CVA6 CV-X-IF Report (Standard Reference)

Deterministic encoding validation with ExaVerif (ev)

SSCCS Foundation

2026-08-06

ExaVerif performed exhaustive verification of the CVA6 core's offloadable custom instruction encodings for the CV-X-IF coprocessor interface. The verified encoding space is derived from the CVA6 hardware decoder mask table (cvxif_instr_pkg.sv and instr_decoder.sv) at commit 6544a714c, the industry-standard reference for what the core accepts: NOP at funct3=000/funct7=0, and the ADD family at funct3=001 with funct7 in {0,1,2,3,4}. The standard pipeline evaluates all 33,554,432 field-value combinations in 18.0 seconds (release build, single Apple silicon core), yielding 196,608 valid combinations. The Tagma-based structural enumeration evaluates the same space in 18.5 milliseconds by enumerating only the combinations that satisfy the cross constraints, a ~970x improvement on this machine. The structural iterator emits exactly the combinations that pass the full constraint set. The Spike backend runs a C reimplementation of the constraint model and assembler under Spike + pk; all 196,608 rows agree between the C and Rust implementations. Spike does not decode the custom-3 instructions: they are illegal in the base ISA by design, which is why CVA6 offloads them. The CVA6 DV class generates additional encodings that the reference coprocessor decoder does not accept; this divergence is documented in this report.

Code	Benchmarks	Fixture	References
Github	bench.rs	CVA6 XIF spec	ExaVerif
References	References	Other Formats	
neXus	OpenHW	HTML	

Verification Target

The CVA6 core (Ariane, OpenHW Group) offloads custom instructions to a coprocessor through the CV-X-IF interface whenever the core decoder encounters an illegal instruction. Four RISC-V custom opcode spaces (`custom-0` through `custom-3`) are always illegal and therefore always offloadable. This verification covers the `custom-3` opcode (`0x7B`) encoding space, using the CVA6 hardware decoder as the authoritative reference (the industry-standard definition of what the core accepts).

The hardware-accepted custom-3 encodings are:

funct3	funct7	Instruction	writeback	register_read
0	0	NOP	0	000
1	0	ADD	1	011
1	1	DOUBLE_RS1	1	001
1	2	DOUBLE_RS2	1	010
1	3	ADD_MULTI	1	011
1	4	ADD_RS3_R	1	111
other	any	illegal	-	-

Standard reference: the hardware decoder mask table

The decoder matches by `(mask & instr) == pattern` (`instr_decoder.sv`), where a mask bit of 1 means the corresponding bit is checked. The entries in `cva6/core/cvxif_example/include/cvxif_instr_pkg.sv` at commit `6544a714c`:

entry	funct7 pattern	funct3	opcode	mask funct7
NOP	0000000	000	0x7B	1111111 (all bits checked)
ADD	0000000	001	0x7B	1111111
DOUBLE_RS1	0000001	001	0x7B	1111111
DOUBLE_RS2	0000010	001	0x7B	1111111
ADD_MULTI	0000011	001	0x7B	1111111
ADD_RS3_R	0000100	001	0x7B	1111111

Because the NOP mask checks all funct7 bits, funct3=000 accepts funct7=0 only. When no entry matches, `instr_decoder.sv` keeps `accept=0`, so the offloaded instruction is treated as illegal by the core.

DV class divergence

The verification suite's DV class (`cvxif_custom_instr.sv`) generates additional encodings that the reference coprocessor decoder does NOT accept:

DV encoding	funct3	funct7	decoder accepts?
CUS_NOP	001	0	yes (as ADD semantics)
CUS_ADD	001	0	yes (as ADD)
CUS_U_ADD	000	2	no
CUS_S_ADD	000	6	no
CUS_ADD_MULTI	000	8	no
CUS_ADD_RS3	000	0	yes (as NOP)
CUS_EXC	010	96	no

This divergence is a finding of this verification: the DV class and the reference coprocessor decoder are not in agreement in current CVA6 main. This report verifies the decoder-accepted space (the industry-standard reference); the DV-generated encodings that the decoder rejects are excluded and reported here as a divergence.

