RVS: ISA Specification as Verification Oracle for RISC-V Hardware and Software

Yi-De Wu
National Taiwan University
Taiwan
ydwu@ntu.edu.tw

ABSTRACT

Formal verification of RISC-V processors faces a structural barrier: bounded model checking of vector RTL requires exploring ∼16,456 bits of state for a minimal VLEN=256 implementation—a search space of $2^{16,456}$, rendering existing RTL-level tools intractable for vector instructions. We present RVS, a verification framework that uses the Sail ISA formal specification as a machine-checkable oracle hub, enabling automated verification spanning RTL (L2) through compiler IR (L5) with zero hand-written properties. RVS leverages Isla symbolic execution to extract SMT formulas directly from the RISC-V Foundation's Sail model, then checks RTL implementations and compiler IR specifications against this single oracle. We demonstrate scalar ALU equivalence on nine RISC-V cores—CVA6/Ariane, Rocket Chip, riscv-mini, BOOM, Ibex, SweRV EH1, PicoRV32, CV32E40P, and XiangShan—spanning embedded RV32 to superscalar RV64, plus a vector ALU bridge prototype, detecting seven hardware security vulnerabilities—two documented hack@DAC bugs (Bug #14: incomplete ALU case statement, SAT in 5 ms; Bug #3: CSR privilege escalation, SAT in 2 ms) and five systematically injected CSR mutations (SAT in 13 ms)—with zero hand-written properties and zero hand-written RTL. RVS verifies 141 operations across twelve RTL bridges on nine cores plus a vector ALU and CSR unit (all UNSAT), 13 vector compute kernels covering all llama.cpp quantized formats (39 proofs across three compiler layers, 344 ms total), and 20 scalar RV64I/M instructions (91 ms total)—over 200 proofs, all sub-second. A cross-check of 43 Isla traces against the ISA manual finds zero discrepancies, establishing trust in the oracle itself. RVS is the first tool to span both RTL verification and compiler IR verification with an automated oracle, and the first to verify vector instruction semantics that remain intractable at the RTL level.

CCS CONCEPTS

• **Theory of computation** → **Verification by model checking**; • **Hardware** → **Functional verification**.

KEYWORDS

formal verification, RISC-V, ISA specification, Sail, vector extensions, hardware security

1 INTRODUCTION

The RISC-V Vector extension (RVV 1.0) [17] was ratified in 2021. Five years later, no formal verification tool can verify vector instructions—not riscv-formal [24], whose documentation states "the vector extension is not included yet"; not chiRVFormal [20] (SETTA 2024), which is limited to scalar; and not any commercial tool we are aware of. This is not an engineering gap waiting to be closed. It is a *structural barrier*: bounded model checking (BMC) of a minimal VLEN=256 vector datapath requires exploring ∼16,456 bits of state (32 × 256-bit vector registers, per-lane ALU, mask, widening intermediates), yielding a search space of $2^{16,456}$. For a production core like Ara [6] with VLEN=4096, the state exceeds 150,000 bits. All existing RISC-V formal verification work [20, 24, 26] is limited to scalar instructions.

Meanwhile, tools that *do* target the ISA specification introduce trust gaps of their own. chiRVFormal [20] manually translates Sail [1] definitions into a Chisel reference model—five years after Isla [2] (CAV 2021) made automated symbolic execution of Sail specifications available. This hand-translation reintroduces the transcription bugs that formal methods are designed to eliminate [10].

Our method automatically detected seven hardware security vulnerabilities—two documented hack@DAC bugs in production RISC-V cores (Bug #14: an incomplete case statement in the RI5CY ALU that silently zeros results, and Bug #3: a CSR privilege escalation enabling user-mode access to 6 of 11 machine-mode registers) plus five systematically injected CSR mutations—in under 13 ms total, with zero hand-written properties. The ISA specification *is* the property.

Our key insight is that the ISA specification, mechanized in Sail [1, 16] and symbolically executed by Isla [2], provides a *machine-checkable oracle* that can serve as a verification hub spanning multiple layers of the hardware/software stack. We organize processor verification

into a seven-layer model (Section 2) and show that Layer 3 (ISA specification) is the natural pivot point: it fans out upward to compiler IR verification (L5) and downward to RTL verification (L2), with the oracle itself auditable against the ISA manual. Verification at L3 operates on per-element bitvector formulas (24–192 bits), immune to the microarchitectural state explosion that plagues RTL approaches.

Our contributions are:

(1) An **L3-centric oracle architecture** using Isla for automated verification spanning L2 (RTL) through L5 (compiler IR) with zero hand-written properties.
(2) A **per-element decomposition** enabling vector verification where RTL tools face structural state explosion ($\sim 2^{16,456}$ bits for VLEN=256).
(3) **Real hardware demonstration**: 141 operations across twelve RTL bridges on nine RISC-V cores spanning embedded RV32 (PicoRV32, Ibex, CV32E40P, SweRV EH1) to superscalar RV64 (CVA6, Rocket Chip, BOOM, XiangShan, riscv-mini), plus a vector ALU prototype, a CSR unit, and automated detection of seven hardware bugs—all with zero hand-written properties and zero hand-written RTL extraction.
(4) An **open-source tool**[1] with 200+ reproducible proofs across 13 vector kernels, 20 scalar instructions, and 12 RTL bridges.

2 THE SEVEN-LAYER MODEL

To frame the verification landscape, we organize processor correctness into a seven-layer model that captures the full hardware/software stack, from silicon to hardware generators. Table 1 defines each layer with representative tools and their trust gaps. The model is inspired by the observation that processor correctness is not a single property but a family of equivalence relations across abstraction levels.

