6,539 lines of Coq written by hand [2]. The case studies used Rocq 8.20.1 (released as Coq), VST 2.15 and CompCert 3.15. The framework and the proofs are available; the agents are not.

Bugs found. The WireGuard properties for Poly1305 and ChaCha20-Poly1305 were false for lengths of 4 GiB or more: `poly1305_update()` added two lengths as `unsigned int`, so the sum could wrap and the input could be copied past a 16-byte buffer. While writing the spec, we also found, by reading the code, that `chacha20poly1305_encrypt` and `_decrypt` truncate the length to 32 bits, so data beyond 4 GiB is neither encrypted nor authenticated. No kernel caller passes such lengths. We reported both bugs with patches [18], and the fixes were applied in Eric Biggers' lib/crypto series [8], which credits our report. The WireGuard proofs are about the code with our patches; the applied series has not been verified. The zlib property that `compress2` into `compressBound` bytes succeeds, part of the round trip of Appendix A, was false too, at level 0 for some inputs over 51 TiB. The appendix describes the bug and its fix [48].

Beyond the case studies, work with QuickProof has turned up many bugs in other projects, among them lz4, xz, zstd, OpenSSL, curl, Redis, nanopb, nginx, c-ares, musl, libfdt, flac and libjpeg-turbo, and more bugs in zlib and in the Linux kernel (lib/crypto and WireGuard). Most of them are bugs on rare paths or undefined behaviour in ISO C with no observed effect. A few let attacker-controlled input to a widely used API corrupt memory or data. Of those we reported, fixes have landed upstream in zlib, lz4, Linux lib/crypto, Redis, curl and OpenSSL. Other reports await a reply or a fix.

Testing. In a separate experiment, we measured how often the checks of Sect. 3 catch a false property before any proof work. On 30 false properties, the native runs of `qp check` found the violation within 10 seconds in 49 of 50 trials on

existing test fixtures and in 173 of 200 on injected faults, but never on five cases that fail only on rare inputs; those are left to the proof.

6 Related Work

The work closest to QuickProof is by Anthropic and Theorem [37]. There, Claude proved in Rocq, for 239 entry points of 123 OSS-Fuzz libraries, that the program exits without memory errors or failed assertions on any input smaller than 64 MB. The statement is derived from the existing fuzzing harness and is about the compiled RISC-V code relative to a model of a Linux process, so the compiler need not be trusted and no human writes a specification. This setup gives scale, but it has its limitations.[4]

- A statement compares the plain build with a build under a sanitizer, so a memory error is what AddressSanitizer detects and, as the authors note, not every memory error. For example, an access may jump over a red zone into another live object.
- A result is checked only where the harness itself checks it: in 6 of the 61 entry points, by the documentation's own list.
- Whether allocation may fail is chosen by hand for each harness, and nothing checks that the choice matches the harness. Most statements let the environment refuse memory only when it runs out, since a harness that aborts on a failed allocation would otherwise make the statement false (`docs/statements.md` in [43]).
- The guarantee is about one binary, not the source code or another build.
- The proofs share only the machine model and the statement forms. Every lemma, down to bzip2's Hoare rules and its lemmas about `memcpy`, belongs to one harness.

QuickProof makes different choices. The property is stated in the spec. A memory error is defined by Clight's semantics, in which each object is a separate memory block, so an access outside an object is undefined behaviour whether or not a sanitizer notices it. There is one environment for all specs, in which `malloc` may always return `NULL`. Function proofs in VST's separation logic are shared by many properties. These choices have limitations of their own: `clightgen` and CompCert's semantics must be trusted, and the guarantee is about the source code, not the binary (Sect. 4.3).

The spec itself is the oldest part of this design. It took shape as a *harness* in software model checking in the late 1990s: VeriSoft [24] modelled the environment of a C program by nondeterministic choice in the code and stated the properties as assertions, and Holzmann and Smith [29] surrounded an extracted model with test drivers and a list of properties. CBMC [14] also checks C programs with nondeterministic inputs and assertions. In verification languages such as Boogie [34], the same roles are played by `havoc`, `assume` and `assert`. Parameterized

[4] We read them off the published subset of the proofs [43], 61 entry points of 26 libraries, and its documentation.

unit tests [44] treated a test with free inputs as a specification, and one harness served both testing and model checking or symbolic execution in PROMELA [27], DeepState [25] and Kani [31]. Property-based testing [13] entered proof assistants to test conjectures before proving them [16,6]. A spec that runs a function twice, for example on a whole input and on its parts, is a self-composition [5] or product program [4].

