

# Contents

<b>Reducing Control Flow to Tensor Algebra: Verifying the Non-Learned Trusted Base of a Neuro-Symbolic Substrate</b>	<b>1</b>
Abstract . . . . .	1
Background . . . . .	2
1. Introduction . . . . .	3
2. The compiled tensor-op graph . . . . .	3
3. The obligation framework . . . . .	5
4. Faithfulness: the reduction is computed exactly . . . . .	9
5. Scope . . . . .	14
6. Related work . . . . .	15
7. Hardware formal verification: the thermodynamic compile target . . . . .	16
8. Source-language frontends: empirical end-to-end verification . . . . .	18
9. Conclusion . . . . .	19
References . . . . .	19

## Reducing Control Flow to Tensor Algebra: Verifying the Non-Learned Trusted Base of a Neuro-Symbolic Substrate

---

### Abstract

Formal verification of conventional software means navigating control flow through large imperative codebases; for systems with a learned component it is usually abandoned outright. We show that **Sutra**, a typed purely-functional language, changes the shape of the problem for the non-learned part of a system, because its compiler turns an entire program (primitives, control flow, string I/O) into a single fused **tensor-op graph** over a frozen substrate, and that graph *is* the program's semantics (as a neural network's weights are its computation), not a residual to be interpreted. The construct that makes conventional verification expensive, the branch, does not survive into the graph: **if/else** compiles to a **single three-valued-Kleene polynomial**, Lagrange-interpolated and exact on the  $\{-1, 0, +1\}$  truth grid, and each loop to a bounded soft-halt recurrence. Verifying the **trusted base** (kernel roles and named critical programs) therefore becomes algebra over a small fixed set of tensor graphs rather than enumeration of control-flow paths.

We make this precise as three per-construct obligation families (contract, branch-range, termination), and we give **mechanical checks for all three**, running on the real compile-and-execute substrate: a codegen-correspondence check that the polynomials the compiler emits agree with the spec on the Kleene grid (worst  $|\text{error}| = 0.0$  across the nine  $\{-1, 0, +1\}^2$  points, a regression guard against codegen drift, *not* a math-discovery claim about Lagrange interpolation, which is exact by construction; §3.2), connective range-soundness (a closed-form proof that outputs lie in  $[-1, +1]$  over the whole fuzzy domain), and loop termination (bounded, monotone halt). The termination story is sharper than a blanket "Sutra bans unbounded loops": the surface syntax distinguishes **loop / while\_loop** (bounded soft-halt recurrence with a termination obligation) from **recur** (an explicitly non-halting form for event-driven recurrences, declared outside the trusted-base scope by construction); §3.3 names the split. We also discharge the kernel-enforced read/write confinement half of the contract obligation. We further give a **decision procedure for program equivalence**

**over the polynomial fragment** (Boolean logic and integer arithmetic): a checker extracts each expression's polynomial via the compiler's own lowering and decides whether two programs compile to the same tensor graph by polynomial identity, exactly, or in poly time (Schwartz–Zippel), for arbitrary depth. This separates two notions that are usually conflated, compiling to the same graph versus being logically equivalent, and we exhibit distributivity as a clean witness that the former is strictly stronger.

The reduction is meaningful because the substrate computes the compiled graph exactly, established with measured results (§4): rotation binding decodes bundles at 100% accuracy through width  $k = 8$  on four frozen embedding substrates where the Hadamard baseline has collapsed to 2.5–7.5%, with a bind/unbind round-trip of  $1.5 \times 10^1$ ; and Sutra's compiled arithmetic, operator selection included, runs exactly on the substrate (`demos/calc` evaluates 11/11 expressions exactly against a rational oracle; `demos/echo` round-trips strings bit-exact at runtime dimension 16). §4.5 reports a worked example of why dispatch-level cleanliness is necessary but not sufficient: a runtime-prelude leak in the `eq()` method (`float(cos.item())` severed autograd, the exact host-extraction pattern the spec bans) survived the broader sweep, which greps user `.su` output, not the `_TorchVSA` prelude itself; a differentiability test caught it downstream. The fix shipped with a sweep-coverage extension that scans the prelude under a method-level allowlist of legitimate host substrate boundaries, and a second constrain-train experiment (training `beta` in `defuzzify_trit` end-to-end through the compiled graph) shows the substrate now carries autograd cleanly across the equality surface, both with measured numbers. The scope is the non-learned trusted base, per published contract; §5 states it precisely and §6 positions the work against neural-network verification, SMT for nonlinear arithmetic, partial evaluation, and vector-symbolic architectures. Then §7 turns from software to **hardware** formal verification, on Sutra's **energy-based** thermodynamic compile target: Lean-machine-checked proofs that its gadgets have the correct output as the **strict global energy minimum** (a ground-state decode is exact), with *sampler-convergence* proven (stationary measure unique, and the two-state mixing rate mechanised: the gadget's Gibbs kernel mixes in one step). Finally, §8 is a brief note on a **separate, weaker** empirical layer, a compile-and-run-against-ground-truth suite for the source-language frontends that compile *into* Sutra, explicitly not conflated with the formal guarantees above.

---

## Background

Sutra is a typed, purely functional programming language whose compiler lowers an entire program to a single fused tensor-op graph over a frozen embedding substrate. There is no bytecode or interpreter beneath the graph: compilation produces a weight-and-rotation structure, and running the program is the forward pass through it, in the same sense that a trained neural network's weights are its computation. The substrate is a high-dimensional vector space supplied by a frozen pre-trained embedding model (nomic-embed-text by default), extended with a small block of synthetic axes that carry the real and imaginary parts of numbers, a truth coordinate, and string-encoding flags.

The primitives the graph is built from come from Vector Symbolic Architectures, also called hyper-dimensional computing: binding composes a role with a filler, bundling superposes several vectors into one, and cleanup decodes a noisy superposition back to its nearest stored item (Plate 1995; Gayler 2003; Kanerva 2009). Sutra uses these as its data-structuring layer: a record is a bundle of role-filler bindings, a string is a synthetic-axis-encoded array of codepoints, and a number lives

on the synthetic number axis. Control flow is expressed without host branching: a conditional becomes a polynomial blend over a three-valued truth grid, and a loop becomes a bounded soft-halt recurrence.

A system built on this substrate has two parts with different epistemic status. The frozen embedding model is a learned, opaque component; its internal semantics are not verified, only its interface. Everything the compiler emits above it, the program's control flow, arithmetic, string handling, and role structure, is non-learned: a fixed tensor graph determined entirely by the source. This paper concerns the verification of that non-learned trusted base, and the contracts that quarantine the learned component behind it. The reader needs no familiarity with the embedding model's training; the claims are about the compiled graph, which is the same object whether the substrate is nomic-embed-text or another frozen model of the same dimension.

---

## 1. Introduction

Two facts are usually taken to be in tension. (i) Critical systems want formal guarantees about their trusted base. (ii) Useful systems increasingly contain learned components, which resist formal guarantees. The common resolution is to verify neither: the imperative trusted base is too large to verify cheaply, and the learned part is given up on, so the whole stack ships on testing alone.

Sutra offers a different decomposition. It is a typed, purely functional language whose compiler reduces an entire program (primitives, control flow, string I/O) to a single fused tensor-op graph over a frozen embedding substrate. The claim is narrow and structural:

For the **non-learned** trusted base, compiling the program to a tensor-op graph turns verification from control-flow path enumeration into algebra over a small fixed set of tensor graphs.

This does not make the learned parts safe. It makes them *separable*: the boundary between "compiles to a checkable tensor graph" and "depends on a learned weight" is syntactically visible, so the trusted base can be verified while the learned part is quarantined behind contracts and monitoring.

**Contributions.** 1. **The reduction** (§2): why the compiled tensor-op graph is the program's semantics rather than a constant-folded residual or a deep-learning computation-graph optimization, and why equivalence on it is algebra. 2. **The obligation framework with mechanical checks** (§3): three per-construct obligation families (contract, branch-range, termination), each with a check that runs on the substrate. The branch-range family (§3.2), built on **three-valued polynomial Kleene logic**, is the one that removes path explosion: branches become closed-form polynomials, not forks. 3. **An equivalence decision procedure for the Kleene fragment** (§2): deciding same-graph by polynomial identity, distinguished from logical equivalence, with distributivity as a witness. 4. **The faithfulness evidence** (§4): measured substrate exactness, including bit-exact arithmetic dispatch through the compiled substrate, restated self-containedly here.