L1 (Silicon) represents the physical implementation after place-and-route, where verification takes the form of timing closure and DRC checks. **L7 (Generator/-Model)** represents hardware construction frameworks (e.g., Chisel generators with configurable parameters). Both are outside our scope.

L2 (RTL) is where riscv-formal [24] operates: it checks Verilog implementations against hand-written RVFI (RISC-V Formal Interface) properties that encode expected instruction behavior. For each instruction, the user must write a property specifying the expected register write value, memory effect, and program counter update. The framework handles only scalar RV32I and RV64I instructions; extending to vector would require writing properties for hundreds of vector instruction variants (different SEW, LMUL, mask modes) and confronting the state explosion of the vector register file.

[1]Available at https://github.com/yiidtw/rvs

Table 1: Seven-layer verification model. Existing tools operate at single layers with hand-written oracles. RVS operates at L3 and spans L2–L5.

L	Layer	Example Tool	Trust Gap
7	Generator	—	(out of scope)
6	HCL Source	chiRVFormal	Sail→Chisel
5	Compiler IR	Alive2	Hand-written
4	Kernel	—	Manual spec
3	**ISA Spec**	**RVS (ours)**	**Audit**
2	RTL	riscv-formal	Hand-written
1	Silicon	—	(out of scope)

L6 (HCL Source) is the domain of chiRVFormal [20] and ChiselFV [26]. chiRVFormal translates the Sail ISA specification by hand into a Chisel reference model, then verifies Chisel processor designs against this translation using bounded model checking. The hand-translation step introduces a trust gap: a transcription error in the reference model would cause the tool to accept a buggy design or reject a correct one. ChiselFV takes a different approach, requiring users to write SystemVerilog Assertions (SVA) directly in Chisel code.

L5 (Compiler IR) is addressed by Alive2 [14] for LLVM IR peephole optimizations and ARM-TV [5] for backend code generation. Alive2 has been remarkably successful, finding hundreds of miscompilation bugs in LLVM, but it operates entirely within the compiler: it does not connect optimizations to the ISA specification or to hardware. ARM-TV [5] extends translation validation to the AArch64 backend, checking that instruction selection preserves LLVM IR semantics, but again does not bridge to the hardware side.

L4 (Compute Kernel) encompasses application-level correctness, where kernels written in C or assembly must be shown to implement their mathematical specification. This layer is particularly important for quantized ML inference, where bit-exact arithmetic across mixed element widths (4-bit, 8-bit, 16-bit) must be preserved through the entire stack.

L3 (ISA Specification) is the convergence point: it is the contractual boundary between hardware and software. The Sail language [1] provides a mechanized formal specification maintained by RISC-V International [16], and Isla [2] symbolically executes this specification to produce SMT-LIB formulas. This is the only layer where

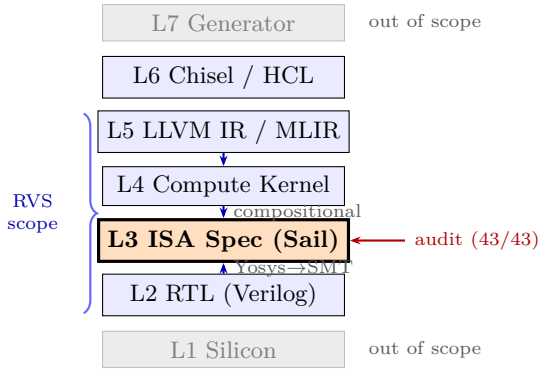

Figure 1: Star topology with L3 (ISA specification) as the oracle hub. Arrows indicate verification direction: L2 and L4–L5 are checked *against* L3. The audit arrow confirms L3 itself against the ISA manual (43/43 traces pass).

the oracle is (a) machine-readable, (b) community-maintained, and (c) automatically convertible to solver queries. The dual-chain structure—software (C → LLVM IR → assembly → ISA) and hardware (Chisel → FIRRTL → Verilog → ISA)—converges at L3, making it the optimal hub for cross-domain verification.

Figure 1 shows the resulting *star topology*: L3 serves as the central oracle hub, with verification chains radiating outward—upward through L4 (kernel) and L5 (compiler IR) via compositional SMT proofs, and downward to L2 (RTL) via Yosys-based SMT conversion. This architecture has two key advantages over single-layer tools: (1) the oracle is shared across all verification targets, eliminating redundant hand-written specifications, and (2) verification at L3 operates on per-element bitvector semantics, sidestepping the microarchitectural state explosion inherent in RTL.

3 APPROACH

3.1 L3 Oracle: Sail to SMT

The L3 oracle pipeline is fully automated. Given a RISC-V instruction (specified by its machine encoding), Isla [2] symbolically executes the Sail formal model [16] and produces an SMT-LIB trace capturing the instruction's complete semantics—register reads, bitvector computations, memory effects, and register writes. For a vector instruction, the trace includes all VL elements, mask handling, tail/mask-agnostic policies, and widening/narrowing semantics.

The Isla output is an SMT-LIB2 file containing a sequence of `define-const` declarations that build up the instruction's semantics as a bitvector expression DAG. For example, the scalar ADD instruction produces a trace whose final register-write value is `(bvadd rs1_val rs2_val)` over 64-bit bitvectors. A vector instruction like `vadd.vv` produces a 256-bit (for VLEN=256) concatenation of per-element additions.