Encoding Model (the YAML fixture)

The verification space is described by a single YAML file. The full-space fixture (`tests/fixtures/cva6/xif_ref.xif.yaml`) defines five fields:

Field	Domain	Size	Meaning
funct3	0..7	8	primary opcode selector
funct7	0..127	128	secondary opcode selector
rs1	0..31	32	source register 1
rs2	0..31	32	source register 2
rd	0..31	32	destination register

The raw space is 8 x 128 x 32 x 32 x 32 = 33,554,432 combinations. The constraint set encodes the hardware decoder directly:

```
constraints:
  - type: oneof
    field: "funct3"
    values: [0, 1]
  - type: cross
    field_a: "funct3"
    field_b: "funct7"
    mapping:
      0: [0]
      1: [0, 1, 2, 3, 4]
```

- `oneof` restricts funct3 to the two accepted bands {0, 1}; funct3=010 (CUS_EXC) and bands 3..7 are structurally impossible because the decoder has no entry for them.
- `cross` restricts funct7 per funct3: funct3=0 requires funct7=0 (NOP), funct3=1 allows funct7 in {0,1,2,3,4} (ADD family).
- There are no `enable_mask` constraints: the decoder accepts every register combination for these encodings, so all combinations are valid and every combination is a distinct instruction word.

Exhaustive Verification Results

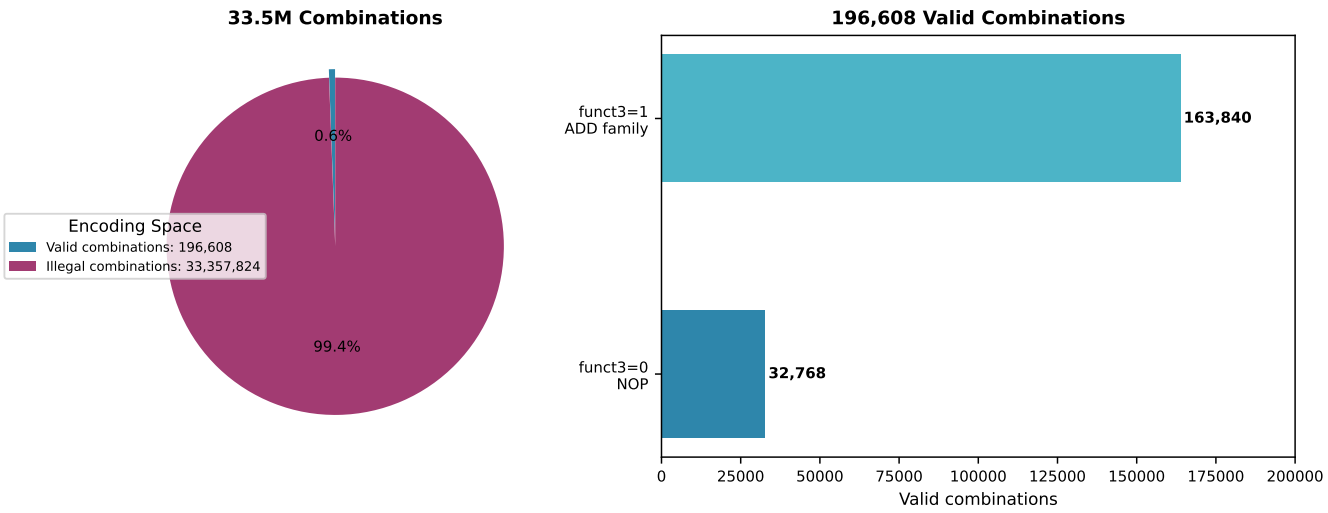

Figure 1: CVA6 XIF encoding space verification: 33.5M combinations evaluated in 18 seconds. funct3=000 accepts only funct7=0 (NOP); funct3=001 accepts funct7 in {0..4} (ADD family); the remaining 99.4% are rejected as illegal.

Per-band breakdown (full-space fixture)

funct3 band	valid combinations	distinct words	instructions
0	32,768	32,768	NOP (funct7=0)
1	163,840	163,840	ADD, DOUBLE_RS1, DOUBLE_RS2, ADD_MULTI, ADD_RS3_R
2..7	0	0	illegal (no decoder entry)
total	196,608	196,608	

The R4 fixture (`xif_ref_r4.xif.yaml`) models the same decoder in R4 format (func2 = bits[26:25], rs3 fixed to 0) with sampled register ranges (rs1, rd in 0..3, rs2 in 0..31):

R4 band	valid combinations	distinct words
funct3=0 (func2=00)	512	512
funct3=1 (func2 in 0..3)	2,048	2,048
total	2,560	2,560

Counting Semantics

This report counts **field-value combinations**. Because the fixtures use no `enable_mask` constraints, every valid combination is also a distinct instruction word: 196,608 combinations and 196,608 distinct words for the full fixture, 2,560 for the R4 fixture. There is no collapsing between the two counts.

