BuildLang: Accountable Compute via Typed Capability Effects and Re-Derivable Receipts

Zain Dana Harper
Seattle, WA | ORCID 0009-0001-7175-5393
https://github.com/HarperZ9/buildlang

July 2026

Abstract

A program's interaction with the outside world—files it reads, sockets it opens, environment it queries—is the natural unit of accountability for AI-assisted and scientific code, yet in most languages it is hidden behind runtime helpers, macros, and foreign calls that never appear in a function's type. BuildLang is a Rust-built compiler for typed algebraic effects in which *ambient capability access must appear in the function type*: reading a file, opening a socket, or calling foreign code each requires a declared capability effect (FileSystem, Network, Foreign, and five others), tracked conservatively through callbacks, closures, struct fields, async blocks, and control-flow selectors so that policy cannot be bypassed by hiding an effect behind an untyped function value. We use this effect system as a *witness mechanism* for two families of re-derivable receipts: a codegen receipt that seals the source and the declared/observed capability facts, and a scientific-runtime receipt that compiles and runs a numeric kernel, checks a stated conservation or boundedness invariant over the captured series, and is verified by *re-running the program*. The scientific receipt ships seven mathematically distinct invariants (energy-monotone, conservation, the discrete maximum principle, an energy-balance residual, cross-column relation, approximate symplectic conservation, and a non-negative complexity bound), each with a paired negative-fixture kernel that demonstrates the checker rejects a crafted violation (a bound on false negatives for those cases, not a completeness proof), and a mandatory overclaim boundary (Not_a_new_physical_law) so a receipt cannot be mistaken for a physical result it does not assert. We report 940 library tests passing with none failing, and a corpus of positive/negative kernel pairs across physics, quantum, chemistry, and algorithmic-complexity domains. The artifact is public; the receipts are designed so that an independent replayer re-derives the verdict without trusting stored bytes.

1 Introduction

The boundary a program crosses when it touches the world is where accountability lives: a network call exfiltrates data, a file read pulls an unlogged dataset, a foreign call hides arbitrary behavior behind an untyped symbol. Mainstream practice treats these as implementation details. The function signature says nothing about whether the body reads a file; the compiler does not track it; and a reviewer or a downstream policy gate has to recover the fact by reading the whole transitive body. The result is that the most security-relevant fact about a piece of code—what it is actually allowed to touch—is the one fact not in its type.

BuildLang inverts this. It is a statically-typed language with Hindley–Milner inference and *typed algebraic effects*, in which the ambient capabilities a function may exercise are part of its type and are checked by the compiler. A function that calls `read_file` must declare the FileSystem effect; one that opens a socket must declare Network; one that calls into a C function must declare Foreign. The same rule applies to compile-time macros (`include_str!`,

`env!`, `println!`) and to known C-runtime helper aliases, so ambient access cannot hide behind a macro expansion or a renamed helper. Crucially, the capability provenance is tracked *through* the language's value channels: an effectful callback passed into a wrapper, stored in a struct field, selected by an `if`/`match`, captured by a closure, or awaited as a future all retain their declared capability row, so a policy gate sees which boundary function a higher-level workflow actually depends on.

The effect system is not the deliverable; it is the *witness mechanism* for two receipt families that are. A *codegen receipt* seals the entry source, the resolved module graph, and the declared/observed/propagated capability facts into a deterministic artifact a CI gate can recheck. A *scientific-runtime receipt* compiles and runs a numeric kernel, captures its numeric output as a measurement series, checks a stated invariant—energy is non-increasing, a quantity is conserved, a residual stays at roundoff, a complexity slack stays non-negative—and seals the verdict; verification *re-runs the program* and re-checks the verdict, deliberately tolerating platform float differences by comparing the verdict rather than exact bits.

The paper makes four contributions:

1. **Capability effects with provenance through values** (Section 3): eight ambient capabilities surfaced in function types, tracked conservatively through the value channels where effects are usually lost (callbacks, closures, fields, selectors, futures).
2. **Accountable codegen receipts** (Section 4): a deterministic receipt and a policy-profile language that turns observed capability evidence into an enforceable CI gate.
3. **Accountable scientific compute** (Section 5): seven invariants over a captured numeric series, each with a paired negative fixture, verified by re-execution, with effect-derived dataset/determinism fields and a mandatory overclaim boundary.
4. **An honesty discipline** (Section 6): every receipt states what it does *not* claim, in machine-readable form; the verifier is shown to be able to fail via negative fixtures and a self-test.