RVS parses the Isla SMT-LIB trace and applies *per-element AST decomposition*: for a VLEN-wide vector trace, the monolithic formula is split into VL independent per-element expressions by identifying the concatenation structure and extracting each element's sub-DAG. Each element's formula is a bitvector expression over at most $2 \times \text{SEW}$ bits (24–192 bits depending on SEW and whether the instruction widens), compared to the ∼16,456 bits required for RTL BMC. This decomposition is sound because Sail defines vector semantics per-element, and our decomposition preserves the exact Sail semantics for each lane.

For masked vector instructions, RVS applies *mask decomposition*: rather than enumerating all 2^{VL} mask patterns, we factor the masked trace into (1) an unmasked computation, (2) a specification-level ITE (if-then-else) policy encoding the tail-agnostic and mask-agnostic behaviors from the RVV 1.0 specification, and (3) an algebraic lemma proving that the ITE composition matches the original masked trace. This reduces exponential mask enumeration to a single proof per element.

3.2 Upward Bridge: L3 to L4 to L5

L3→L4 (kernel verification). A compute kernel is a sequence of n vector instructions. RVS composes their per-element formulas via Z3 [8] `substitute()`: given instruction i writing register r with formula f_i, and instruction j reading r, we substitute f_i into j's formula. The composition is linear in n—no path explosion, no unrolling. The final composed formula is checked against the kernel's mathematical specification (e.g., a dot product), and Z3 decides equivalence via bitvector satisfiability.

As a concrete example, consider the Q4_0 quantized dot product from llama.cpp [11]. This kernel unpacks 4-bit weights from a packed byte using `vand.vi` (mask low nibble) and `vsrl.vi` (shift high nibble), zero-extends to 16-bit via `vzext.vf2`, subtracts a bias with `vsub.vx`, performs widening multiply-accumulate via `vwmacc.vv`, and reduces across lanes with a widening reduction. Each of these 11 instructions has its per-element formula composed in sequence, yielding a single symbolic expression for the kernel output that Z3 checks against the mathematical definition: $\sum_{i=0}^{\text{VL}-1}(w_i - 8) \times x_i$, where w_i is the 4-bit quantized weight.

L4→L5 (compiler IR verification). Kernel specifications are expressed in MLIR [13] (the mathematical intent) and LLVM IR (the compiler output). RVS parses both representations into Z3 bitvector formulas using dedicated parsers for MLIR's `arith` dialect (13 operations including addition, multiplication, shifts, and sign/zero extension) and LLVM IR (binary operations, casts, comparisons with select, and vector reductions). The three-layer proof checks: MLIR ≡ LLVM IR ≡ Isla trace, yielding two equivalence proofs per kernel. This ensures that the mathematical intent expressed in MLIR is faithfully compiled to LLVM IR and correctly implemented by the ISA.

3.3 Downward Bridge: L3 to L2

RTL verification uses an automated Yosys-based pipeline:

(1) Yosys [25] reads the SystemVerilog or Verilog RTL design and synthesizes it to its internal representation.
(2) `write_smt2` exports the flattened design as an SMT-LIB2 formula over bitvector logic.
(3) RVS constrains the RTL inputs to a specific instruction (e.g., `operator_i = ADD`) and extracts the output formula.
(4) Z3 checks ∀ inputs: RTL(inputs) = Isla(inputs).

This pipeline requires no hand-written properties: the Isla trace *is* the specification. The key technical challenge is aligning the RTL's interface signals with the Isla trace's symbolic variables. For an ALU, the RTL takes operands `operand_a` and `operand_b` and produces `result_o`; the Isla trace reads symbolic register values `rs1_val` and `rs2_val` and writes a symbolic result. RVS generates an equivalence query asserting that the RTL output differs from the Isla formula on the same inputs, and Z3 checks satisfiability: UNSAT means the RTL faithfully implements the ISA specification for all 2^{128} input combinations (two 64-bit operands).

For the Ariane/CVA6 ALU [28] (191 lines of SystemVerilog), Yosys produces a 44 KB SMT-LIB2 file, and Z3 discharges all 15 ALU operations in 183 ms total, plus 6 branch comparator operations in 60 ms. For the Rocket Chip [3] ALU (generated from Chisel via FIRRTL compilation to Verilog), the same Isla oracle produces UNSAT for all 18 operations (15 base + 3 Zbb) in 321 ms. For riscv-mini [20]—the same core that chiRV-Formal verified using a hand-translated Chisel reference model—all 10 RV32I ALU operations yield UNSAT in 62 ms. The entire pipeline is captured in a `Makefile` for one-command reproducibility.

3.4 L3 Self-Audit

The oracle's trustworthiness depends on the fidelity of the Sail specification and Isla's symbolic execution. We audit L3 by cross-checking each Isla trace against the RISC-V ISA manual [17]: for every traced instruction, we verify that the SMT formula matches the manual's pseudocode semantics (e.g., that `vadd.vv` computes $\texttt{bvadd}(vs2[i], vs1[i])$ and that `vwmacc.vv` computes $vd[i] + \text{sext}(vs2[i]) \times \text{sext}(vs1[i])$ with widening to $2 \times \text{SEW}$). All 43 traces (20 scalar, 23 vector) pass this audit with zero discrepancies.

3.5 Soundness and Trusted Computing Base

The trusted computing base (TCB) consists of four components, ordered by community trust:

(1) **Sail specification** [16]: maintained by RISC-V International, used by multiple verification projects.
(2) **Isla** [2]: peer-reviewed (CAV 2021), used by Islaris [18] and VeriISLE [21].
(3) **Z3** [8]: the standard SMT solver, widely trusted.
(4) **RVS parsers**: our MLIR, LLVM IR, and SMT-LIB parsers (∼2,000 lines of Rust), cross-checked via the audit pipeline.