§5 states the boundary; §6 positions the work in the literature.

## 2. The compiled tensor-op graph

Sutra's compiler emits, for each program, a fused tensor-op graph over a frozen embedding substrate: compilation produces the weight/rotation structure, and execution is the forward pass. The graph

is the program's semantics in the same sense that a neural network's weights are its computation: there is no residual program underneath waiting to be interpreted.

This distinguishes the compiled graph from three neighbours it is easy to confuse it with. **Against the specialization spectrum:** constant propagation, partial evaluation / Futamura projections (Futamura 1971) and multi-stage programming (Taha & Sheard 2000) remove known subexpressions from a program that still runs in a conventional operational model; here there is no conventional model left to run in. **Against symbolic execution**, which enumerates path conditions through an interpreter and suffers exactly the path explosion we remove, the compiled graph has no path set to enumerate: a conditional is a single polynomial, not a branch in an execution tree. **Against deep-learning graph optimization** (operator fusion, XLA-style rewriting), those preserve a graph that already exists; here the graph *is* the program's semantics, produced by compilation from source, and the verification question is about that semantics, not about speeding up an existing tensor program.

The verification-relevant consequence: equivalence checking moves onto the compiled graph as **algebraic comparison**, not a traversal of possible executions.

**A decision procedure for the polynomial fragment, Kleene logic and integer arithmetic.** The fragment is not limited to the Boolean connectives. For programs built from the Kleene connectives ( $\&\&$ ,  $\|\|$ ,  $!$ ) and integer arithmetic ( $+$ ,  $-$ ,  $*$ ), nested to any depth and freely mixed, we decide equivalence outright. A checker (`fv_obligation_checker.py`) extracts each expression's polynomial by running the compiler's own inliner pass, not a hand-copied formula, and walking the lowered arithmetic into a polynomial, then decides whether two programs **compile to the same graph** by polynomial identity. Two routes decide the *same* notion: an *exact* symbolic check  $\text{expand}(p - p) = 0$ , and a *poly-time* randomized check (Schwartz-Zippel over a finite field) that scales to deep nestings the exact route cannot reach, §3 quantifies both and the trade-off. The checker also decides the weaker **logical** equivalence, agreement on the  $\{-1, 0, +1\}$  Kleene grid, and reports both, refusing (rather than guessing) on any term outside the polynomial fragment, such as a comparison or a runtime intrinsic.

These two notions are not the same, and separating them is a result in its own right. De Morgan, commutativity, and double negation compile to *identical* polynomials, same graph. **Distributivity does not:**  $a \ (b \ c)$  and  $(a \ b) \ (a \ c)$  agree at all 27 grid points (they are logically equivalent) but compile to *different* polynomials off-grid. So "compiles to the same graph" is strictly stronger than "logically equivalent"; the graph comparison decides a well-defined sublattice of logical equivalences, and the checker decides exactly which side of that line any given pair falls on. The *arithmetic* side of the fragment sharpens the picture: arithmetic distributivity  $(a + b) \cdot c = a \cdot c + b \cdot c$  is a same-graph identity (the two compile to the *same* polynomial), the exact mirror image of Kleene distributivity, which is not. The same checker decides both, Boolean and integer-arithmetic equivalence, by the one polynomial-identity test. This has a direct use beyond the trusted base's own logic: **verifying that a compiler optimization preserves semantics.** Horner's method  $a \cdot x^3 + b \cdot x^2 + c \cdot x + d$  and  $((a \cdot x + b) \cdot x + c) \cdot x + d$  compile to the *same* graph (the rewrite is sound); constant folding and reassociation likewise; and an incorrect rewrite, a sign flip  $a \cdot x^2 + b \cdot x + c$  vs  $a \cdot x^2 - b \cdot x + c$ , is caught as a *different* graph. The decision procedure is the same; only the inputs change.

### 3. The obligation framework

Verifying the trusted base concentrates into a small set of closed-form obligation families, one per Sutra construct that survives into the compiled graph: contracts (§3.1), branches (§3.2), loops (§3.3), and, once the base does arithmetic the substrate's native float range cannot hold exactly, digit-array carry propagation (§3.4). Each has a mechanical check that runs on the real compile-and-execute pipeline.

**3.1 Contract obligations.** Each trusted program carries an *axon-typed contract*. An **axon** is a structured embedding, a single vector carrying named role→filler slots via rotation binding (the VSA operations of §4), so a program's typed interface is "the set of named roles it reads and writes." The contract names the input roles the program may read, the output roles it may write, and its status conditions. For program *p* with contract *C*, the obligation is that *p*'s compiled graph reads only *C.read\_roles*, writes only *C.write\_roles*, and that the role-to-role function it computes is the one *C* specifies. The compiler already emits the static read/write key sets (`AXON_KEYS_READ`, `AXON_KEYS_BOUND`) that seed the role half of this obligation.

The **read/write confinement** part is **discharged at the runtime kernel**, the capability-checked axon router that enforces Sutra's role model: a program can only emit on roles in its `write_roles` (capability-checked at routing) and is delivered only axons on roles in its `read_roles`, with no cross-role leakage, mechanically tested (three kernel tests, including a two-role read-isolation check). The **role-to-role function** part is **discharged for the Kleene-logic fragment**: when a contract states the intended function as a reference expression, "does the implementation compute it?" is exactly `reduces_to_same_graph(implementation, reference)` (§2), decided exactly, any depth. (Demonstrated: a NAND contract `!(a&&b)` is satisfied by the De Morgan implementation `!a||!b` and correctly rejects a NOR implementation.) The **key-soundness** part, that the static `AXON_KEYS` analysis matches the keys a program touches at runtime, is **discharged by opt-in runtime key-usage instrumentation**: the runtime's axon read/bind methods record, when enabled, the key of each access (a string by name; a non-string, statically-unnamable key as `<dynamic>`), and soundness is the set inclusion `runtime_keys ⊆ AXON_KEYS`. The check is non-vacuous: a program touching only its statically-collected keys is sound, while a read or bind of an uncollected key, or any `<dynamic>` key, is caught. (The instrumentation is off by default, so it adds nothing to the compiled hot path; it is a monitoring recorder around the substrate ops. The check witnesses the executed paths; a path-coverage argument or a key-level manifest would make it exhaustive rather than execution-witnessed.) With role confinement (kernel), function-correctness (Kleene fragment), and key-soundness all in hand, the contract obligation of §3.1 is discharged rather than half-done.

**3.2 Branch-range obligations (from polynomial Kleene logic).** This family carries most of the weight, because branches are what make conventional verification expensive: each `if/else` doubles the path set, so a trusted base with *b* branches presents up to  $2^b$  paths. Sutra removes the branch as a control-flow object. Source `if/else` compiles to a **single polynomial** that interpolates between the branch values on a fuzzy truth value; the connectives are the **three-valued Kleene operators** (`and`, `or`, `not`, the t-norms) realised as **Lagrange-interpolated polynomials exact on the 3×3 Kleene grid** over  $\{-1 = \text{false}, 0 = \text{unknown}, +1 = \text{true}\}$ , branchless and smooth (hence gradient-compatible) off the grid.

Two consequences matter. First, **branchlessness collapses the path set**: a branch is a polynomial whose value the truth-axis scalar determines, so the obligation is a closed-form bound on that polynomial's range and sign over  $[-1, +1]$ , a polynomial extremum problem, not a path walk. Second, **three-valued rather than Boolean is the right logic for a substrate that mixes**

**exact symbolic and uncertain learned signals:** the middle value (unknown) is first-class, so the verifier reasons about "undetermined" directly, while crisp true/false stays bit-exact because the interpolation is exact on the grid.

**Grid-exactness is discharged mechanically, as a codegen-correspondence check, not a math-discovery claim.** A degree- 2-per-variable polynomial interpolated through the nine  $\{-1, 0, +1\}^2$  grid points hits those nine points exactly by construction; that piece is Lagrange interpolation, not a result. What the check verifies is something distinct and load-bearing: that the polynomial the *compiler actually emits* at the end of `parse`  $\rightarrow$  `inline`  $\rightarrow$  `simplify`  $\rightarrow$  `tensor-op` `codegen`  $\rightarrow$  `runtime` agrees with the spec polynomial on the grid. A typo or rewrite-pass bug in the codegen, a stray sign, a missing  $a^2b^2$  term, a constant folded the wrong way, would show up as a non-zero grid error even though Lagrange interpolation as a method is untouched. So "worst |error| = 0.0 across the grid" is a regression guard against codegen drift, asserting that the chain ending at the substrate's tensor ops still produces the spec connectives. Measured value reported as the empirical discharge of the check, not as a mathematical discovery. The polynomials checked are the ones the compiler emits:  $a \&\&b = (a+b+ab-a^2-b^2+a^2b^2)/2$ ,  $a || b = (a+b-ab+a^2+b^2-a^2b^2)/2$ ,  $!a = -a$ .