BuildLang is the systems instance of a broader idea pursued in a companion artifact, EMET [1]: witnessed, re-derivable receipts with closed verdict lattices and mandatory failure-proofs. EMET applies the idea to byte integrity; BuildLang applies it to codegen facts and numeric invariants, with the typed-effect system as the witness that supplies the facts.

2 Background and related work

Algebraic effects. Algebraic effects and handlers [2, 3] separate the *performance* of an effect from its *interpretation*, giving compile-time effect tracking with caller-controlled handlers and no runtime overhead beyond a continuation jump. BuildLang adopts this model (effects declared with `~`, handled with `handle`/`with`) and compiles effects to `setjmp`/`longjmp` on its C backend. Its departure from the effects-language line is the treatment of *ambient* capabilities as first-class effect requirements rather than user-defined operations, and the use of the effect row as sealed evidence rather than only a type-system guarantee.

Capability security. Object-capability languages restrict authority to what is reachable; BuildLang's capability effects are the static analogue, lifted into the type so that capability use is visible at every call site rather than inferred from reachability. The closest contrast is with languages that track effects but do not bind them to enforceable receipts, and with security linters that recover capability use by post-hoc analysis rather than by construction.

Reproducible and verified scientific compute. Conserved-quantity and boundedness checks are standard diagnostics in numerical analysis (discrete maximum principles, energy estimates for parabolic schemes, symplectic integrators' near-conservation of energy). BuildLang's scientific receipt is not a new numerical method; it is an *accountability wrapper* that seals the fact

that a compiled kernel's observed series satisfies a stated such property, verified by re-running the program, with negative fixtures proving the check is not vacuous. The funnel-hashing probe bound we reuse as an algorithmic-complexity example is from prior work [4].

Reproducible builds. The codegen receipt shares the reproducible-builds [5] instinct—seal the source, re-derive the artifact—but seals the *capability facts and verdict* rather than requiring byte-identical artifacts, an explicit non-goal we defend in Section 5.

3 Capability effects

3.1 Eight ambient capabilities

A function that exercises an ambient capability must declare it. The eight built-in capabilities and their direct surfaces are:

Capability	Direct surfaces
FileSystem	`read_file`, `write_file`, `file_exists`, `list_dir`, compile-time `include!`/`include_str!`/`include_bytes!`
Network	`tcp_connect`, `tcp_send`, `tcp_recv`, `tcp_close`
Process	`exit`, `process_exit`
Environment	`getenv`, `args_get`, compile-time `env!`/`option_env!`
Clock	`clock_ms`, `time_unix`
Console	`read_line`, `println!`/`print!`/`eprintln!`, diagnostic logging
Foreign	calls to unknown functions in `extern` blocks; reads from foreign statics
Gpu	direct `build_vk_*` and `build_gfx_*` runtime helpers

Known C-runtime helper aliases declared through `extern` blocks are classified by their specific capability (e.g. `build_read_file` is FileSystem, `build_gfx_init` is Gpu); only genuinely unknown extern functions remain Foreign. Compile-time macros are direct capability surfaces: `println!(read_file("ops.toml"))` requires both Console and FileSystem, and the receipt records both. The compiler rejects a function that performs a capability without declaring it, with a diagnostic naming the ambient call or macro.

3.2 Provenance through value channels

The hard part is not declaring effects on direct calls; it is retaining capability provenance through the value channels where effects are conventionally lost. BuildLang tracks, conservatively, the following escape classes, each backed by regression tests:

- **First-class function values.** An effectful callback type (`fn() with FileSystem`) carries its effect row; a pure `fn()` parameter cannot accept `read_file`, and a cast to a pure callback type is rejected. Aliases (`let loader = read_file`) and closures inherit the capability at the call site.
- **Aggregate storage.** Effectful callbacks stored in struct fields, tuple slots, enum variants (including struct-like variants), and indexed tables retain their origins; calls through `ops.loader`, `loaders.0`, `loaders[0]` record the stored callee as propagated evidence.
- **Control-flow selection.** A function selected by `if`, `if let`, or `match` records every possible branch target; conditional and loop assignment merge the possible exits rather than keeping only the last path visited.
- **Async.** An async block is a delayed effect value; construction is pure, and `await` inherits the body's capability row, recording both the awaited expression and the latent origin.