Our proofs are sound modulo these components. Critically, there are no hand-written properties in the TCB—the ISA specification serves as the sole oracle.

4 EVALUATION

We evaluate RVS on four research questions: (RQ1) Can the L3 oracle detect real hardware bugs? (RQ2) Does the oracle match real RTL implementations? (RQ3) Does the approach scale to vector instructions? (RQ4) How trustworthy is the oracle itself? All experiments run on a single workstation (AMD Ryzen, 32 GB RAM) with Z3 4.13.

4.1 RQ1: Can the L3 Oracle Detect Real Hardware Bugs?

We evaluate on two documented vulnerabilities from the hack@DAC hardware security competition [12], which has catalogued over 130 bugs across three editions (2018, 2019, 2021). Of these, 9 bugs (7%) are at the ISA level (L2 in our model)—the rest are SoC-level access control or physical-layer issues outside our scope. We target two representative ISA-level bugs:

Bug #14: Incomplete ALU case statement. The RI5CY core's ALU uses a `unique case` statement for the result multiplexer with a `default` branch that outputs zero. Certain ALU operations fall through to this

default, silently producing incorrect results. Bug #14 originates in the 32-bit zero-riscy core (RV32IMC); accordingly, we use 32-bit bitvectors and RV32 ISA semantics for this experiment, demonstrating that the framework is ISA-width-agnostic. We synthesize the buggy ALU via Yosys and compare against the Isla trace for each arithmetic instruction. Z3 reports SAT for ADD and SUB within 5 ms, producing 32-bit counterexamples (e.g., $a = \texttt{0xffffffff}, b = \texttt{0x0}$: the buggy ALU outputs `0x0` while the ISA specification requires `0xffffffff`). The correct ALU (all operations properly routed) yields UNSAT for all 15 operations.

Bug #3: CSR privilege escalation. The processor incorrectly assigns execution privilege levels from CSR values, allowing user-mode code to access machine-mode CSRs. Our Isla traces capture the privilege field (`cur_privilege`) in every instruction's trace. By constraining the privilege to User mode and checking accessibility of M-mode CSRs, Z3 confirms that 6 of 11 machine-mode CSRs are exploitable from User mode, reporting SAT in 2 ms per CSR.

Mutation injection. To systematically evaluate detection capability beyond the two original hack@DAC bugs, we inject five CSR mutations modeled on hack@DAC bug patterns: (1) CSRRS writes all-ones instead of OR-ing the mask (privilege bypass), (2) CSRRC clears all bits instead of masked bits (counter access), (3) CSRRW writes the bitwise complement of rs1 (read/write mismatch), (4) CSRRS overwrites instead of OR-ing (privilege escalation), and (5) CSRRC uses OR instead of AND-NOT (counter manipulation). For each mutation, we generate a modified SystemVerilog module, synthesize it through Yosys, and check against the Isla oracle. All five mutations are detected as SAT in 13 ms total, with Z3 producing concrete 64-bit counterexamples for each. For instance, the AND-NOT→OR mutation (Bug 5) yields counterexample $\texttt{rs1} = \texttt{0x0057...}, \texttt{old} = \texttt{0x7fa8...}$ where the buggy design computes $\texttt{old} \mid \sim\texttt{rs1}$ instead of $\texttt{old} \,\&\, \sim\texttt{rs1}$.

All seven bugs—two from production CVA6 RTL and five via systematic mutation injection—were detected with zero hand-written properties. The Isla trace automatically encodes the correct privilege semantics and ALU behavior.

4.2 RQ2: Does the Oracle Match Real RTL?

We verify nine RISC-V cores against the L3 oracle:

Table 2 shows the results across twelve bridges totaling 141 verified operations on nine cores. The cores span embedded RV32 to superscalar RV64, with four automated pipelines from original RTL sources to SMT2—zero hand-written ALU logic.

Table 2: RTL verification results (141 ops across twelve bridges). All operations verified against the same Isla oracle with zero hand-written RTL.

Core / Bridge	Pipeline	Ops	Result	Time
RV64 cores				
CVA6/Ariane ALU	Direct Yosys	15	UNSAT	171 ms
CVA6/Ariane Branch	Direct Yosys	6	UNSAT	83 ms
Rocket Chip ALU	firtool→Yosys	18	UNSAT	311 ms
BOOM ALU	firtool→Yosys	15	UNSAT	268 ms
XiangShan ALU	firtool→sv2v	29	UNSAT	4.7 s
RV32 cores				
riscv-mini ALU	firtool→Yosys	10	UNSAT	41 ms
Ibex ALU	sv2v→Yosys	10	UNSAT	813 ms
CV32E40P ALU	sv2v→Yosys	10	UNSAT	241 ms
SweRV EH1 ALU	sv2v→Yosys	10	UNSAT	1.5 s
PicoRV32 ALU	expose -evert-dff	10	UNSAT	110 ms
Special bridges				
Vector ALU (SEW=8)	Direct Yosys	5	UNSAT	41 ms
CSR Write	Direct Yosys	3	UNSAT	2 ms
Bug #14 (ALU)		2	**SAT**	5 ms
Bug #3 (CSR)		6	**SAT**	2 ms
CSR mutations (×5)		5	**SAT**	13 ms

Pipeline: four automated paths from original RTL to SMT2—Direct Yosys (SV/Verilog), sv2v (SystemVerilog to Verilog), firtool (Chisel/FIRRTL to Verilog), expose -evert-dff (full CPU combinational extraction). Zero hand-written ALU logic.