**Range-soundness is discharged in closed form.** What soundness requires is that the connectives never produce an out-of-range truth value anywhere in  $[-1, +1]^2$ . We prove this with a polynomial range-bouder (`fv_poly_bound.py`) that computes the exact global extrema of a polynomial over an axis-aligned box by the compact-domain extremum argument, the extrema lie at stationary points of the restriction to some face of the box, so the candidate set is the box corners and the edge-interior and interior gradient-zero points, solved and evaluated in exact (rational/algebraic) arithmetic. On the three connectives it returns **exact range**  $[-1, +1]$ , a proof, not a sampled min/max. To ensure the bound applies to *what the compiler emits*, the test first cross-checks the symbolic polynomial against the substrate on the  $\{-1, 0, +1\}^2$  grid (which uniquely determines a degree- 2-per-variable polynomial) plus off-grid points (agreement to  $6 \times 10^{-6}$ ), then bounds. (`test_fv_poly_obligation_checker.py`; `grid-exactness: test_fv_kleene_grid_exactness.py`.)

The same grid saturation makes selection exact: a sufficiently sharpened softmax `select` is a *true* one-hot, because `exp(-k)` underflows to exactly 0 (in float32 for modest `k`, far below ulp in float64), so unselected branches are multiplied by exact zero, the mechanism behind the bit-exact operator dispatch in §4.3.

**3.3 Termination obligations (from soft-halt loops).** Each loop is a bounded recurrence `state`  $\leftarrow$  `R`  $\cdot$  `state` with a fixed-width state vector and a halt cell. Termination reduces to "the halt signal is monotone within bounded steps," discharged per loop, far smaller than proving an arbitrary `while` terminates.

We are explicit about what this is and is not, since "all loops are bounded" can read as a sidestep. It is a deliberate **language design choice**, and one that has been made *visibly* at the surface syntax level, Sutra distinguishes two forms with two purposes:

- **loop (cond) / while\_loop** (this section): a bounded soft-halt recurrence over a fixed-width state vector. Termination obligation applies, discharges as described below, and the trusted base is composed exclusively of this form.
- **recur(...)** (non-halting; Sutra's explicit non-halting-loop primitive, shipped in this work's reference implementation): an *explicitly non-halting* loop, used for UI tick-loops, event-driven recurrences carrying substrate state across iterations, and other cases where the program *should* run forever. `recur` does not pose a termination obligation because it asserts non-

termination as its declared semantics. The trusted base does not use `recur`; programs that do are outside the scope of the FV agenda by construction (and the obligation framework reports this without an attempt to prove a property the form does not claim).

Naming both forms explicitly addresses the natural worry that "Sutra bans unbounded loops" is a sidestep: the language design **separates** the cases rather than collapsing them, so the absence of an unbounded `while` in the trusted-base fragment is a meaningful scope claim, not a missing feature. With this split, what §3.3 covers is bounded recurrences specifically. **We do not claim a novel attack on the halting problem, and the by-construction nature is the point, not a hidden circularity:** by *excluding* the undecidable case (unbounded `while`) from the trusted-base fragment, and reporting, rather than silently accepting, any program that uses the non-halting `recur`, we are left with a fragment on which a *decidable* obligation remains. That obligation is not vacuous: for each bounded soft-halt loop one must still check that the halt signal is **monotone** and **crosses its threshold within the bound** (rather than the loop running to the bound every time), which is a real mechanical check on the emitted recurrence, not an assumption. The contribution here is the clean *separation* that turns the trusted base's loops into a checkable fragment, plus that convergence check, not a claim to decide termination of arbitrary programs. It is a real, useful property for a trusted base, a kernel role must not hang, but it is **not** functional correctness, which is a separate obligation (§3.1, discharged for the Kleene fragment) and not subsumed by termination.

This is discharged structurally and observably. Structurally the emitted loop is `for _t in range(max_iters)` (bounded by construction) with `halted = min(halted + halt, 1)` and `halt = sigmoid(·) * 0` (monotone, capped at 1; on saturation `state = (1-halted)·cand + halted·state` freezes). Observably on the torch substrate: a non-converging loop runs to the bound and stops (`iters_active = 9.998/10`, never exceeding `max_iters`); a converging loop is **exactly frozen** across unroll depth, its state at `T=20` equals its state at `T=10`, `diff = 0.0`. (`test_fv_termination.py`)

**Tooling.** Off-the-shelf SMT solvers target Boolean and linear arithmetic, not the polynomial obligations the compiled graph produces; §6 discusses where nonlinear solvers such as dReal fit. The per-construct discharges above use concrete finite methods: grid-exactness is a nine-point evaluation; range-soundness is a closed-form critical-point bound; termination is structural plus a saturation observation; equivalence is symbolic polynomial identity.

**Range-soundness scales to arbitrary depth by composition, the bounder is NOT on the critical path for depth.** This is worth stating directly, because the natural worry is that deep nesting produces a high-degree polynomial the closed-form bounder cannot handle. It does, and we do not bound it. The closed-form critical-point bound gives the exact range of a *single* connective; the *composed* polynomial of a deeply nested expression is high-degree and bounding it directly is expensive — measured, the exact box bound completes a single connective in 0.1 s but does not finish even a depth-2 composition (`(a && b) || c`) within 30 s, because the critical-point box search blows up the moment degree and arity climb past one connective. We do not need to: each connective is proven to map `[-1, +1]` into `[-1, +1]` (its exact range *is* `[-1, +1]`), so any expression built solely from the connectives, over truth-axis inputs in `[-1, +1]`, has range within `[-1, +1]` **by induction on the expression tree**, independent of nesting depth and degree. The check (`range_sound_by_composition`) verifies an expression is such a composition (refusing if it uses a non-connective operator), and decides range-soundness for arbitrarily deep nestings instantly. So the equivalence procedure (degree-insensitive polynomial identity) and range-soundness (degree-insensitive composition) both scale; the closed-form bounder remains the exact tool for the per-

connective lemma they rest on.

**The composition argument is structural, not numerical, substrate noise is a separate concern, addressed in §4.** A reasonable critique is that VSA operations accumulate noise at increasing bundle width, so the per-connective lemma (range =  $[-1, +1]$  exactly) is "leaky" once the connectives are realised on a real substrate. That critique conflates two layers that are deliberately kept separate. The composition argument here is *about the polynomial*: given the inputs of each connective are in  $[-1, +1]$ , the output of the polynomial the connective lowers to is in  $[-1, +1]$ , a closed-form fact about the polynomial, independent of how it's executed. Whether the substrate computes that polynomial *faithfully* (within machine epsilon, or bit-exactly under the integer-exact-range conditions named in §4.3) is a separate, measured question. The two layers stack: §3.3 says the *abstract* range is sound for any depth; §4 (esp. §4.1's capacity curve and §4.3's bit-exact dispatch) says the *substrate* realises that abstract range to documented precision within the trusted-base usage envelope. Conflating them would let either layer's limitations contaminate the other's claim; keeping them separate is what lets each layer's argument be precise.

**Cost of the equivalence check, and a poly-time decision procedure that scales.** The *exact* identity check `expand(p_a - p_b) == 0` is expensive, and we name why plainly: the Kleene lowering duplicates operands (`a && b` expands to a formula that mentions `a` and `b` several times), so the *inlined* arithmetic, and `sympy.expand` of it, is exponential in nesting depth, before any cancellation. Measured on the same `extract_truth_polynomial` pipeline the checker uses (balanced Kleene trees, var pool 3; `experiments/randomized_pit_scaling.py`): depth 1 → **6 monomials**, depth 2 → **312**, depth 3 → **infeasible** (`expand` killed after 30 s). This is the wall reviewers correctly flagged.