CBMC proves contracts [33,11] and is used at scale in industry [12], SeaHorn infers invariants [28], Frama-C/WP proves ACSL contracts that E-ACSL runs as checks [32,42], and Fulminate runs CN's separation-logic specifications of C [40,3]. These tools can take the same kind of spec as QuickProof but differ in what discharges it. CBMC unrolls the loops up to a bound and needs nothing but the spec, but a loop over an input of up to `PTRDIFF_MAX` bytes, as in the cryptographic properties of Sect. 5, cannot be unrolled. For such loops, CBMC with contracts, Frama-C/WP and CN need loop invariants and function contracts, which humans write in the tool's language, and a solver discharges what follows from them. SeaHorn leaves the invariants to a Horn-clause solver and to abstract interpretation. KestRel [15] uses SeaHorn to verify product programs and reports cases where the invariants were out of SeaHorn's reach and the authors proved the programs by hand in VST. In QuickProof, agents do this work for every property (Sect. 4.1). We have not compared the tools on the case studies of Sect. 5.

CBMC, SeaHorn, Frama-C/WP and CN rely on trusted encodings, solvers or verification-condition generators. In QuickProof, the proofs are checked by Rocq, in a program logic proved sound for CompCert's Clight. As in proof-carrying code [39], untrusted proof production is separated from checking against a fixed policy; in QuickProof, the policy is generated from the spec.

LLM agents already build proofs about C. SynVer synthesizes C programs with VST proofs [38], AutoRocq [45] and LemmaNet [46] discharge WP-generated obligations in Rocq, and Kan et al. [30] require every obligation handed to a code agent to be closed by a kernel-checked proof. Our earlier work on liblzma [41] proved VST theorems about individual functions, with function specifications approved by humans. Gopinathan et al. [26] also observe that with agents the bottleneck moves from proof to specification. There, the agent writes the semantics and the existing implementation is the ground truth, while in QuickProof humans write the spec and the proof checks the code.

7 Limitations and Future Work

Two arguments remain to be formalized in Rocq. The first is the step from the VST judgement to a statement about runs (Sect. 4.2): instantiating VST's adequacy theorem for the QuickProof environment requires a concrete allocation model in place of VST's abstract library. The second is that a spec linked with only the C files it needs has undefined behaviour exactly when the full program has (Sect. 4.3). This is proved only on paper. Beyond these, we plan to publish reproducible proof-search protocols and to measure success rates, human effort, variance across

repeated runs, costs against other tools, and the cost of maintaining proofs as the code changes.