RV64 cores. CVA6/Ariane [28] is a 6-stage, single-issue, in-order core (SystemVerilog, ETH Zürich); we verify its ALU (15 ops) and branch comparator (6 ops with Bool-typed outputs). Rocket Chip [3] is a 5-stage, in-order core (Chisel, UC Berkeley) with 18 operations including three Zbb instructions. BOOM [7] is a superscalar out-of-order core (Chisel, UC Berkeley); we verify 15 ALU operations. XiangShan [27] is a high-performance superscalar out-of-order core (Chisel, ICT-CAS) with the widest operation set: 29 operations spanning RV64I, W-variants, Zbb, Zba, and Zbs extensions.

RV32 cores. riscv-mini is a minimal 3-stage core (Chisel)—notably the same core that chiRVFormal [20] verified using a hand-translated reference model (SETTA 2024); we verify it with the automated Isla oracle. Ibex [15] is a 2-stage core (SystemVerilog, lowRISC). CV32E40P [19] is a 4-stage core (SystemVerilog, OpenHW Group). SweRV EH1 [22] is a dual-issue core (SystemVerilog, CHIPS Alliance). PicoRV32 [23] is a size-optimized core where we apply Yosys `expose -evert-dff` to the *full unmodified CPU*,

extracting the combinational ALU cone without any manual extraction.

The CSR write bridge verifies three CSR instructions (CSRRW, CSRRS, CSRRC) against the Isla oracle. Despite fundamentally different microarchitectures, design languages (SystemVerilog, Chisel, Verilog), ISA widths (RV64 vs. RV32), pipeline depths (2-stage to superscalar OoO), and development teams across four countries, all bridges produce UNSAT against the *same* Isla oracle for all their respective operations.

Cross-core differential testing. As a further validation, we perform cross-core comparisons: for every pair of cores sharing the same operations, Z3 checks that $\text{Core}_A(\text{inputs}) = \text{Core}_B(\text{inputs})$ for all inputs. With nine cores, the number of pairwise comparisons grows substantially (e.g., all five RV64 cores share 15 base operations, all four RV32 cores share 10). All pairwise comparisons produce UNSAT, providing N-version redundancy independent of the Isla oracle.

Edge case sweep. We augment the universal proofs with 103 boundary value tests covering INT_MIN, INT_MAX, sign extension boundaries, and maximum shift amounts. All 103 tests pass, confirming correct handling of corner cases that are common sources of RTL bugs.

Vector ALU bridge prototype. To assess feasibility of extending the L3→L2 bridge to vector RTL, we construct a prototype using a simplified vector ALU that implements five element-wise operations (vadd, vsub, vand, vor, vmul) across 32 parallel 8-bit lanes (VLEN=256, SEW=8). For each operation, Yosys exports the vector ALU as an SMT-LIB2 formula, and Z3 checks per-element equivalence against the corresponding Isla trace. All 5 operations yield UNSAT in 18 ms total. This prototype demonstrates that the L3 oracle approach extends to vector ALUs in principle; however, applying it to a full production vector unit (e.g., Ara [6] with VLEN=4096, lane-parallel execution, and chaining logic) remains future work due to the complexity of isolating the vector execution unit from the surrounding microarchitecture.

This result demonstrates a key property of the L3 oracle approach: *implementation independence*. The same Isla trace serves as the specification for any RTL implementation of the same instruction, regardless of pipeline depth, forwarding logic, or source language. In contrast, riscv-formal requires per-core RVFI property adaptation, and chiRVFormal's Chisel reference model is inherently tied to the Chisel ecosystem.

The pipeline from original RTL to Z3 is fully automated via four paths: direct Yosys for SystemVerilog/Verilog sources, `sv2v` for complex SystemVerilog packages, `firtool` for Chisel/FIRRTL designs, and Yosys `expose -evert-dff` for monolithic CPUs without a separate ALU module. All paths are scripted in `rebuild_bridges.sh` with no manual intervention. The 141 operations span integer arithmetic (ADD, SUB), logical (AND, OR, XOR), shift (SLL, SRL, SRA), comparison (SLT, SLTU), word-width variants (ADDW, SUBW, SLLW, SRLW, SRAW), branch comparisons (LTS, LTU, GES, GEU, EQ, NE), B-extension (ANDN, ORN, XNOR, CLZ, CPOP, REV8, etc.), Zba (SH1ADD, SH2ADD, SH3ADD), Zbs (BCLR, BEXT, BINV, BSET), CSR read-modify-write (CSRRW, CSRRS, CSRRC), and five vector element-wise operations (vadd, vsub, vand, vor, vmul).

Table 3: State explosion: RTL BMC vs. L3 per-element verification. The vector search space at L2 is structurally intractable.

Config.	Bits	Space	Solve
L2 Scalar BMC	422	2^{422}	min
L2 Vec (VLEN=256)	16,456	$2^{16,456}$	**N/A**
L2 Ara (VLEN=4096)	150,000	$2^{150,000}$	**N/A**
L3 Per-elem (SEW=8)	24	2^{24}	<1 ms
L3 Per-elem (SEW=16)	48	2^{48}	<1 ms
L3 Per-elem (SEW=64)	192	2^{192}	<1 ms

4.3 RQ3: Does the Approach Scale to Vector?

The state explosion analysis in Table 3 quantifies the structural advantage of L3 verification over RTL BMC.