The wall is not intrinsic to deciding equivalence, only to deciding it by *expansion*. We add a **randomized identity test (Schwartz–Zippel)** that decides the same notion in polynomial time. Instead of distributing the polynomial, it evaluates the difference `p_a - p_b` at random points of a finite field  $F_p$  ( $p = 2^{61} - 1$ ), applying each connective's closed-form truth polynomial to its operands' *values* on the *original, un-inlined* expression tree, one number per node, **O(tree size) per trial**, no duplication, no `expand`. It is sound one-sided: any nonzero evaluation is an **exact disproof** with a witness point, and all-zero over `k` trials certifies identity with false-positive probability  $(\text{deg} / (p-1))^k$ . Measured on the same trees (32 trials):

nesting depth	leaves	<code>expand</code> (exact)	randomized PIT
3	8	infeasible (> 30 s)	0.003 s
6	64	infeasible	0.017 s
8	256	infeasible	0.039 s
10	1 024	infeasible	0.152 s
12	4 096	infeasible	<b>0.822 s</b>

So the procedure decides at **depth 12 (4 096 leaves)** in under a second what expansion cannot do at depth 3, with verdicts agreeing with the exact check wherever the exact check still terminates (De Morgan, commutativity, distributivity, and absorption are cross-checked in `test_fv_general_checker.py`). The connective formulas the evaluator applies are verified against the compiler's own inliner (`test_kleene_connective_formulas_match_inliner`), so the randomized check decides the *same* polynomial as `reduces_to_same_graph`, not a drift. The trade-off: the exact check is certain when it terminates; the randomized check trades that for a quantified,

negligible error (at depth 12 the bound is  $(1.7 \times 10^{-32} / 2^{61})^{32} \approx 10^{-360}$ ). The degree grows  $4^{\text{depth}}$ , so beyond  $\sim$ depth 30 a larger prime or CRT over several primes restores the margin, unnecessary for any realistic nesting. Full data: a companion finding in the repository (with the original expansion-cost table).

**3.4 Range-soundness and termination for unbounded-precision arithmetic (digit-array carry propagation).** The three families above cover the control-flow surface (branches, loops, contracts). A fourth obligation shape appears once the trusted base does *arithmetic the substrate's native float range cannot hold exactly*: arbitrary-precision integers, represented as a fixed-width digit array with carry propagation. The `digit_array_add` substrate intrinsic computes radix- $r$  addition entirely in tensor ops, pairwise sum, floor-division carry extraction, and an  $N$ -step shift-and-propagate, no `.item()` and no host scalar branch on a digit value. It carries two obligations, both discharged by the same finite reasoning the §3.2/§3.3 families use, lifted to the digit-array domain.

*Range-soundness (by induction on the step index).* The obligation is that every digit stays in  $[0, r)$  and every carry in  $\{0, 1\}$  at every step. Initially  $s = a + b \in [0, 2r)$ , so  $c = s/r \in \{0, 1\}$  and  $d = s - cr \in [0, r)$  by the floor-division identity. The invariant is preserved across each propagation step:  $d_{\text{new}} = d + c_{\text{shifted}} \in [0, r+1)$  (the maximal  $d_{\text{new}} = r$  is exactly the "9 + 1" cascade), so  $\text{new\_c} = d_{\text{new}}/r \in \{0, 1\}$  (since  $r+1 < 2r$ ) and the re-extracted  $d \in [0, r)$  again. By induction the output digit array contains only values in  $[0, r)$ ; the terminal carry is dropped (overflow saturates, by design). This is a closed-form invariant proof in the same spirit as the §3.2 range-bound, a fact about the arithmetic, independent of how the substrate executes it, for any positive integer radix (the shipped path uses  $r = 10$ ).

*Termination (structural, not measured).* The runtime is `for _step in range(n)` where  $n$  is the digit-array width, a *structural* shape parameter, not a data-dependent value. The body has no `break/continue/early-exit` and is a finite composition of tensor ops, so the loop runs exactly  $n$  iterations ( $O(N^2)$  element-wise work; the queued Hillis–Steele form would be  $O(N \log N)$ ) and cannot fail to halt on any input. The loop count depends only on the width, never on a digit value, the same "the trusted base does not pose the halting problem" property as §3.3, here because the bound is a shape, not a soft-halt signal. End-to-end the shipped intrinsic is bit-exact on the worked cases ("12345678" + "87654321" = "99999999"; "99999" + "1" = "100000"; overflow saturates at `max_digits = 16`; `experiments/bigint_worked_example.py`, nine cases).

What §3.4 does *not* yet cover: signed digit arrays (`v1` is unsigned), and expressing these bounds in the §3.2 polynomial-Kleene style rather than as a step-indexed induction (a wiring task, not a new result). The obligations, proofs, and arbitrary-precision spec are given in full in the repository.

## 4. Faithfulness: the reduction is computed exactly

A reduction to algebra is worth something only if the substrate computes the compiled graph *exactly*. This is not a circular assumption about an opaque substrate, and it is worth being precise about why.

**The substrate operations are formally-defined VSA operations with algebraic laws.** Bind, unbind, and bundle, the primitives the compiled graph is built from, are vector-symbolic-architecture operations, not ad-hoc tensor code. The holographic-reduced-representation algebra (Plate 1995) gives their laws, binding is **invertible** (`unbind(R, bind(R, x)) = x`) and bundling is a **linear superposition** whose decodable capacity grows with dimension (Frady, Kleyko & Sommer

2018; Kleyko, Rachkovskij, Osipov & Rahimi 2023). So the obligations the verifier discharges are algebra over operations that *have* a formal algebra; what is left to establish empirically is narrower and non-circular: how exactly a given **frozen embedding substrate** realises those laws. ("Frozen" = a pretrained embedding model whose weights are fixed and never updated, e.g. nomic-embed-text at 768 dimensions; Sutra binds and bundles *in that fixed space* rather than learning a new one.) The three results below are that realisation, the invertibility law to machine epsilon, and exact decode within capacity at the widths the trusted base uses, measured, with protocols restated here so the paper stands on its own.

**4.1 Bundle decoding, accurate well beyond  $k = 8$ , not just at it.** Protocol: for each bundle width  $k$ , bind  $k$  role-filler pairs by rotation, superpose (bundle) them into one vector, and decode each filler by unbind + nearest-codebook (argmax-cosine), 10 trials per width. The headline result is the **measured capacity curve**, not a single-point claim at  $k = 8$ :

$k$	nomic (768-d)	mxbai (1024-d)	all-minilm (384-d)
2	100.0%	100.0%	100.0%
4	100.0%	100.0%	100.0%
8	100.0%	100.0%	100.0%
16	100.0%	98.8%	92.5%
24	100.0%	95.8%	76.2%
32	99.1%	85.3%	66.9%
48	93.3%	(mem)*	42.3%

\*mxbai  $k = 48$  hit a memory-allocator error during Haar-QR matrix construction on this configuration; reported as missing data rather than guessed.

Read the table directly: **rotation binding stays at or above 99% accuracy through  $k = 32$  on the 768-d substrate, and 95% through  $k = 24$  on the 1024-d substrate.** Capacity grows with dimension exactly as the VSA literature predicts (Plate; Frady, Kleyko & Sommer). This is *not* a method whose ceiling is  $k = 8$ . That is the *comparison width* where the textbook Hadamard (element-wise) binding has already collapsed (2.5% on mxbai-embed-large, 7.5% on all-minilm) while rotation binding holds. Hadamard never exceeds 95% on any substrate even at  $k = 2$ , and is below 50% by  $k = 48$  on all three. Beyond text, the same protocol gives 100% through  $k = 8$  on the ESM-2 protein model, where Hadamard is similarly collapsed at modest widths, the property is substrate-independent within the dense-encoder family.

The capacity curve's roll-off at large  $k$  does **not** undercut the verification claim, because the two concern different objects. Bundling capacity is a property of VSA *associative memory*, how many items survive superposition, and that lossy, graceful-degradation regime is **not part of the trusted base** and is **not what the obligations verify**; the verified object is the compiled arithmetic/control graph, whose exactness (§4.3) is bit-level integer dispatch, independent of how many items a bundle could hold. What verification needs from bundling is narrower than maximum capacity: the bundle/bind/unbind primitives the compiled graph is built from recover their inputs exactly at the small, fixed widths the trusted base actually uses (a kernel role's axon carries a handful of named slots, not hundreds). The trusted-base widths are typically 8, and the curve shows the primitives work accurately at order-of-magnitude more capacity than that requirement, so the measured roll-off is reported headroom, not a crack in the exactness it is sometimes misread as contradicting. (10

trials per  $k$ ; the full table including signal cosines and the Hadamard comparison is a companion finding in the repository.)