Verification Pipeline

The fixture is the single source of truth. Both pipelines read the same YAML and classify every point of the encoding space identically; they differ only in enumeration strategy.

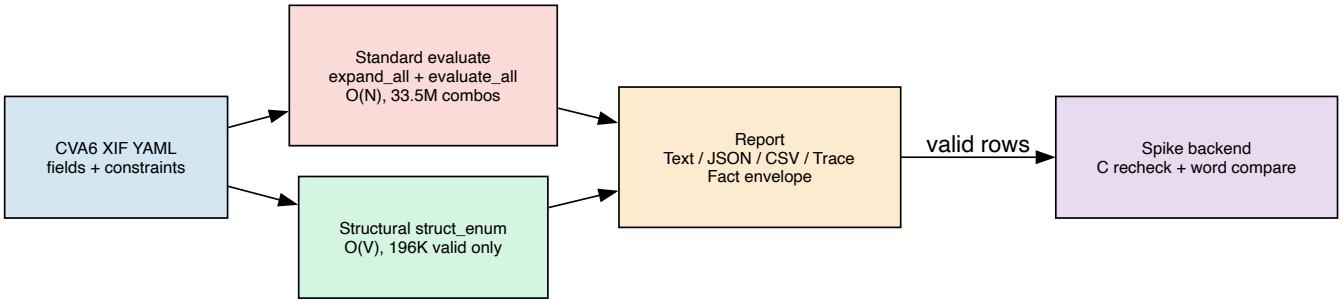

Figure 2: ExaVerif verification pipeline: the YAML fixture drives both the standard evaluate pipeline and the Tagma-based structural pipeline; results are reported and every valid row is re-checked by the Spike backend.

- The standard pipeline materializes the full Cartesian product (33.5M combinations, peak memory ~4-6 GB) and runs every constraint check on every combination.
- The structural pipeline encodes the constraints into the enumeration space itself and visits only the 196,608 valid combinations, lazily, with O(1) memory.
- Both pipelines produce the same valid set; the regression suite asserts this on the CVA6 fixtures (see Correctness Guarantees).
- Every valid row is also re-checked by a C reimplementation under Spike + pk (see How It Works).

Structural Enumeration Algorithm

The structural pipeline (`StructuralEnum` in `src/verify/compose.rs`) converts constraints into an enumeration strategy:

1. `oneof`, `range`, and `bitmask` constraints restrict each field's allowed value list (structural filters).
2. `cross` constraints are kept as parent-to-child mappings and enforced during enumeration instead of by per-combination checks.
3. Fields are topologically ordered so that every parent precedes its cross-constrained children; this guarantees each field can be validated against an already-fixed parent.
4. A lazy odometer runs over the ordered fields. When a field changes, the less significant fields are reset and re-fixed to the smallest valid values using a precomputed first-valid index map, which makes each re-fix O(1).

5. `enable_mask` and `enable_set` constraints are applied to each emitted combination, rewriting masked fields to their forced values.

The iterator emits exactly the combinations that satisfy the full constraint set. This was not always the case: an earlier implementation re-emitted stale child values after a parent advanced. The fix and the regression tests are part of this report's evidence (see Correctness Guarantees).

The advantage scales with space density. Measured per-state costs are about 100-500 ns, and the speedup follows approximately `speedup = k / D`, where D = V/N is the valid density and k is a fixture-dependent constant (about 3-14 here):