- **Destructuring and update.** Tuple, struct, enum-variant, and slice destructuring, and struct-update expressions, refresh access paths without forcing policies to allow stale construction aliases.

The unifying rule is that *operational access has to appear in the function type instead of hiding behind a runtime helper, macro expansion, dynamic dispatch, or an untyped function value.* Trait-object method calls are checked from the trait signature, so `loader.load()` through `dyn Loader` records `Loader.load` rather than disappearing behind dispatch.

4 Accountable codegen receipts

`buildc check <file> --receipt <path>` writes a deterministic `buildlang-check-receipt/v1` JSON artifact: compiler/language version metadata, a SHA-256 digest of the entry source bytes, an `input_digests` ledger for every entry/import/include/module file read by the check pipeline, an `input_graph_digest` fingerprint for the whole checked graph, the declared effects, the observed capability sources (`observed_capabilities`), the effectful callees a caller inherits (`propagated_effects`), pass/fail status, and compact diagnostics.

`buildc receipt verify` re-checks a saved receipt against the current source graph: it verifies the receipt schema and compiler/language identity, re-derives the entry source and input-graph digests, replays the compiler check, and compares the saved `declared_effects`, `observed_capabilities`, `propagated_effects`, diagnostics, and policy violations against the current result. Verification can be pinned to a built-in profile (`--expect-profile ci-review`) or to an exact policy digest (`--expect-policy-digest`), so a CI gate proves the receipt was accepted under a specific policy rather than whichever policy it happens to carry.

Policy profiles. A `buildlang-check-policy/v1` document turns receipt evidence into a gate: `allowed_effects` (denied effects always fail), `direct_effect_allowlist` and `direct_capability_sour` (narrow approved direct boundaries to specific helpers/macros/FFI sources), `propagated_effect_allowlist` and `propagated_effect_source_allowlist` (narrow approved propagated callers to specific callees), and coverage requirements that reject stale allowlist entries. Built-in profiles range from `pure` (denies every ambient capability) through `console-only`, `offline` (local file/environment/clock/console, no network/process/FFI/GPU), `ci-review`, to `strict-accountability`. The effect-system facts—not assertions—are what the policy is evaluated against.

5 Accountable scientific compute

5.1 The receipt

`buildc run --emit-receipt` compiles a program to C, runs it, captures every whitespace- or newline-separated token that parses as a finite `f64` as one entry of a measurement series, checks a stated invariant over that series, and emits a sealed `buildlang-scientific-runtime-receipt/v0`. Non-finite values mean divergence: parsing stops at the offending token, only the finite prefix is stored, and the receipt is UNVERIFIABLE with a NONFINITE_OBSERVED label—a diverged run is never a PASS.

5.2 The invariant family

Each invariant reduces the series to a *violation count*; the verdict is uniform across the family: PASS iff the violation count is zero and the series has at least two points. The tolerance is a fixed property of the invariant, sealed in the receipt, and the verifier rejects a receipt that resealed a non-canonical tolerance, so a receipt cannot weaken its own check. The seven invariants are

mathematically distinct—the same series yields different verdicts across them, which is why they are separate:

Invariant	sealed name	a step violates when	tolerance
`energy-monotone`	`energy_monotone_nonincreasing`	$s[k{+}1] > s[k] + tol$	10^{-12}
`conservation`	`conserved_quantity_constant`	$\|s[k] - s[0]\| > tol$	10^{-9}
`bounded`	`bounded_by_initial_maximum`	$s[k] > s[0] + tol$	10^{-9}
`energy-identity`	`energy_identity_residual`	$\|s[k]\| > tol$	10^{-9}
`relation`	`relation_columns_agree`	row columns differ by $> tol$	10^{-9}
`conserved-band`	`conserved_within_band`	$\|s[k] - s[0]\| > tol$	5×10^{-3}
`non-negative`	`non_negative`	$s[k] < -tol$	10^{-9}

Conservation fences both sides of $s[0]$; *bounded* (the discrete maximum principle) fences only the upper side, so a series that dips and returns to its initial value passes *bounded* while failing *conservation*; *energy-monotone* forbids any step-wise rise; *energy-identity* is the odd one out—its reference is zero (an absolute bound), so every step including step 0 is checked, and its series is a per-step energy-balance residual rather than a trajectory. *Conserved-band* is approximate conservation: a leapfrog oscillator's energy oscillates in a measured $\sim 1.25\times10^{-3}$ band around H_0 with no secular drift, passing the 5×10^{-3} budget, while explicit Euler's energy grows by $(1{+}dt^2)$ per step and leaves the band within two steps. *Non-negative* is the first algorithmic member: a binary search on 1024 elements prints the slack $11 -$ probes per lookup, which stays non-negative (pass), whereas a linear scan's slack drops to about -1013 (fail).