**4.2 Reversibility.** A single bind+unbind cycle returns the input at the floating-point noise floor: mean  $\|\text{unbind}(R, \text{bind}(R, x)) - x\| = 1.5 \times 10^{-1}$  across all four substrates, the rotation is invertible to machine epsilon.

**4.3 Exactness of the compiled arithmetic dispatch.** Bit-exactness here is a property of *Sutra's compilation*, not of any application. Two kernel-free demos in this repository exercise it with no OS, kernel, or router in the loop, each compiles a `.su` source and calls its substrate entry point directly. In `demos/calc`, the operator is *selected on the substrate* from its character's codepoint (`string_char_at` + a softmax-saturated `select`, §3.2) rather than by a host dispatch table, and the arithmetic runs on the substrate in float64 (exact integers to  $2^3$ ): **11/11 expressions evaluate exactly** against an exact-rational oracle, **6/6** inexact or unparseable inputs are *refused* rather than approximated, and **7/7** result strings are decomposed exactly on the substrate. In `demos/echo`, a string rides a single rotation binding into an axon and back, **bit-exact on 5/5 round-trips** down to runtime dimension 16. Both run at small width, `demos/calc` at the audited floor of `runtime_dim = 8`, with no `basis_vector` calls so the semantic codebook is unused, so the exactness is the dispatch's, not an artifact of high dimension. Reproduce in-repo: `python -m pytest demos/calc demos/echo` (32/32, measured on torch + CUDA with nomic-embed-text). The property follows from the lowering, so it holds for any Sutra program that compiles arithmetic the same way.

A fair objection, and the standard one against any "bit-exact on GPU" claim, is that float32 on a GPU is generally non-deterministic across runs: warp scheduling reorders work, and reductions (sum-of-many-elements, particularly under atomic add) accumulate in non-deterministic order, so identical inputs produce different bit patterns on different runs. We name the objection and dispatch it explicitly:

1. **The dispatch pipeline contains no reductions over many elements.** It is element-wise tensor ops + a single saturated `select` per branch point. Reduction non-determinism is a property of operations like `sum(x)` over arrays where the addition order matters at floating-point precision; our path has none.
2. **Every intermediate is an exact float.** Integers below the exact-integer bound ( $2^2$  for float32,  $2^3$  for the float64 the calc demo runs) and the values 0.0/1.0 are represented exactly in IEEE-754; integer  $+/-/\times$  of them is exact (no rounding to be reordered); element-wise multiplication of two exact floats is exact. So even if the *order* of element-wise ops differed across warp schedules, the *result* of each op is identical to the bit regardless.
3. **The saturated select multiplies off-branches by exact zero.** This does not depend on denormal-handling flags (DAZ/FTZ):  $\exp(-1000) \approx 5 \times 10^{-3}$  is far below the smallest *subnormal* of both float32 ( $\sim 1.4 \times 10^{-4}$ ) and float64 ( $\sim 4.9 \times 10^{-32}$ ), so it rounds to 0.0 whether or not subnormals are flushed; it is not a value DAZ/FTZ could change.

So these are not tolerance-band results and the measured  $|\text{err}|$  of 0.0 reproduces across runs and across hardware revisions within the IEEE-754 envelope. The scope is precise: this is exactness *for integer-valued computation in the exact range on IEEE-754 hardware*, not a claim that arbitrary float pipelines are bit-portable. In particular, the soft-halt's `sigmoid`, a transcendental, and indeed *not* bit-portable across hardware or library versions, is deliberately **outside** this claim: the termination obligation (§3.3) needs only that the halt signal is monotone and *eventually crosses* its threshold, an inequality/monotonicity property that is robust to the sigmoid's exact bits, not a bit-identity of the sigmoid. Bit-exactness is claimed for the *integer arithmetic dispatch*; termination rests on

*monotone thresholding*; the two are separate properties and neither leans on a transcendental being bit-portable. This is the §3.1 contract property in miniature: the compiled graph computes exactly what the source denotes, end-to-end on the substrate.

These are existence results for exactness on the substrates and programs measured, which is what the reduction's premise requires.

**4.4 Substrate-faithfulness: dispatch-level discharge is necessary, not sufficient.** A natural way to claim a Sutra program "runs on the substrate" is to confirm every operation dispatches to a substrate primitive, no host scalar branch, no `float()` extraction inside an op, no Python control flow on a substrate value. The repository's leak catalogue enumerates these dispatch-level breaches and which sites have been closed. Dispatch-level cleanliness is necessary, but it is not sufficient for the faithfulness claim §4 needs, three further measurements separate "every op dispatched correctly" from "the substrate carries the signal the claim asserts," and we name them here because conflating the two has been the silent failure mode caught in a substrate-purity audit of downstream Sutra programs.

- **Dimension audit.** A program can dispatch every op to the substrate but at a runtime dimension that encodes nothing, paying substrate cost for unused capacity. If a Sutra source has no `basis_vector` invocations, the semantic-codebook capacity is unused and the synthetic axes carry all the work; the runtime dimension can drop from the default of semantic + synthetic (768 + 100 on nomic-embed-text) to a small fraction with no change in observable output. A dimension audit confirms `runtime_dim` matches what the source actually needs. *Caught downstream:* every audited downstream Sutra application was at the default 768-d substrate despite zero `basis_vector` calls, a  $\sim 96\times$  over-dimensioning paid silently for weeks until the audit cut each app to the dimension its `.su` actually exercised.
- **State-locus audit.** A function that takes a scalar, returns a substrate vector, and is invoked in a host loop that extracts the scalar between calls is not a recurrent network even when every internal op dispatches to the substrate, the recurrence lives in a host variable, not on the substrate. Any claim of "recurrent" or "substrate-pure state" requires the state vector to survive across time steps without an intermediate host scalar extraction (`vsa.real(...)`-style). The state-locus audit traces where the state lives between steps. *Caught downstream:* counter and toggle demos and a font cycle-step demo were labelled as RNNs until the audit corrected the framing to "stateless substrate function in a host loop"; the rewrite to a real substrate loop carrying the hidden state as a vector is the natural fix (Sutra's `loop (cond)` lowers to a bounded substrate recurrence, §3.3).
- **Signal-separation audit.** A substrate classifier, any function whose output is a decision, can return numbers from substrate ops while those numbers fail to distinguish the classes the function is supposed to distinguish. The audit measures `gap = min(positive-class output) - max(negative-class output)` over the program's input distribution; without a positive gap the classifier is not separating the classes the dispatch claim implies. *Caught downstream:* an initial font-glyph encoding (`bundle(bind(p, LIT)) / bind(p, UNLIT)` per cell) returned LIT-cell cosines that overlapped UNLIT-cell cosines at every runtime dimension between 16 and 256, every dispatch correct, signal-separation gap negative. The corrected sparse-only encoding ships with a measured positive gap reported alongside its rendered output.

The §4 results above already discharge the third check for the substrate primitives: §4.1's multi-width capacity table is a signal-separation report (positive-class accuracy across  $k$ , negative class being the alternative codebook entries) and §4.3's  $|\text{err}| = 0.0$  is its strongest possible form. We

name the three checks here because they apply across the trusted base, not only to the substrate primitives, and the silent failure mode is treating dispatch-level cleanliness as if it were the full claim. The composition with §3 is structural: dispatch-level cleanliness keeps the obligation-checker inputs faithful (the polynomial extracted from the lowered graph is the one the substrate executes); the three measurements keep the §4 faithfulness claim sound at the program level.

**4.5 Coverage of the dispatch-level check itself: a worked failure.** §4.4 argues dispatch-level cleanliness is necessary; this subsection reports a concrete leak the dispatch sweep silently missed, and how it surfaced. To be explicit about its status: the leak sweep is an **engineering defense-in-depth guard, not a formal method**, and it is deliberately *syntactic*, and this subsection's contribution is precisely to show that a syntactic check is *necessary but not sufficient*. There is no tension with the formal results: the formal claims are §3's obligations (Kleene-polynomial range-soundness, the termination convergence check, the equivalence decision procedure); the sweep is a separate, lighter-weight CI guard against a *different* failure class (substrate-purity breaches), and we report its blind spot rather than present grep as a proof. The repository ships this automated leak sweep (wired as a CI gate) that re-emits every user `.su` program in the test corpus to Python and greps the emitted module for the banned patterns, `float(...)/.item()` on a substrate tensor, `host for/if` on a scalar, `libm` calls on values pulled off the substrate. The sweep runs across 67 user programs and asserts zero operator leaks. It returns green.