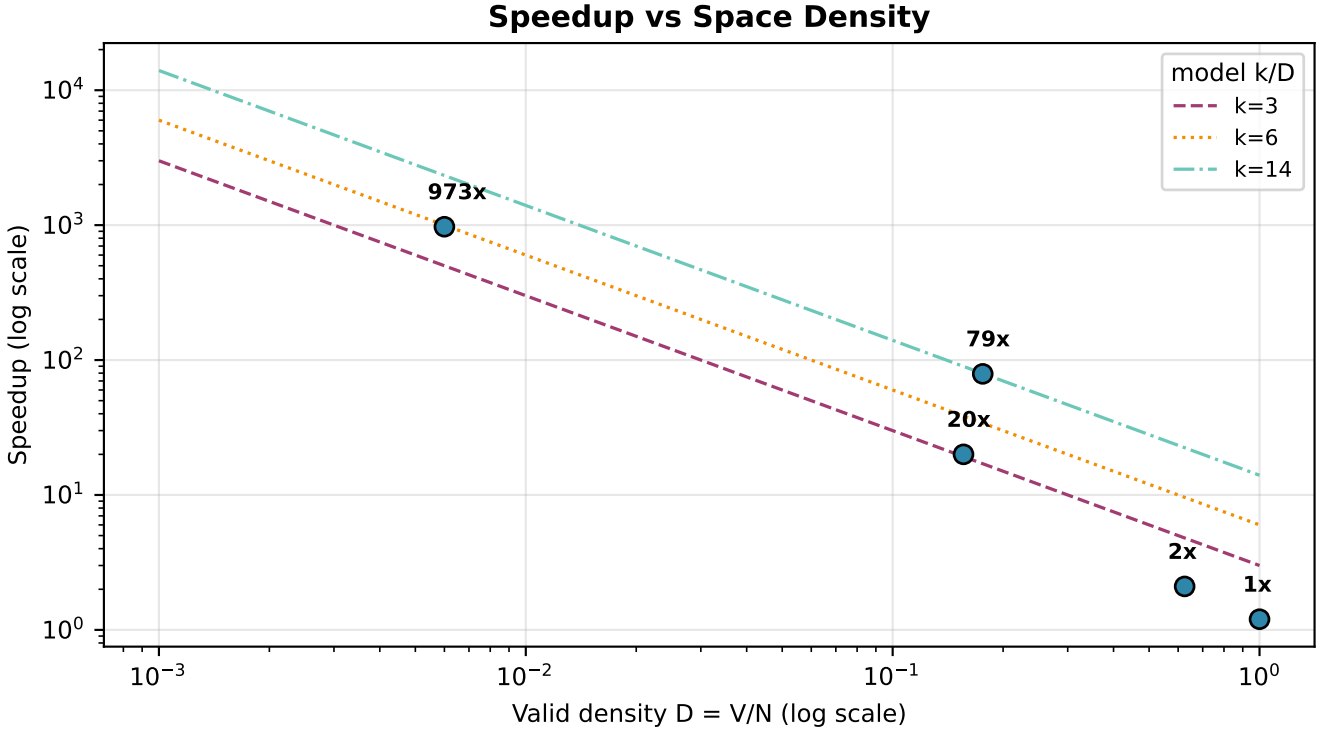

Figure 3: Speedup as a function of valid-space density. The measured points follow the k/D model: sparser spaces give larger speedups. The CVA6 full space at 0.6% density measures ~970x; hundreds of times appear when the density is a few percent or lower.

Memory behavior is the other scaling axis. The standard pipeline allocates O(N) combinations; the structural iterator is lazy and holds O(1) state:

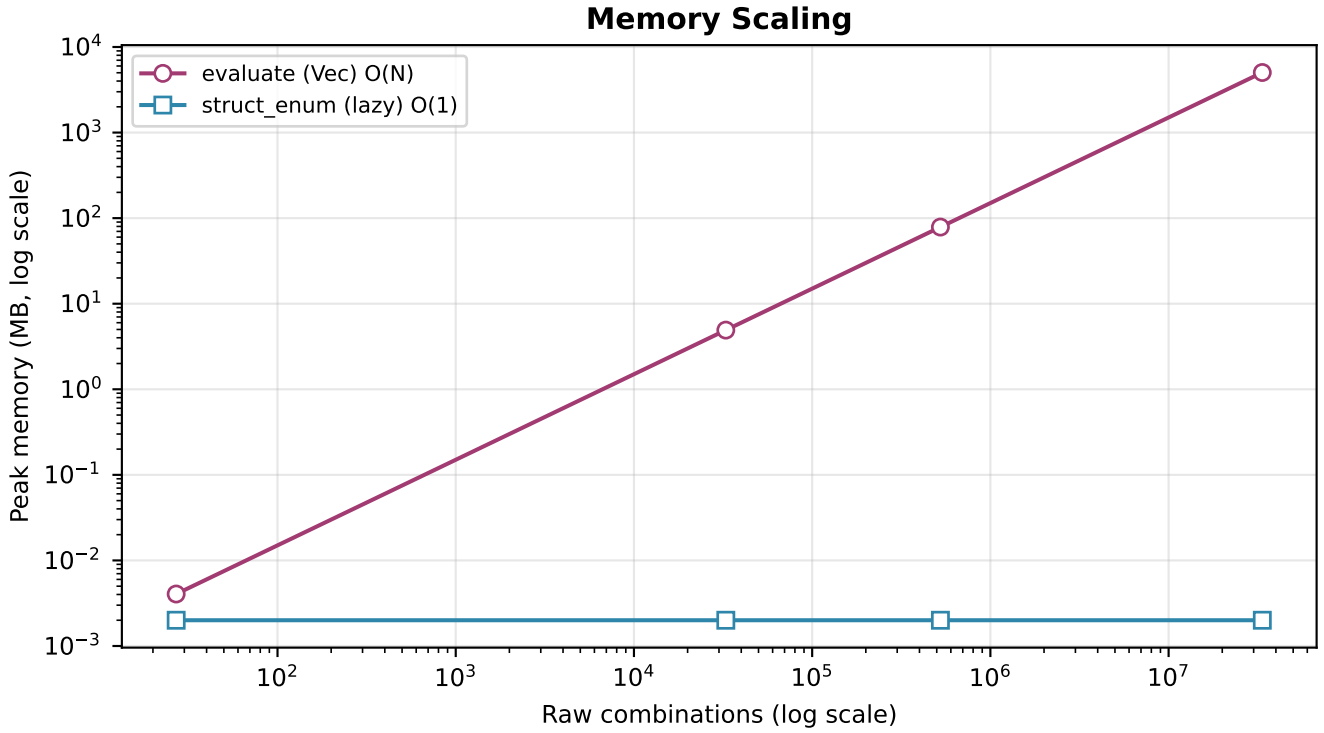

Figure 4: Peak memory vs space size: evaluate allocates the full combination Vec (O(N)); struct_enum streams combinations and stays flat. At 33.5M combinations the standard pipeline peaks at several GB, while the structural path uses a few KB.

At very large spaces the standard pipeline becomes infeasible on memory before the structural path does; the comparison then stops being "slower" and becomes "cannot run".

Correctness Guarantees

The structural pipeline is guarded by regression tests (`tests/structural_enum.rs`) that run in CI:

Test	Guards
`structural_enum_cross_wrap_regression`	the exact parent-advance wrap pattern that produced stale child values
`structural_enum_skips_invalid_initial_state`	the all-zero raw state is never emitted when invalid
`structural_enum_matches_evaluate_cva6_full`	struct_enum emits exactly 196,608 combinations, all satisfying the full constraint set
`structural_enum_matches_evaluate_cva6_r4`	struct_enum emits exactly 2,560 combinations
`validate_into_space_matches_evaluate_cva6_fCombosSpace`	stores exactly the valid paths (196,608 / 2,560)

In addition, the `struct_enum_validity/cva6_full_33M` benchmark asserts on every run that the number of emitted combinations equals the number that pass the full constraint set; a regression fails the benchmark.