References

1. Appel, A.W.: Verified software toolchain. In: ESOP. LNCS, vol. 6602, pp. 1–17 (2011). https://doi.org/10.1007/978-3-642-19718-5_1
2. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Trans. Program. Lang. Syst. **37**(2), 1–31 (2015). https://doi.org/10.1145/2701415
3. Banerjee, R., Memarian, K., Makwana, D., Pulte, C., Krishnaswami, N., Sewell, P.: Fulminate: Testing CN separation-logic specifications in C. Proc. ACM Program. Lang. **9**(POPL), 1260–1292 (2025). https://doi.org/10.1145/3704879
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: FM. pp. 200–214. LNCS (2011). https://doi.org/10.1007/978-3-642-21437-0_17
5. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: CSFW. pp. 100–114 (2004). https://doi.org/10.1109/CSFW.2004.1310735
6. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: SEFM. pp. 230–239 (2004). https://doi.org/10.1109/SEFM.2004.1347524
7. Beringer, L.: Verified software units. In: ESOP. pp. 118–147. LNCS (2021). https://doi.org/10.1007/978-3-030-72019-3_5
8. Biggers, E.: lib/crypto: Use size_t lengths for Poly1305 and ChaCha (2026), https://lore.kernel.org/all/20261014152604.323310-1-ebiggers@kernel.org/, patch series on linux-crypto, 14 October 2026
9. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. J. Autom. Reason. **43**(3), 263–288 (2009). https://doi.org/10.1007/s10817-009-9148-3
10. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-Floyd: A separation logic tool to verify correctness of C programs. J. Autom. Reason. **61**(1–4), 367–422 (2018). https://doi.org/10.1007/s10817-018-9457-5
11. CBMC: Code contracts user documentation, https://diffblue.github.io/cbmc/contracts-user.html, accessed 20 October 2026
12. Chong, N., Cook, B., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Code-level model checking in the software development workflow. In: ICSE-SEIP. pp. 11–20 (2020). https://doi.org/10.1145/3377813.3381347
13. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP. pp. 268–279 (2000). https://doi.org/10.1145/351240.351266
14. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. pp. 168–176. LNCS (2004). https://doi.org/10.1007/978-3-540-24730-2_15
15. Dickerson, R., Mukherjee, P., Delaware, B.: KestRel: Relational verification using e-graphs for program alignment. Proc. ACM Program. Lang. **9**(OOPSLA1), 118:1–118:28 (2025). https://doi.org/10.1145/3720474
16. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining testing and proving in dependent type theory. In: TPHOLs. pp. 188–203. LNCS (2003). https://doi.org/10.1007/10930755_12
17. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic: With proofs, without compromises. In: IEEE S&P. pp. 1202–1219 (2019). https://doi.org/10.1109/SP.2019.00005

18. GenProof: lib/crypto: Handle large lengths in poly1305 and chacha20poly1305 (2026), https://lore.kernel.org/r/20261012-libcrypto-4gib-lengths-v1-0-ea566c129a2e@genproof.com, patch series on linux-crypto, 12 October 2026
19. GenProof: OpenSSL's punycode decoder: Proved C specs (2026), https://github.com/genproof/openssl-punycode-auto-proofs-public, commit 7556530
20. GenProof: QuickProof: Source, guarantees, and case-study reports (2026), https://github.com/genproof/quickproof-public, v0.1.0, commit fab76ef; README, Guarantees, Scope
21. GenProof: WireGuard in Linux: Proved C specs (2026), https://github.com/genproof/wireguard-auto-proofs-public, commit 3c1fc13
22. GenProof: zlib 1.3.2, the core library: Proved C specs (2026), https://www.genproof.com/quickproof/proofs/zlib, proof repository github.com/genproof/zlib-full-auto-proofs-public, commit 37189f4
23. GenProof: zstd's decoder: Proved C specs (2026), https://github.com/genproof/zstd-auto-proofs-public, commit 2e46017
24. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL. pp. 174–186 (1997). https://doi.org/10.1145/263699.263717
25. Goodman, P., Groce, A.: DeepState: Symbolic unit testing for C and C++. In: Workshop on Binary Analysis Research (BAR) (2018)
26. Gopinathan, K., Pimpalkhare, N., Liu, Y., Tavares, Z.: Ship your interpreter (2026), https://www.basis.ai/blog/ship-your-interpreters/, basis, 16 October 2026
27. Groce, A., Joshi, R.: Random testing and model checking: Building a common framework for nondeterministic exploration. In: WODA. pp. 22–28 (2008). https://doi.org/10.1145/1401827.1401833
28. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: CAV. pp. 343–361. LNCS (2015). https://doi.org/10.1007/978-3-319-21690-4_20
29. Holzmann, G.J., Smith, M.H.: A practical method for verifying event-driven software. In: ICSE. pp. 597–607 (1999). https://doi.org/10.1145/302405.302710
30. Kan, S., Kan, S., Ertel, S.: Harnessing code agents for automatic software verification. arXiv:2607.06341 (2026)
31. Kani Team: From fuzzing to proof: Using Kani with the Bolero property-testing framework. Kani Rust Verifier Blog, October 27 (2022)
32. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
33. Kroening, D., Schrammel, P., Tautschnig, M.: CBMC: The C bounded model checker. arXiv:2302.02384 (2023), book chapter preview
34. Leino, K.R.M.: This is Boogie 2 (2008), manuscript KRML 178
35. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
36. Lipp, B., Blanchet, B., Bhargavan, K.: A mechanised cryptographic proof of the WireGuard virtual private network protocol. In: EuroS&P. pp. 231–246 (2019). https://doi.org/10.1109/EuroSP.2019.00026
37. Lucas, K., Lynch, A., Tong, L., Champine, L., Straznickas, Z., Distefano, D., Agrawal, H., Ben Asher, E., Howell, A., Jawhar, S., Rohlf, C., Zeldovich, N., Carlini, N., Agrawal, R., Gross, J.: LLMs are making formal software verification feasible at a much larger scale (2026), https://www.anthropic.com/research/formal-verification, anthropic, 22 October 2026