It missed a leak in the runtime prelude itself. The emitted `_TorchVSA.eq(a, b)` method computed the cosine similarity as a 0-D tensor with a live autograd chain (`cos = (av·bv) / (||av||·||bv|| + )`) and then returned `self.make_truth(float(cos.item()))`, a host scalar extraction followed by a re-wrap. The numerical value is identical to a substrate scatter, but the autograd connection is severed, and the pattern is the exact banned host-extraction-inside-an-op that the leak catalogue formalises. It survived because the sweep is over user programs' emitted Python; the `_TorchVSA` class that *is* the substrate prelude is not part of any user program's emitted output, so the sweep never reads it.

The leak surfaced downstream rather than from any check named in this paper: a constrain-train experiment compiled a Sutra source that used `==`, made the output of `==` depend on a trainable scalar parameter, and called `loss.backward()` to update the parameter. The backward pass failed with "element 0 of tensors does not require grad and does not have a grad\_fn" because the autograd chain ended inside `_TorchVSA.eq`. Tracing back through the chain, input tensor with `requires_grad=True` survives until the `make_truth(float(cos.item()))` line, then does not, located the leak in one read. The fix is a substrate-pure scatter: `out = zeros(self.dim, ...); out[truth_axis] = cos; return out`, with `cos` kept as a 0-D tensor; numerics identical, autograd preserved. After the fix, the differentiability test returned `out.requires_grad = True`, `out.grad_fn = <MulBackward0>`, and a non-None `gain.grad` after `backward()`.

The lesson for the framework is structural, not a one-off bug report. The audit's BNF-shaped leak check (a syntactic grep) has a precise blind spot: it sees the surface where user programs touch the substrate, not the substrate's own surface where it touches PyTorch. Closing the blind spot is an implementation move, and the implementation move has now shipped: the leak sweep gains a second scan pass that targets the runtime prelude (`_TorchVSA`) with a **method-level allowlist of legitimate boundaries**, the `make_*` constructors, the `_st / _as_*_vector` entry boundaries, the `real / imag / truth / similarity / argmax_cosine / select` output edges, the Promise inspectors, the JS-interop coercion methods, embedding bootstrap, and the loop machinery's structural `for` ranges. Inside any other prelude method, `float() / .item() / host scalar coercion` is flagged. Conceptually this is the right shape because the allowlist names *what each boundary is for*: an entry

boundary lifts a host literal into a substrate tensor; an output edge collapses a substrate value to a host scalar for monitoring. Both are documented host substrate edges by design. What lives *inside* an operation's definition between those edges should stay on the substrate. The leak is now recorded in the leak catalogue; the user- program sweep continues at zero leaks across the 67-program corpus; the runtime-prelude sweep also returns zero leaks after the `eq()` / `eq_synthetic` fix. A per-op differentiability unit test is an additional layer that can catch a leak even when both grep passes are clean, autograd connectivity is a semantic property the syntactic grep cannot fully discharge, but the sweep extension is now load-bearing, not aspirational. The conceptual point above the implementation: a syntactic audit family discharges a syntactic claim, and substrate faithfulness is a semantic claim. This is the positive-result side of §4.4's argument: when a dispatch-level check misses something, the §4 program-level measurements (here, an autograd-based differentiability probe of a trainable program) are what catch it.

### **A separate live evidence point that the substrate supports differentiable computation.**

A second constrain-train experiment ships in the reference implementation: `defuzz` trains a scalar `beta` parameter inside Sutra's `defuzzify_trit` operator against a polarization task, with the loss backpropagated end-to-end through the compiled tensor-op graph (3 seeds, 40 epochs, baseline loss 0.21 → trained loss 0.01, `beta*` 6.5–6.8; `experiments/defuzz_gain_adjustment.py`). The diagnostic-with-teeth here: the task surface in the first iteration of the harness was the cosine-form (`gain * v`) == `true`, which is *scale-invariant* in `gain` (cosine normalizes away positive scaling), so gradient was zero everywhere despite the autograd chain being intact, caught as a measurement, not a guess, once the `eq()` autograd fix removed the prior obstruction. Rewriting the harness to `defuzzify_trit(v, iters=1, beta=beta)` (a scale-sensitive surface) put gradient on the parameter and the experiment converged. We flag this because it is the same shape as §4.4/§4.5's broader point: a "the substrate computes the graph faithfully" claim is only as strong as the experiment that distinguishes a working channel from a failing one, and the right experiment was an end-to-end training run that actually moved a parameter, not a syntactic check that the operator dispatched.

## **5. Scope**

The reduction buys the *shape* of a certification effort, DO-178C-style: a fixed image and fixed critical-program set (Plan); axon-typed contracts (Requirements); Sutra source whose compiled graphs are the designs (Design); mechanical proofs that the graphs meet contracts plus discharged polynomial obligations (Verification artefacts); an append-only capability/admission log (Trace); and the compiler in scope for qualification with its compiled-graph output, not the source, as the artefact under review (Tooling assurance).

The scope is bounded in three ways. The method covers the **non-learned** trusted base: anything that invokes an embedding model or depends on a learned weight is outside it, and gets bounded behaviour, capability discipline, provenance, and runtime monitoring rather than a proof, the reduction makes the learned parts *quarantinable*, not *safe*. Equivalence-as-algebra and the obligation checks apply to the **contract surface of individual programs** whose compiled graphs are individually tractable, not to a closed-form whole-system proof. And a certified configuration is per-customer and per-mission; the present contribution is the framework, the reduction, and the discharged obligations.

**The frozen substrate is a foundational trust assumption, not a verified property, and that is the same posture every formally-verified system has had to take.** A formally-verified C compiler trusts the CPU's IEEE-754 unit; a verified OS trusts the silicon's MMU; a

verified bytecode interpreter trusts the machine that runs it. Sutra trusts the **frozen-substrate semantic mapping**: that `embed("cat")` returns a particular vector and that that vector's relationships to other embeddings have whatever properties the substrate provides. We do not prove the semantic mapping is correct, that would require verifying the pretrained embedding model itself, which is the learned-component verification problem we explicitly *do not* claim to solve. What we do claim: once the substrate is fixed (a particular pretrained model at particular weights, say `nomic-embed-text` at the published checkpoint), the *algebra over those embeddings*, `bind`, `unbind`, `bundle`, `similarity`, the polynomial connectives, behaves as our §3 obligations specify, measured to the precision §4 documents. The trust boundary is named: `substrate-vector identity` is foundational; everything built on top is verified or quarantined. Conflating "the substrate is trusted" with "the system is unverified" misreads where the boundary is, in the same way that "the CPU is trusted" does not invalidate the verified-compiler above it.

## 6. Related work

**Neural-network verification.** A large line verifies properties of *learned* networks: Reluplex (Katz et al. 2017) and its successor Marabou (Katz et al. 2019) extend SMT to ReLU networks; abstract-interpretation systems such as AI2 (Gehr et al. 2018) and `-CROWN` (Wang et al. 2021) bound network outputs over input regions. Our posture is orthogonal and complementary: rather than verify the learned network, Sutra verifies the **non-learned trusted base** by reduction and *quarantines* the learned part behind contracts, the two could compose, with NN-verification bounds feeding the runtime monitors Sutra places at the learned boundary.

**SMT and nonlinear arithmetic.** The obligations the compiled graph produces are polynomial, not Boolean or linear, so general-purpose SMT (Z3, de Moura & Bjørner 2008) does not apply directly; solvers for nonlinear real arithmetic such as dReal (Gao et al. 2013) are the natural backend for the *general* range/equivalence obligations, while the per-construct obligations here admit the closed-form critical-point, grid, and polynomial-identity methods of §3.

**Program specialization.** Partial evaluation and the Futamura projections (Futamura 1971) and multi-stage programming (Taha & Sheard 2000) specialise a program that still runs in a conventional model; §2 argues the compiled graph is beyond this spectrum, and beyond symbolic execution and deep-learning graph optimization.