Speed and Scale

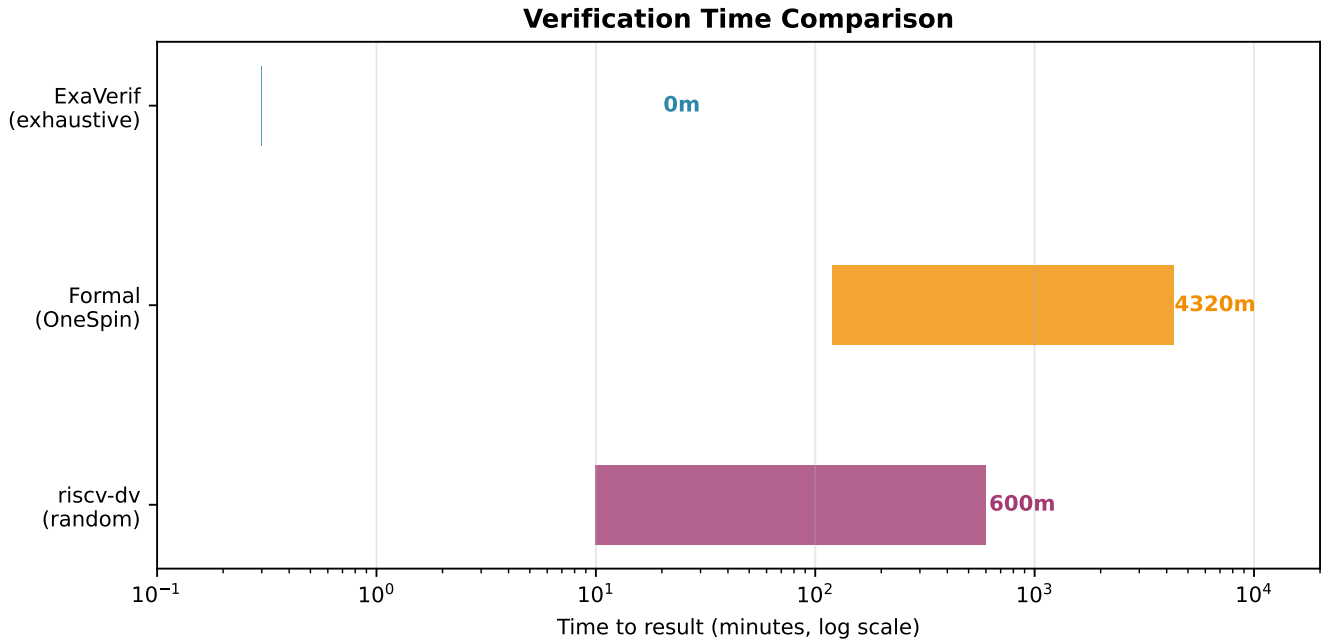

Figure 5: Verification time comparison across three approaches for the CVA6 XIF encoding space. ExaVerif completes in 18 seconds on a single core (release build).

Scope of the comparison: the riscv-dv and OneSpin figures are indicative, not measured here. Formal RTL verification (OneSpin) proves properties about the actual core and is a different class of guarantee. ExaVerif verifies the encoding model derived from the RTL decoder, not the RTL itself; the two approaches are complementary.

Structural Enumeration Performance

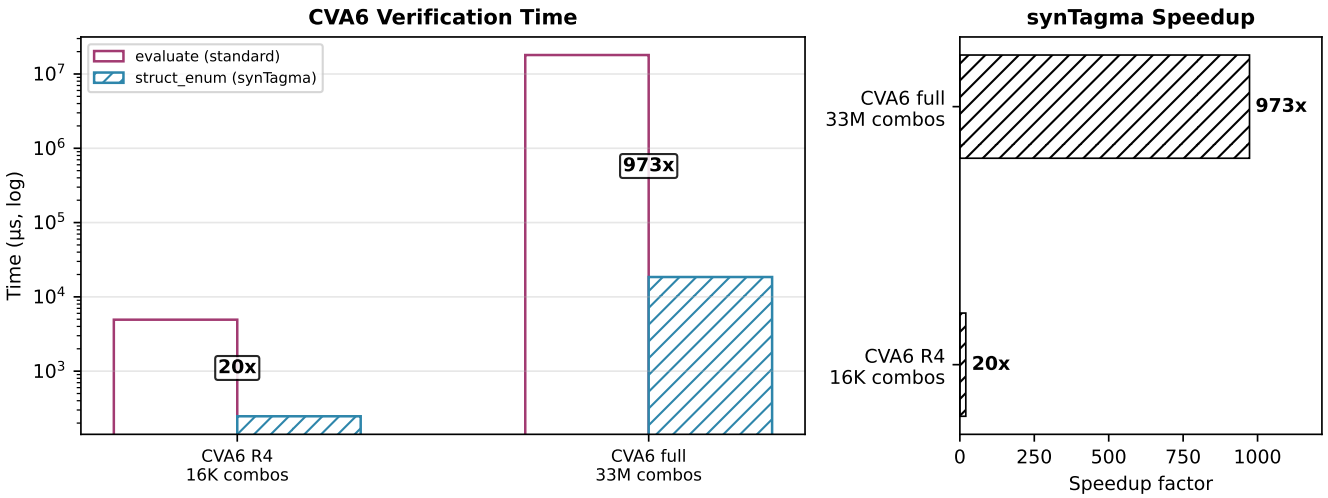

Figure 6: Verification time: standard evaluate (4.93ms for R4, 18.0s for full) vs struct_enum (247us for R4, 18.5ms for full). At CVA6 scale, struct_enum achieves 20x (R4) and 973x (full) speedup on this machine.