38. Mukherjee, P., Lu, M., Delaware, B.: LLM-assisted synthesis of high-assurance C programs. In: ASE (2025). https://doi.org/10.1109/ASE63991.2025.00255
39. Necula, G.C.: Proof-carrying code. In: POPL. pp. 106–119 (1997). https://doi.org/10.1145/263699.263712
40. Pulte, C., Makwana, D.C., Sewell, T., Memarian, K., Sewell, P., Krishnaswami, N.: CN: Verifying systems C code with separation-logic refinement types. Proc. ACM Program. Lang. **7**(POPL), 1–32 (2023). https://doi.org/10.1145/3571194
41. Shlyakhtun, P., Gryzlov, A., Kukharenko, V., Nesterov, V., Vasiliev, N., Ziborov, K., Zolotarev, E., Pokras, A.: Agent-driven verification of memory safety for liblzma decoder components with VST. arXiv:2608.29716 (2026)
42. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a runtime verification tool for safety and security of C programs (tool paper). In: RV-CuBES. Kalpa Publications in Computing, vol. 3 (2017). https://doi.org/10.29007/fpdh
43. Theorem: oss-verify-proofs (2026), https://github.com/theorem-labs/oss-verify-proofs
44. Tillmann, N., Schulte, W.: Parameterized unit tests. In: ESEC/FSE. pp. 253–262 (2005). https://doi.org/10.1145/1081706.1081749
45. Tu, H., Zhao, H., Song, Y., Zafar, M., Meng, R., Roychoudhury, A.: Agentic verification of software systems. Proc. ACM Softw. Eng. **3**(FSE), 3558–3581 (2026). https://doi.org/10.1145/3808164
46. Zhao, H., Tu, H., Liu, Z., Rinard, M.C., Roychoudhury, A.: Automated lemma discovery in agentic program verification. In: ASE. pp. 323–335 (2026). https://doi.org/10.1145/3832783.3834356
47. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A verified modern cryptographic library. In: CCS. pp. 1789–1806 (2017). https://doi.org/10.1145/3133956.3134043
48. zlib issue #1348 (2026), https://github.com/madler/zlib/issues/1348, fixed in commit 2bed3d61