**Arithmetic-circuit compilation (cryptography).** Compiling a program's control flow into a polynomial arithmetic circuit is a well-studied technique in zero-knowledge proofs and verifiable computation: Pinocchio (Parno, Howell, Gentry & Raykova 2013) compiles C-like programs into quadratic arithmetic programs over a finite field; Groth16 (Groth 2016) gives a succinct preprocessing-SNARK over the resulting QAP; `libsnark`, `ZoKrates`, and `Circom` are the practical compiler frontends. The mechanism is similar to ours, surface control flow becomes polynomial, but the *purpose* is different: ZK-SNARKs compile in order to *prove* program execution succinctly to a verifier without revealing inputs; we compile in order to *verify* program properties by closed-form algebra on the same graph the substrate runs. The cost surfaces also differ: ZK-SNARKs pay setup + proof time + verifier time per execution and the field is finite (mod p); we pay polynomial-identity / range-bounding wall once per equivalence check and the field is the reals embedded in IEEE-754. The shared ancestor is "compile branches into a polynomial circuit"; the divergence is what you do with the resulting polynomial.

**Vector-symbolic architectures.** The substrate primitives are VSA/HRR operations, binding, bundling, cleanup (Plate 1995; Gayler 2003; Kanerva 2009), and they have a formal foundation

we rely on rather than reinvent: the holographic-reduced-representation algebra (Plate 1995) gives binding and bundling their laws, and the capacity of bundling, how many superposed items decode correctly as a function of dimension, is characterised in the VSA literature (Frady, Kleyko & Sommer 2018; Kleyko, Rachkovskij, Osipov & Rahimi 2023). Our use of this is in §4: the obligations are algebra over operations with formal laws, and the measured result this work rests on is that *rotation* binding stays exact through bundle widths where the standard Hadamard binding collapses. The three-valued Kleene polynomial encoding of branches as a verification lever is, to our knowledge, new.

**Certification.** The plan/requirements/design/verification/trace framing follows DO-178C, the avionics software-assurance standard, adapted so the artefact under review is the compiler's tensor-graph output rather than imperative source.

## 7. Hardware formal verification: the thermodynamic compile target

Everything to this point is **software** formal verification: we verify the *compiled program*, the tensor-op graph that is Sutra's non-learned trusted base, as a mathematical object, independently of the machine that runs it. This section adds the complementary half, **hardware** formal verification: verifying that the computation is correct on the *physical substrate that executes it*. Sutra's second, energy-based compile target runs not on a deterministic tensor engine but on **thermodynamic, probabilistic-bit hardware**, the kind Extropic is building (a sparse, locally-connected grid of p-bits performing block-Gibbs sampling, with the host programming the weights and reading the result). There the computation is physical: a Sutra value is a register of spins, an operation is a *factor* (a local energy term), and the answer is the configuration the hardware *settles into*, the ground state of the gadget's energy. Verifying it means proving a property of the hardware's energy landscape, not of an emitted polynomial.

The reduction of §3 is what makes both halves one story. Turning verification into a small, fixed set of finite or closed-form obligations on the compiled object is not specific to the software target; it is portable, and on the hardware target it takes its cleanest possible form, a finite **ground-state** question. The verification is in principle hardware-portable: the same energy-gadget obligations would certify the computation on *any* sampler that minimizes the same energy. We instantiate it on the specific thermodynamic hardware we are building toward, and we regard this hardware-level verification as the more important direction the work opens, a route to computation that is correct by construction at the level of the *physics*, not only the software. The per-gadget obligation reads **”is the arithmetically-correct output the global minimum of the gadget's energy?”** rather than **”is the polynomial exact?”**, because a ground-state / lowest-energy decode is exact precisely when it is.

This is a finite question for each gadget (the spins range over  $\{-1, +1\}$ ), and finite questions are exactly where machine-checked proof is cheapest. We give Lean 4 proofs (core only, no `mathlib`) that the energy-based gadgets the backend emits have their correct output as the **strict global energy minimum**, every theorem sorry-free, depending only on `[propxt, Quot.sound]`:

- the derived **AND** gadget (biases and pairwise couplings), its output `a b` is the unique energy minimiser;
- the 3-body **XOR/parity** gadget, `x y` is the unique minimiser, which pins the *sign* of the factor (the opposite sign silently encodes XNOR, a bug we hit empirically and the proof now excludes);
- the 1-bit **full adder**, `sum a b cin` (a 4-body parity factor) and carry `MAJ(a,b,cin)` (a

pairwise factor) are jointly the strict minimiser for all inputs, so **integer addition's ground-state decode is provably exact**. A multiplier is these gates composed, so its correctness follows from theirs.

**How the gadget proofs compose to a circuit, machine-checked in general.** A complete arithmetic circuit is gadgets *wired together*, one gadget's output spin is another's input, and on the energy-based target wiring is **addition of energies**: the circuit's energy is the sum of its gadget energies over the shared spin register. Composition of the ground-state proofs is a sum-of-minimized-terms argument, and we prove it *in general* (`Composition.lean`, core Lean, no `sorry`): for any finite list of penalty terms over a shared state, if a state  $\mathbf{s}$  minimizes every term and every other state makes at least one term strictly larger at  $\mathbf{s}$ , then  $\mathbf{s}$  is the **strict** global minimum of the sum (`strict_global_min_of_terms`). Each gadget's `_strict` theorem supplies exactly these two hypotheses, its energy is uniquely minimized at its correct local output, so a circuit assembled only from verified gadgets inherits a correct strict global minimum from its parts, **for any number of gadgets, with no monolithic re-proof**. This converts the composition methodology from an informal argument into a machine-checked theorem. One subtlety the proof makes precise: a gadget's *raw* energy is not a constant-zero-at-correct quantity (its minimum value varies with the inputs), so the terms that compose are the gadgets' **proper penalties**, each raw energy shifted by its own strict minimum, so it is 0 when the gadget is satisfied and  $> 0$  otherwise. We machine-check the lemma applied to a concrete *two-gate* circuit, a 3-input AND built from two AND gadgets wired on a shared spin (`and3_circuit_strict_min`), whose correct output is the strict global energy minimum for every input, discharged from the two gadget penalties via the general lemma rather than a re-proof of the composite. The  $2 \times 2$  multiplier (AND + XOR + adder) is the larger worked gate instance, and the general lemma certifies the pattern at any size.

We also begin on *reachability*. The single-site (Glauber) block-Gibbs chain on the AND gadget's  $\{-1, +1\}^3$  state space is machine-checked **irreducible** (every state reaches every state, the configuration cube is connected, and every Glauber move has positive probability at finite  $\beta$ ) and **aperiodic** (every state has a self-loop, the conditional resampling a spin to its current value). Irreducibility and aperiodicity are *exactly* the hypotheses the classical fundamental theorem of finite Markov chains requires for a unique stationary distribution  $\pi$  and convergence to it from any start. We additionally prove that for **any** strictly-antitone weight, and the Boltzmann weight  $w(E) = e^{-\beta E}$  is strictly antitone for every  $\beta > 0$ , the strict energy-minimiser is the strict unique **mode** of  $\pi$ . So the finite chain converges (classical theorem, hypotheses now mechanised) to a stationary distribution whose unique mode is the arithmetically-correct answer.

A second, `mathlib`-backed layer pins the stationary object itself, over the reals. We machine-check (i) a general lemma that reversibility (detailed balance  $\pi(s)P(s,t) = \pi(t)P(t,s)$ ) of any finite row-stochastic kernel implies  $\pi$  is stationary; (ii) that the gadget's Gibbs kernel with the *real* Boltzmann weights  $e^{-\beta E}$  is reversible with respect to the Gibbs measure, so that measure is stationary; and (iii) two-state Perron–Frobenius **uniqueness** of the stationary distribution. With the irreducibility/aperiodicity above, this is the full reversible-chain picture: a positive, irreducible, reversible finite chain has a *unique* stationary distribution, and it is the Gibbs measure.

Each proof is a finite case analysis discharged by `omega` after a Boolean split (integer `decide` does not reduce in the kernel here). These same gadgets were independently *measured* to compute correctly at  $\sim 100\%$  on the real sampler, and the AND gadget was even *re-learned* from data by contrastive divergence, recovering the hand-derived couplings, so measurement, learning, and proof agree on the same energy landscape.