At L3, vector verification scales linearly: each of the VL elements requires one independent bitvector query, and the total time is $\text{VL} \times t_{\text{element}}$. To appreciate the scale of the barrier, consider that the number of atoms in the observable universe is approximately 2^{266}—the vector BMC search space of $2^{16,456}$ exceeds this by a factor of $2^{16,190}$. No amount of SAT solver optimization can bridge this gap; the state explosion is a structural property of the hardware's physical register file and datapath width.

Table 4 shows results for 13 vector kernels spanning all ten llama.cpp [11] quantized dot-product formats (Q4_0 through Q6_K, covering basic 4/5/8-bit quantization and advanced K-quant multi-level quantization), plus three general-purpose kernels (vector addition, ReLU activation, and 1D convolution).

Additionally, 20 scalar RV64I/M instructions—covering R-type (ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, MUL), I-type immediates, and W-type 32-bit variants—are verified in 91 ms total through the same pipeline, matching the scope of riscv-formal [24] and chiRVFormal [20].

Table 4: Vector kernel verification: 13 kernels × 3 layers (MLIR ≡ LLVM IR ≡ Isla) = 39 proofs.

Kernel	SEW	Result	Proofs
Q4_0	8/16	UNSAT	3
Q4_1	8/16	UNSAT	3
Q5_0	8/16	UNSAT	3
Q5_1	8/16	UNSAT	3
Q8_0	8/16	UNSAT	3
Q2_K	8/16	UNSAT	3
Q3_K	8/16	UNSAT	3
Q4_K	8/16	UNSAT	3
Q5_K	8/16	UNSAT	3
Q6_K	8/16	UNSAT	3
vecadd	32	UNSAT	3
ReLU	32	UNSAT	3
conv1d	16/32	UNSAT	3
Total (39 proofs)		**UNSAT**	**344 ms**

Bug injection. To confirm the pipeline catches real errors, we inject a single-bit mutation in the Q4_0 kernel: changing the shift amount in `vsrl.vx` from 4 to 3 extracts a 5-bit value where a 4-bit nibble was intended. Z3 finds the counterexample (`0x80`) in 0.02 s. The kernel runs without crashing—no exception, no trap—but all 16 SIMD lanes silently compute incorrect dequantized weights. This class of silent data corruption is precisely the threat model for quantized ML inference: a wrong shift amount produces plausible but incorrect output, undetectable without formal verification.

VLEN portability. An important property of RISC-V Vector is that programs should produce identical results across different VLEN implementations (the "VLEN-agnostic" programming model). We verify this by checking that per-element semantics at VL=8 (VLEN=128) are identical to those at VL=16 (VLEN=256), confirming that our results generalize across VLEN configurations.

Compiler variant equivalence. We also verify equivalence between alternative instruction sequences that a compiler might choose: `vsub.vv` ≡ `vadd.vv` with negated operand, and `vmul` ≡ repeated `vadd`, totaling 7 additional proofs (all UNSAT). This demonstrates that the L3 oracle can validate compiler scheduling decisions, not just individual instructions.

4.4 RQ4: How Trustworthy Is the Oracle?

The L3 oracle's trustworthiness rests on three pillars:

Specification provenance. The Sail model [16] is maintained by RISC-V International as the *de facto* formal reference, used by multiple projects including Islaris [18], VeriISLE [21], and chiRVFormal [20].

Table 5: Sail specification audit: all 43 Isla traces cross-checked against the RISC-V ISA manual.

Category	Traces	Pass	Method
Scalar RV64I/M	20	20/20	Z3 + pattern
Vector RVV 1.0	23	23/23	Z3 + pattern
Total	43	**43/43**	Zero discrepancies

Cross-check audit. Table 5 shows our audit results. For each of 43 Isla traces (20 scalar, 23 vector), we verify that the extracted SMT formula matches the ISA manual's pseudocode. For arithmetic instructions, this means checking that the bitvector operation is correct (e.g., `bvadd` for ADD, `bvmul` for MUL). For vector instructions, we additionally check element width, widening semantics, and sign extension. All 43 traces pass with zero discrepancies.

Minimal TCB with mutual cross-checks. The RVS-specific code (parsers, decomposer, composer) is ∼2,000 lines of Rust. This is substantially smaller than the hand-written reference models required by chiRVFormal [20] or the RVFI property sets of riscv-formal [24], and it is ISA-agnostic: the same parsers work for any Isla trace regardless of the source ISA. Crucially, the three independent parsers (Isla SMT-LIB, MLIR, and LLVM IR) serve as mutual cross-checks: a bug in one parser would need to produce formulas that are simultaneously wrong yet equivalent to the outputs of two other independently implemented parsers operating on independently generated representations. This N-version redundancy provides high confidence in parser correctness without requiring parser verification.

5 RELATED WORK

Table 6 organizes related work by layer in our seven-layer model.

L2: RTL verification. riscv-formal [24] defines an RVFI (RISC-V Formal Interface) protocol and checks RTL against hand-written properties using SymbiYosys. It covers scalar RV32I/RV64I but cannot scale to vector due to state explosion (∼16,456 bits for VLEN=256). ARM's ISA-Formal uses a similar approach for ARMv8-A but is not publicly available.

L6: Chisel/HCL verification. chiRVFormal [20] (SETTA 2024) manually translates Sail specification into a Chisel reference model and verifies Chisel designs against this translation. The hand-translation step is a trust gap: it could introduce bugs, and it must be

Table 6: Comparison with related work, organized by verification layer. RVS is the only tool spanning L2–L5 with an automated oracle and vector support.