Scaling: O(N) vs O(V)

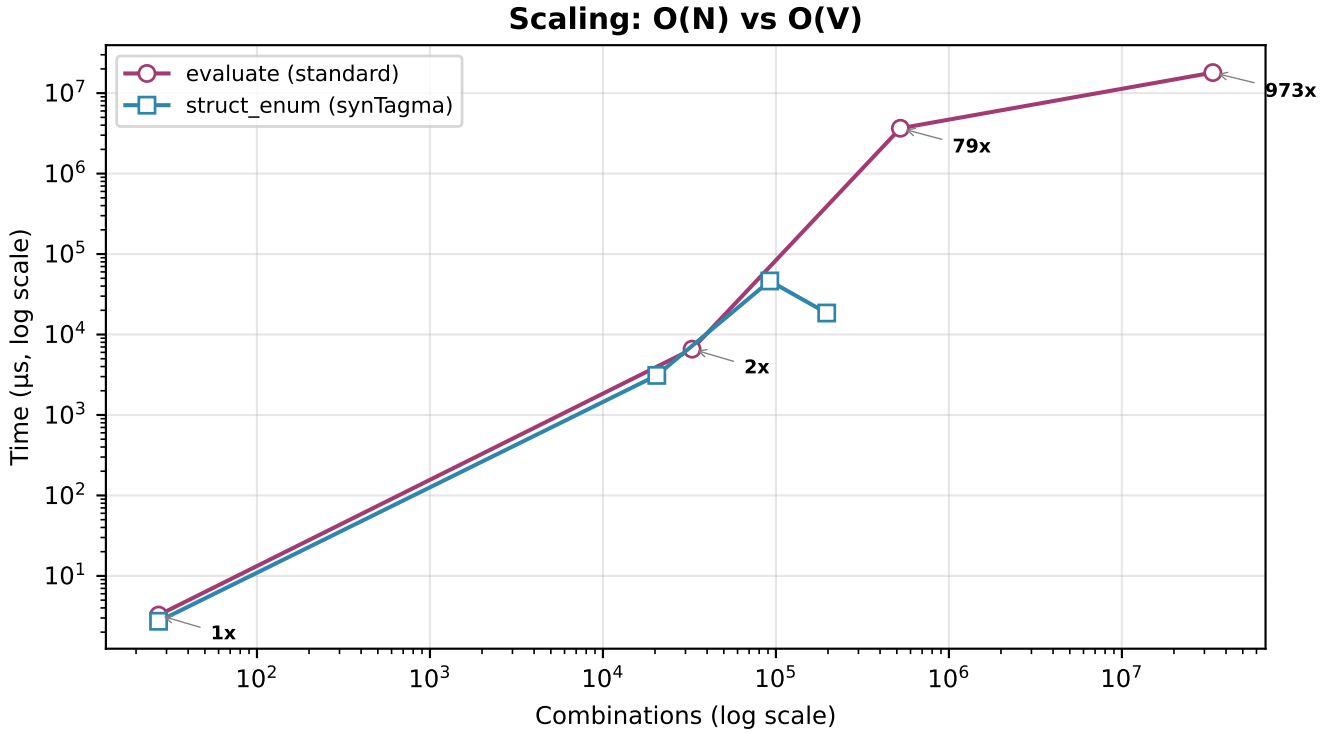

Figure 7: Scaling behavior: evaluate runs in O(N) where N is the full encoding space; struct_enum runs in O(V) where V is the structurally valid subset. The CVA6 full space (0.6% density) shows 973x speedup; denser spaces see smaller gains.

Benchmark Results (release build, this machine)

Fixture	Raw combos	Valid combos	Density	evaluate	struct_enum	Speedup
Small	27	27	100%	3.26 µs	2.73 µs	1.2x
Medium	32,768	20,480	62.5%	6.58 ms	3.11 ms	2.1x
Ibex R-type	524,288	92,160	17.6%	3.66 s	46.3 ms	79x
CVA6 R4	16,384	2,560	15.6%	4.93 ms	247 µs	20x
CVA6 full	33,554,432	196,608	0.6%	18.0 s	18.5 ms	**973x**

Figure 8: Complete benchmark data including CVA6 fixtures. struct_enum achieves 1.2x-973x speedup over the standard pipeline across all real-world RISC-V verification targets. Absolute times are machine-specific.

The valid-combination counts in the table are combination counts, and with no enable_mask constraints they equal the distinct instruction-word counts.