5.3 Negative fixtures

Every invariant ships a paired positive/negative kernel. A negative fixture is a program whose invariant is *expected* to fail; run with `--negative-fixture` it yields FAIL_EXPECTED, demonstrating that the checker catches these specific crafted violations (evidence against false negatives on the fixture set, not a proof of completeness). Without the flag the same failing run is FAIL_UNEXPECTED, because an unexpected invariant violation is a red flag, not a demonstration of the checker. The corpus spans physics (1-D heat equation FTCS energy dissipation; symplectic versus explicit-Euler oscillator), quantum (Born-rule normalization under a unitary gate versus a non-unitary gain), chemistry (reversible-reaction atom balance versus a stoichiometry bug), and algorithms (binary-search and funnel-hashing [4] probe bounds versus linear scans).

5.4 Verify by re-running

`buildc receipt verify` dispatches on the receipt schema. For a scientific-runtime receipt it *re-runs and re-checks* rather than trusting stored numbers: it recomputes the seal (an integrity gate before any sealed field is interpreted), re-derives the source and input-graph digests, re-runs the program with the recorded arguments, re-checks the measurement count, and recomputes the verdict triple (status, violation count, receipt status); any drift is a typed verification failure. Crucially, it checks the *verdict*, not exact floats, so a receipt emitted on one machine re-verifies on another despite last-bit differences—the verdict-count rule is robust to platform float variation and codegen reassociation. Diverged runs are the exception: the finite-prefix length is platform-dependent, so when both the receipt and the re-run record divergence, the reproduced divergence itself is the faithfulness signal.

A receipt is faithful if it reproduces, but a faithful receipt that records an unexpected failure is not a pass: the exit code reflects the verdict too. `buildc receipt verify r.json && deploy` is therefore safe—it will not deploy on a receipt that records an unexpected invariant violation or a diverged run, while a negative fixture reproducing its expected failure legitimately passes.

5.5 Effect-derived honesty fields

The typed-effect system supplies three fields of the scientific receipt as *witnessed facts*, derived fail-closed from the capability set rather than asserted: `input_dataset` (a capability is a dataset hazard unless it provably cannot feed external data—FileSystem, Network, Environment, Foreign, Gpu, and stdin-reading Console all fence the field as Possible_unwitnessed; a program with none of them provably consumed no external dataset), `seed` (not applicable in v0, since the language has no RNG builtin), and `determinism` (derived from every nondeterminism source the language exposes). Any capability this build does not recognize is treated as a hazard, so a capability added later cannot silently widen the claim.

6 The honesty discipline

Every scientific receipt carries the label Not_a_new_physical_law and a machine-readable `not_claimed` set that *must* include `physical_law` or verify rejects the receipt outright. The receipt witnesses one thing: the compiled program's observed output series satisfies the stated invariant (or, for a negative fixture, expectedly violates it). It does not prove the underlying PDE is solved correctly, that the discretization is convergent, that the parameters are physical, or that the result matches a reference solution; those are the program author's responsibility, and the receipt makes no statement about them.

The discipline is symmetric: a verifier worth trusting must be shown to fail. Negative-fixture kernels close the can-it-fail gap on the invariants; `buildc receipt verify --self-test` closes it on the verifier, tampering several distinct sealed fields and asserting each is rejected with its expected typed failure class. The failure taxonomy is closed and stable: Malformed, Schema_unsupported, Compiler_mismatch, Overclaim_boundary_missing, Field_contract_vi
Effect_policy_drift, Source_digest_mismatch, Measurement_count_drift, Invariant_status_drift, Seal_mismatch, and Invariant_not_held, each with a stable exit code. Receipts are parsed through a strict reader that rejects duplicate object keys at any depth—with a permissive last-duplicate-wins reader, a document carrying two `receipt_status` keys can show one value to a hasher and another to a reader, a seal-forgery vector.