A Case Study: zlib One-Shot Roundtrip

The spec below, `rt_roundtrip` of the 62 in Sect. 5, states a round trip through zlib's one-shot API, `compress2` and `uncompress2`. The example is abridged: the full spec also passes `Z_NULL` for empty buffers. `upto(n)` returns any value from 0 to `n`, `alloc_any_bytes(n)` allocates `n` bytes with arbitrary contents, `copy_of` copies a buffer and `bytes_eq` compares two. The compressed-buffer bound is computed with the exact formula of zlib's `compressBound`.

```
int main(void) {
// 1. Arrange data for compression

  // Original (uncompressed) text's length
  size_t original_len = upto(PTRDIFF_MAX);
  // Size of the buffer for decompression: at least
  // the length of the original text
  size_t decompressed_size =
    original_len + upto(PTRDIFF_MAX - original_len);
  // Compression level: -1 to 9
```

```
  int level = -1 + (int) upto(10);
  // Maximum length of the compressed text, which is
  // length of the original text + overhead.
  // This is the exact formula of zlib's compressBound.
  size_t compressed_bound = original_len + (original_len >> 12) +
    (original_len >> 14) + (original_len >> 25) + 13;
  // Original text
  unsigned char *original = alloc_any_bytes(original_len);
  if (original == NULL)
    return 0;
  // Copy of original text.
  // Buffer to compress is passed by pointer to the
  // compress function and therefore we need to store the
  // original copy for further comparisons.
  unsigned char *original_copy = copy_of(original, original_len);
  // Buffer for compressed text.
  unsigned char *compressed = malloc(compressed_bound);
  // Buffer for uncompressed text (after the roundtrip).
  unsigned char *decompressed = alloc_any_bytes(decompressed_size);
  if (original_copy == NULL || compressed == NULL ||
      decompressed == NULL)
    return 0;
  // Copy of the buffer for uncompressed text, to check
  // that its bytes past the original text stay unchanged.
  unsigned char *decompressed_copy =
    copy_of(decompressed, decompressed_size);
  if (decompressed_copy == NULL)
    return 0;
  uLongf compressed_len = compressed_bound;

// 2. Run the compression

  // Call the compress function
  int code = compress2(compressed, &compressed_len,
                       original, original_len, level);
  // Bail out on Z_MEM_ERROR
  if (code == Z_MEM_ERROR)
    return 0;

// 3. Check the compression result

  // Check the returned value and that bound guarantee held
  qp_assert(code == Z_OK && compressed_len <= compressed_bound);

// 4. Arrange data for the decompression

  // Arbitrary suffix length
  size_t suffix_len = 0;
  if (compressed_len < PTRDIFF_MAX)
    suffix_len = upto(PTRDIFF_MAX - compressed_len);
```

```
  // Decompression input: compression result + arbitrary suffix
  unsigned char *source = alloc_any_bytes(compressed_len + suffix_len);
  if (source == NULL)
    return 0;
  for (size_t i = 0; i < compressed_len; i++)
    source[i] = compressed[i];
  uLongf decompressed_len = decompressed_size;
  uLong source_consumed_len = compressed_len + suffix_len;

// 5. Run the decompression

  // Call the decompress function
  code = uncompress2(decompressed, &decompressed_len,
                     source, &source_consumed_len);
  // Bail out on Z_MEM_ERROR
  if (code == Z_MEM_ERROR)
    return 0;

// 6. Check the decompression result

  // Check the return value, the length of the uncompressed
  // text and the number of source bytes consumed
  qp_assert(code == Z_OK && decompressed_len == original_len &&
            source_consumed_len == compressed_len);
  // Check that we got the original text back
  qp_assert(bytes_eq(decompressed, original_copy, original_len));
  // Check that the rest of the buffer stays unchanged
  qp_assert(bytes_eq(decompressed + original_len,
                     decompressed_copy + original_len,
                     decompressed_size - original_len));

  return 0;
}
```

The proved property is: on 64-bit x86 Linux (LP64), for any data of length up to `PTRDIFF_MAX` and any level from -1 to 9, in runs where zlib does not return `Z_MEM_ERROR`,

1. `compress2` returns `Z_OK`, and the compressed length is at most `compressBound` of the data length;
2. `uncompress2` of the compressed data followed by an arbitrary suffix (up to `PTRDIFF_MAX` bytes in total), into a buffer at least as long as the original data (up to `PTRDIFF_MAX` bytes), returns `Z_OK` and gives back exactly the original data, leaving the rest of the buffer unchanged. It sets `decompressed_len` to the original length and `source_consumed_len`, the number of source bytes it reports consumed, to the compressed length: the suffix is not consumed.

A proof attempt of the size bound, the first `qp_assert` above, showed that it was false before zlib issue #1348 was fixed [48]: `deflate_stored()` added

two 32-bit `unsigned` values, so the sum could wrap on 64-bit platforms too, and level-0 `deflate(Z_FINISH)` could end the stream with about 4 GiB of input unread. Through `compress2` this happens for some inputs over 51 TiB. Mark Adler fixed the bug the same day it was reported, and the proof is about code that includes the fix.

The spec has the arrange–act–assert structure of a unit test, but the property is proved for every input in the stated ranges, not for sampled ones. It uses only zlib's public API (`zlib.h`), so humans review the spec but not zlib's internals, which are left to the proof.

Choosing properties that can be stated through the public API and are still useful and non-trivial is the main work of verification with QuickProof. It takes knowledge of C and of the library rather than of Rocq, so QuickProof lets C programmers apply their expertise to formal verification.