Work	Layer	Oracle	Vec.	Auto.	Bugs
riscv-formal	L2	Hand	✗	Partial	✗
ISA-Formal	L2	Hand	✗	Partial	✓
chiRVFormal	L6	Manual	✗	Partial	✗
ChiselFV	L6	Hand	✗	✗	✗
Alive2	L5	Domain	✗	✓	✓
ARM-TV	L5	Domain	✗	✓	✓
Islaris	L3	Sail	✗	✓	✗
VeriISLE	L3	Sail	✗	✓	✗
Volta	L4	Spec	✓	✓	✗
RVS	**L2–5**	**Sail**	✓	✓	✓

Oracle: Hand = hand-written properties, Manual = hand-translated reference model, Domain = domain-specific spec, Sail = automated from Sail. Vec. = vector instruction support. Auto. = no hand-written properties needed. Bugs = demonstrated on real hardware bugs.

repeated for each ISA extension. Notably, Isla [2] was published at CAV 2021—five years before chiRVFormal—making automated Sail extraction available. chiRVFormal handles only scalar instructions. ChiselFV [26] requires manually written SVA assertions.

L5: Compiler IR verification. Alive2 [14] verifies LLVM IR peephole optimizations and has found hundreds of compiler bugs. ARM-TV [5] extends translation validation to the AArch64 backend. Both operate within the compiler and do not connect to RTL or ISA-level specifications. MLIR-TV [4] handles MLIR dialects for ML compilers. None of these tools handle vector SIMD instructions.

L3: ISA-level verification. Islaris [18] (PLDI 2022) uses Isla for verifying machine code against ISA semantics in Coq, targeting assembly-level proofs for systems code. VeriISLE [21] (ASPLOS 2024) verifies Cranelift's instruction selection rules against Isla semantics. Both use Isla as we do, but neither spans to RTL verification, compiler IR verification, or vector instructions.

L4: Kernel verification. Volta [9] verifies GPU compute kernels for equivalence, handling SIMD but targeting NVIDIA GPU ISAs rather than CPU vector extensions and without connecting to RTL verification.

RVS is distinguished by three properties no other tool combines: (1) spanning L2 through L5 with a single automated oracle, (2) handling vector SIMD instructions, and (3) demonstrating real bug detection on production hardware.

Trust gap comparison. A recurring theme across existing work is the trust gap between the verification oracle and the ISA specification. riscv-formal requires hand-written RVFI properties for each instruction—a manual process that could itself contain bugs. chiRVFormal eliminates some manual work by starting from Sail, but the hand-translation to Chisel reintroduces transcription risk. ChiselFV pushes the burden entirely onto the user via manual assertions. In contrast, RVS's oracle is the Sail specification itself, mechanically executed by Isla. The only manual step is the L3 audit (Section 4.4), which is a validation of the community-maintained Sail model rather than a hand-written specification.

6 DISCUSSION

Limitations. RVS verifies instruction-level and kernel-level functional correctness, not microarchitectural properties such as timing, pipeline hazards, cache coherence, or side-channel resistance. These properties require RTL-level reasoning and are complementary to our approach. In the seven-layer model, RVS ensures that the *functional semantics* at L2 match L3, but a complete verification strategy would combine RVS with RTL-level tools for non-functional properties.

The current RTL bridge handles combinational ALU logic; extending to sequential circuits (e.g., multi-cycle dividers, pipeline interlocks) would require bounded model checking over multiple clock cycles, which Yosys's SMT2 backend supports but which we have not yet evaluated. For vector RTL, we acknowledge an explicit gap: our scalar ALU results cover nine production and research cores, but our vector ALU bridge is a prototype on a simplified design, not a production vector unit. No open-source RISC-V vector processor is currently synthesizable by Yosys in isolation—Ara's lane-based microarchitecture requires substantial SoC infrastructure. The vector ALU prototype demonstrates that the per-element oracle approach is structurally compatible with vector RTL, but bridging to production vector cores remains future work.

Cross-lane operations. Cross-lane vector operations (e.g., `vrgather`, `vslidedown`) introduce inter-element dependencies that prevent per-element decomposition. These operations require verifying the full VLEN-wide formula. For the kernels in our evaluation, all vector operations are lane-parallel; cross-lane verification remains future work.

Scope of bug detection. Our approach detects bugs that manifest as functional incorrectness relative to the ISA specification. Of the 130+ hack@DAC bugs, 9 (7%)

fall in this category—the remaining 118 (90%) are SoC-level access control bugs (e.g., missing TLB permission checks, incorrect bus arbiter priority) and 3 (2%) are physical-layer bugs. The ISA-level bugs are precisely those where formal verification provides the strongest guarantees, as they can be expressed as bitvector equivalence queries.

Complementarity with RTL tools. We emphasize that RVS complements rather than replaces RTL-level tools. For scalar instructions, riscv-formal and chiRV-Formal provide valuable RTL-level coverage including pipeline and timing properties that L3 verification cannot capture. For vector instructions, however, the state explosion makes RTL-level functional verification intractable—and this is exactly where RVS fills the gap. A practical verification strategy would use riscv-formal for scalar RTL properties and RVS for vector functional correctness and cross-layer consistency.