A receipt chain binds multi-stage computations into one ordered, tamper-evident bundle: `chain build` records each member's seal in order and computes a chain seal; `chain verify` recomputes the chain seal, checks each member's current seal against the pinned seal, and re-verifies each member through the real verify path. A corpus command runs every kernel pair, asserts the emitted status equals the declared one, and re-verifies each receipt—a gate that can fail rather than a rubber stamp.

Seals are not byte-reproducible, by design. The scientific receipt seals the SHA-256 of the compiled program binary, and that binary is not reproducible across builds (the C toolchain embeds timestamps, temp paths, link order). Verify never re-checks the executable digest for equality; it re-runs the program and re-checks the verdict, and executable reproduction is *reported*, never required. Byte-reproducibility of the receipt artifact is a non-goal; re-checkability of the verdict is the guarantee.

7 Evaluation

The compiler passes 940 library tests, 140 binary, 309 CLI, 52 lexer, and 88 parser tests with none failing (baseline 2026-07-02), with `cargo fmt --check` clean. The library tests cover type inference and effects, the experimental `#[linear]` no-cloning attribute (which conservatively rejects a regression-tested set of compositional escapes but is honestly marked not yet fully

sound), lexer/parser units, the MIR and codegen paths across backends, the semantic-corpus receipt builders, and LSP dispatch.

The scientific-compute evaluation is the ten positive/negative kernel pairs of Section 5.3. Each pair is a falsifiable demonstration: the positive kernel passes its invariant, the negative kernel fails it, and the corpus command asserts both classifications hold and both receipts re-verify. The capability-provenance evaluation is the regression-tested coverage of the escape classes of Section 3.2—each escape class (callback alias, struct-field storage, control-flow selection, async, destructuring, and the rest) has a test that a pure context cannot silently absorb an effectful value.

8 Discussion

BuildLang's two receipt families share a single mechanism: the typed-effect system witnesses the facts a receipt seals. The codegen receipt seals capability facts directly; the scientific receipt derives its dataset, seed, and determinism fields from the same capability facts, fail-closed. This is the systems analogue of the companion EMET witness [1]: both seal a re-derivable verdict rather than trusting stored bytes, both carry a closed verdict lattice, both insist on a failure-proof, and both bound their own claims in machine-readable form. The difference is the witness source—SHA-256 over raw bytes for EMET, the type-and-effect system for BuildLang.

The open questions are honest. The `#[linear]` no-cloning attribute is not yet fully sound; full soundness needs an affine/borrow checker on the MIR. The scientific receipt checks one invariant over one series per receipt (v0). Cross-language parity of the experimental stripped-credential rebind is a follow-on. The receipt seal is an unkeyed SHA-256 over the canonical form: it detects accidental corruption, but anyone can recompute it after editing descriptive metadata; integrity for the verdict that matters comes from the re-run, not from the seal. We state this rather than overclaim cryptographic tamper-proofing.

9 Reproducibility

BuildLang, its compiler, the eight capability effects, the codegen and scientific-runtime receipts, and the ten positive/negative scientific kernel pairs are public under the project license at `https://github.com/HarperZ9/buildlang`. A reader can re-emit any receipt from its source and re-verify it by re-running the program, without trusting the stored series byte-for-byte.

Availability

BuildLang is at github.com/HarperZ9/buildlang.

References

[1] Harper, Z. D. *EMET: An Authority-Incapable Byte-Level Integrity Witness for AI Oversight.* Companion paper, 2026.

[2] Bauer, A., and Pretnar, M. *Programming with Algebraic Effects and Handlers.* J. Log. Algebr. Meth. Program. 84(1):108–123, 2015. doi:10.1016/j.jlamp.2014.02.001.

[3] Plotkin, G. D., and Pretnar, M. *Handlers of Algebraic Effects.* ESOP, 2009.

[4] Farach-Colton, M., Krapivin, A., and Kuszmaul, W. *Optimal Bounds for Open Addressing Without Reordering.* arXiv:2501.02305, 2025.

[5] Reproducible Builds Project. *Reproducible Builds: a set of software development practices.* `https://reproducible-builds.org`, 2026.

[6] NIST. *Secure Hash Standard (SHS)*. FIPS PUB 180-4, 2015.