**What is now machine-checked, including the rate.** We have machine-checked the gadget energies are *correct*; the finite chain's ergodicity hypotheses (irreducible, aperiodic) and Gibbs **mode**; and, over the reals with `mathlib`, detailed balance, stationarity of the Gibbs measure, and its uniqueness. The **mixing rate** (*how fast* the chain reaches that unique stationary measure, the  $t \rightarrow \infty$  total-variation / spectral-gap statement) is now mechanised too, for the two-state clamped-decode chain the gadget inhabits. The transition matrix's second eigenvalue  $\lambda_2 = 1 - P_{f \rightarrow t} - P_{t \rightarrow f}$  is the per-step contraction factor: one step multiplies the deviation from the stationary  $\pi$  by exactly  $\lambda_2$  (`two_state_step_contraction`), so after  $n$  steps it is  $\lambda_2^n$  times the initial deviation (`two_state_geometric_mixing`) and the total-variation distance decays as  $|\lambda_2|^n$  (`two_state_tv_mixing`). Instantiated for the gadget's own single-site Gibbs kernel, which fully resamples the spin,  $\lambda_2 = 0$  exactly (`gibbs_lambda2_zero`): the chain reaches the Gibbs measure in a *single* step (`gibbs_mixes_in_one_step`; spectral gap = 1). All [`propext`, `Classical.choice`, `Quot.sound`], no sorry. So the full convergence picture for the gadget chain (the gadgets *correct*, the chain *ergodic with the right unique stationary Gibbs measure*, and now the *rate*) is machine-checked. What we do **not** claim: a mixing-rate bound for a general multi-state block-Gibbs chain (only the two-state case is mechanised), nor the continuous-time Langevin SDE limit. (Proofs: `fv-lean/`, `core`, no `mathlib`, and `fv-lean/mathlib/` for the reversibility/stationarity/uniqueness/rate layer; the measured exploration and the host/sampled hardware mapping: the companion findings.)

## 8. Source-language frontends: empirical end-to-end verification

The verification in §§3–7 is *formal*: closed-form obligation discharge (§3–§4) and machine-checked Lean proofs (§7). This is a **brief complementary note**, not a co-equal contribution, on a **separate and weaker** assurance layer we are careful not to conflate with the formal results. Beyond hand-written Sutra, the language is also a compile *target* reached from several source languages by research lowering passes, **not complete or production compilers, a claim we explicitly do not make**. Those passes are verified **empirically, by end-to-end test, not by proof**: the bar is **compile-AND-run against ground truth**. A fixture is a small source program with a known result; the pass lowers it to Sutra, the §2 compiler lowers that to the tensor-op graph, the graph runs on the substrate, and the decoded output is compared to the source language's own answer, a wrong number is a *failure*, not a pass. The fixture suite measures the *breadth of verification-relevant constructs* that reach the §3 target (conditionals/`match`  $\rightarrow$  the §3.2 branch polynomial; the recursion forms  $\rightarrow$  bounded substrate loops; algebraic data  $\rightarrow$  structured axons), not any language's full surface.

**What this does and does not establish.** It establishes that, on the inputs in the suite, each lowering preserves the source semantics through to a substrate run, a regression-grade, executable check that the pass emits *correct* Sutra. It does **not** establish a formal proof that a lowering is correct for all inputs, nor a complete compiler for any language; we make neither claim, and this layer must not be read as extending the §3–§4 obligations or the §7 Lean proofs to the frontends. The relationship to the formal layer is one-directional: because every pass emits *ordinary* Sutra and the §2 compiler is the only component that lowers to tensors, the lowered program inherits the *target-level* trusted-base properties of §§3–4 exactly as a hand-written one does; what is *not* inherited, and is supplied here only empirically, is assurance about the **lowering step itself**. Formal verification of that step (a verified frontend, in the CompCert (Leroy 2009) sense) is outside scope, and we name the present assurance as exactly what it is: empirical, not formal.

## 9. Conclusion

Compiling the non-learned trusted base to a tensor-op graph turns formal verification from imperative-path enumeration into algebra over a small fixed set of tensor graphs, with the load concentrated into three closed-form obligation families. All three have mechanical checks that run on the substrate, Kleene-gate exactness (worst error 0.0), connective range-soundness (a closed-form proof of outputs in  $[-1, +1]$ ), and loop termination, together with the kernel-enforced confinement half of the contract obligation, and a decision procedure for program equivalence over the Kleene-logic fragment that separates same-graph from logical equivalence. The premise that the compiled graph is computed exactly is borne out by measured substrate exactness, including bit-exact arithmetic dispatch through the compiled substrate. The reduction, framework, and discharged obligations are the contribution; extending the equivalence decision procedure beyond the Kleene fragment, and building the general checker that discharges an arbitrary reduced-graph obligation, are the road ahead. All of that is **software** formal verification, verifying the compiled program independent of its machine. The same reduction is target-agnostic, and §7 carries it to the complementary half, **hardware** formal verification: on Sutra's energy-based target the obligation collapses to a finite **ground-state** question about the *physical* energy landscape of thermodynamic, probabilistic-bit hardware, where the substrate physically realizes Sutra's fuzzy-by-default premise. There the gadgets' ground-states are already machine-checked correct, and the single-gadget Gibbs chain is machine-checked ergodic (irreducible, aperiodic) with a unique stationary Gibbs measure (reversibility, stationarity, and uniqueness verified over the reals in `mathlib`), and the convergence *rate* for that two-state chain is mechanised: its spectral gap is 1 ( $\lambda_2 = 0$ , the kernel fully resamples) so it reaches the Gibbs measure in one step. The road ahead there is the general multi-state spectral-gap / Langevin limit of block-Gibbs. The verification is hardware-portable in principle, but we instantiate it on the specific thermodynamic hardware we are building toward; carrying the §3 framework fully onto that hardware is, in our view, the more consequential direction this work opens, correctness established at the level of the physics, not only the software.

---

## References

- de Moura, L. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.
- Fraday, E. P., Kleyko, D. and Sommer, F. T. (2018). A Theory of Sequence Indexing and Working Memory in Recurrent Neural Networks. *Neural Computation*, 30(6).
- Futamura, Y. (1971). Partial Evaluation of Computation Process: An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5). Reprinted in *Higher-Order and Symbolic Computation*, 12(4), 1999.
- Gao, S., Kong, S. and Clarke, E. M. (2013). dReal: An SMT Solver for Nonlinear Theories over the Reals. In *International Conference on Automated Deduction (CADE)*.
- Gayler, R. W. (2003). Vector Symbolic Architectures Answer Jackendoff's Challenges for Cognitive Neuroscience. In *Joint International Conference on Cognitive Science (ICCS/ASCS)*.
- Gehr, T., Mirman, M., Drachler-Cohen, D., Tsankov, P., Chaudhuri, S. and Vechev, M. (2018). AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *IEEE Symposium on Security and Privacy (S&P)*.

- Groth, J. (2016). On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology (EUROCRYPT)*.
- Kanerva, P. (2009). Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors. *Cognitive Computation*, 1(2).
- Katz, G., Barrett, C., Dill, D. L., Julian, K. and Kochenderfer, M. J. (2017). Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification (CAV)*.
- Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D. L., Kochenderfer, M. J. and Barrett, C. (2019). The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Computer Aided Verification (CAV)*.
- Kleyko, D., Rachkovskij, D. A., Osipov, E. and Rahimi, A. (2023). A Survey on Hyperdimensional Computing aka Vector Symbolic Architectures. *ACM Computing Surveys*, 55(6).
- Leroy, X. (2009). Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7).
- Parno, B., Howell, J., Gentry, C. and Raykova, M. (2013). Pinocchio: Nearly Practical Verifiable Computation. In *IEEE Symposium on Security and Privacy (S&P)*.
- Plate, T. A. (1995). Holographic Reduced Representations. *IEEE Transactions on Neural Networks*, 6(3).
- RTCA (2011). *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*.
- Taha, W. and Sheard, T. (2000). MetaML and Multi-stage Programming with Explicit Annotations. *Theoretical Computer Science*, 248(1–2).
- Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.-J. and Kolter, J. Z. (2021). Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Neural Network Robustness Verification. In *Advances in Neural Information Processing Systems (NeurIPS)*.

---

*Reproducibility. The compiler, the obligation checker, the Lean 4 proofs, and the scripts that produce every measured number reported here are in the project repository; each result is regenerated by a named test or experiment, and the substrate-leak sweep and proof checks run under continuous integration.*