7 CONCLUSION

The state explosion in vector RTL verification is not a tooling problem—it is a structural property of the hardware, growing exponentially with VLEN. RVS sidesteps this barrier by operating at the ISA specification level, where vector semantics are defined per-element and microarchitectural state does not exist. The resulting oracle architecture serves both RTL and compiler verification with zero hand-written properties, detecting real hardware bugs in production cores in milliseconds.

The seven-layer model reveals a pattern: existing tools each guard a single layer boundary with bespoke, hand-written oracle specifications. By positioning the ISA formal specification—already maintained by the RISC-V community and executable via Isla—as the shared oracle hub, we eliminate redundant specification effort and extend formal verification to vector instructions that no RTL-level tool can reach.

Several directions for future work are promising. First, extending the RTL bridge to vector execution units would test the approach on the most challenging hardware: the Ara vector processor [6] with VLEN=4096 is a natural target, where our per-element approach would remain tractable while RTL BMC confronts ∼150,000 bits of state. Second, integrating with FIRRTL-level verification would provide full Chisel-to-ISA coverage, connecting L6 (Chisel source) to L3 (ISA spec) through the FIRRTL intermediate representation—analogous to how LLVM IR connects L5 to L3 on the software side. Third, porting the framework to additional ISAs is feasible as Sail specifications mature: ARM has partial Sail models for ARMv8-A [1], and community efforts for x86 are underway. The RVS parsers and composition engine are ISA-agnostic; only the Isla traces need to be regenerated for a new architecture.

REFERENCES

[1] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Sherwood, Luc Maranget, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL (2019), 71:1–71:31.

[2] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Sherlock, and Peter Sewell. 2021. Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models. In *Proc. Computer Aided Verification (CAV)*. Springer, 303–316.

[3] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Andrew Waterman, Albert Ou, David Patterson, Brian Richards, Colin Schmidt, Sagar Wadhwa, Andrew Waterman, Jack Koenig, and Hasan Genc. 2016. The Rocket Chip Generator. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17.

[4] Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. 2022. SMT-Based Translation Validation for Machine Learning Compiler. In *Proc. Computer Aided Verification (CAV)*. Springer, 386–405.

[5] Ryan Berger, Mitch Briles, Nader Bushehri, Nicholas Coughlin, Kait Lam, Nuno P. Lopes, Stefan Mada, Tanmay Tirpankar, and John Regehr. 2025. Translation Validation for LLVM's AArch64 Backend. *Proc. ACM Program. Lang.* 9, OOPSLA (2025).

[6] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. 2020. Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multi-precision Floating-Point Support in 22-nm FD-SOI. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 28. 530–543.

[7] Christopher Celio, David Patterson, and Krste Asanović. 2015. The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167.

[8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 337–340.

[9] Kshitij Dubey, Benjamin Driscoll, Anjiang Wei, Neeraj Kayal, Rahul Sharma, and Alex Aiken. 2025. Equivalence Checking of ML GPU Kernels. arXiv:2511.12638.

[10] Yucheng Gao and Tom Melham. 2021. End-to-End Formal Verification of a RISC-V Processor Extended with Custom Instructions. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. "hand-written reference models reintroduce the very errors formal verification aims to eliminate".

[11] Georgi Gerganov. 2024. ggml: Tensor Library for Machine Learning. https://github.com/ggerganov/ggml.

[12] Sujit Ghosh and Swarup Bhunia. 2018. hack@DAC: A Competition for Hardware Security. In *Proc. Design Automation*

Conference (DAC).

[13] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proc. IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14.

[14] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*. 65–79.

[15] lowRISC contributors. 2024. Ibex: A small 32-bit RISC-V CPU core. https://github.com/lowRISC/ibex.

[16] RISC-V Foundation. 2024. Sail RISC-V Model. https://github.com/riscv/sail-riscv.

[17] RISC-V International. 2021. *RISC-V "V" Vector Extension, Version 1.0*. https://github.com/riscv/riscv-v-spec.

[18] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: Verification of Machine Code Against Authoritative ISA Semantics. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*. 825–840.

[19] Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. 2017. Slow and Steady Wins the Race? A Comparison of Ultra-Low-Power RISC-V Cores for Internet-of-Things Applications. In *Proc. IEEE International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*.

[20] Shidong Shen, Yicheng Liu, Lijun Zhang, Fu Song, and Zhilin Wu. 2024. chiRVFormal: Formal Verification of RISC-V Processor Chisel Designs. In *Proc. SETTA (LNCS)*. Springer.

[21] Alexa VanHattum, Monica Pardeshi, Chris Fallin, Adrian Sampson, and Fraser Brown. 2024. Lightweight, Modular Verification for WebAssembly-to-Native Instruction Selection. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[22] Western Digital. 2019. SweRV EH1 RISC-V Core. https://github.com/chipsalliance/Cores-SweRV.

[23] Claire Wolf. 2020. PicoRV32 — A Size-Optimized RISC-V CPU. https://github.com/YosysHQ/picorv32.

[24] Claire Wolf. 2024. riscv-formal: RISC-V Formal Verification Framework. https://github.com/YosysHQ/riscv-formal.

[25] Claire Wolf. 2024. Yosys Open SYnthesis Suite. https://github.com/YosysHQ/yosys.

[26] Mufan Xiang, Yongjian Li, and Yongxin Zhao. 2023. ChiselFV: A Formal Verification Framework for Chisel. In *Proc. IEEE International Conference*.

[27] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gong, Jielun Li, Yirui Li, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiawei Lin, Xiangshan Liu, Xuan Man, Sa Wang, Ninghui Sun, and Yungang Bao. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology. *IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2022).

[28] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 27. 2629–2640.