Reproducibility

All numbers in this report are reproducible from the committed benchmark suite and fixtures:

```
# full-space CVA6 group: evaluate, struct_enum, validate, validity guard
cargo bench -- cva6_full

# structural enumeration only
cargo bench -- "struct_enum/cva6"

# validity guard (must stay green)
cargo bench -- struct_enum_validity
```

Fixtures: `tests/fixtures/cva6/xif_ref.xif.yaml` (full space), `tests/fixtures/cva6/xif_ref_r4.xif.yaml` (R4). Regression tests: `tests/structural_enum.rs`. The fixtures are derived from the CVA6 hardware decoder mask table (`cvxif_instr_pkg.sv`, `instr_decoder.sv`) at commit `6544a714c`. Measurements were taken in release mode on a single Apple silicon core; absolute times are machine-specific, while the O(N) vs O(V) relationship is not.

How It Works

A single YAML file describes the encoding space and its constraints. The entire space is enumerated exhaustively and evaluated deterministically. Every point is classified as valid or invalid based on the specification.

```
ev verify --target cva6_xif_ref.xif.yaml
```

Every valid combination is also cross-checked by a C reimplementation. The `simulate` command packs all 196,608 valid combinations into a single ELF binary and runs it under Spike + pk. The C program re-evaluates the constraint model and re-assembles each instruction word, comparing it against the Rust-computed reference. All 196,608 rows agree. This validates the C/Rust code generation and the assembly layout. It is not ISA-level simulation: custom-3 opcodes are illegal in the base RISC-V ISA, which is exactly why CVA6 offloads them, so Spike never executes them.

```
EV_SIM_BACKEND=spike ev simulate --target cva6_xif_ref.xif.yaml
```

What the Spike backend actually checks

The generated C program (`src/synth/backends/spike.rs`) contains:

- the full encoding table as data, one row per valid combination;
- `check_encoding()`, a C reimplementation of the same constraint model (`oneof`, `cross`, and the other constraint types);
- `assemble_instr()`, a C reimplementation of the instruction-word assembly;
- `INSTR_WORDS[]`, the Rust-computed reference words for the same layout;
- a per-row comparison of the C-assembled word against the Rust reference.

Each row prints `ENC:<idx>:<ok>:<word_ok>:<word>`. A row fails when the C constraint check disagrees with the Rust evaluation (reason: constraint re-check failed) or when the C assembler disagrees with the Rust reference (reason: instruction word mismatch). This is a cross-language codegen check, not an ISA simulation; see the abstract for the boundary.

Structural vs Standard: the mechanism

Aspect	Standard (`evaluate`)	Structural (`struct_enum`)
Combinations generated	**All** 33,554,432	**Valid only** 196,608
Constraint evaluation	10 checks × 33.5M	**0** (structurally encoded)
Invalid detection	`check.allows()` → false	Vacant slot → 1.65 ns
Memory	4 GB peak (Vec)	0 (lazy iterator)
Time (CVA6 R4)	4.93 ms	247 µs
Time (CVA6 full)	18.0 s	18.5 ms

The structural iterator is verified against the standard pipeline: the regression suite asserts that every emitted combination satisfies the full constraint set, and that `validate_into_space` stores exactly the valid paths (see Correctness Guarantees).

Why the gain is algorithmic, not linguistic

The 973x is not a language effect. Both pipelines are Rust, compiled in the same release profile, and measured by the same criterion harness. The ratio is therefore the enumeration-strategy difference alone: O(N) traversal with per-combination checks versus O(V) structural access with the checks encoded into the enumeration. Porting both pipelines to another language preserves the ratio (both sides carry the same per-element overhead). The comparison targets in the field are also different categories: riscv-dv runs in a UVM/simulator stack that does not enumerate, and formal tools use SAT/BMC engines that are not exhaustive searches. ExaVerif is an exhaustive enumeration of the encoding space; that is a different category from both.

Comparison with synTagma Benchmarks

The struct_enum speedup mirrors synTagma's core thesis: structural addressing eliminates hash-based lookup overhead. The primitive timings (1.65 ns, 5.2 ns) are synTagma measurements, reproduced here for context.

synTagma primitive	ev application	Measured	Advantage
Nonexistent prefix lookup	Invalid encoding detection	**1.65 ns**	14x vs HashMap
CoordPath indexing	Combination addressing	**5.2 ns**	O(depth) = O(5)
CoordSpace sparse get	Valid encoding iteration	**247 µs** (16K space)	20x vs evaluate
Axis projection	funct7/funct3 filtering	**46.3 ms** (524K space)	79x vs evaluate

Comparison with Random Verification (riscv-dv)

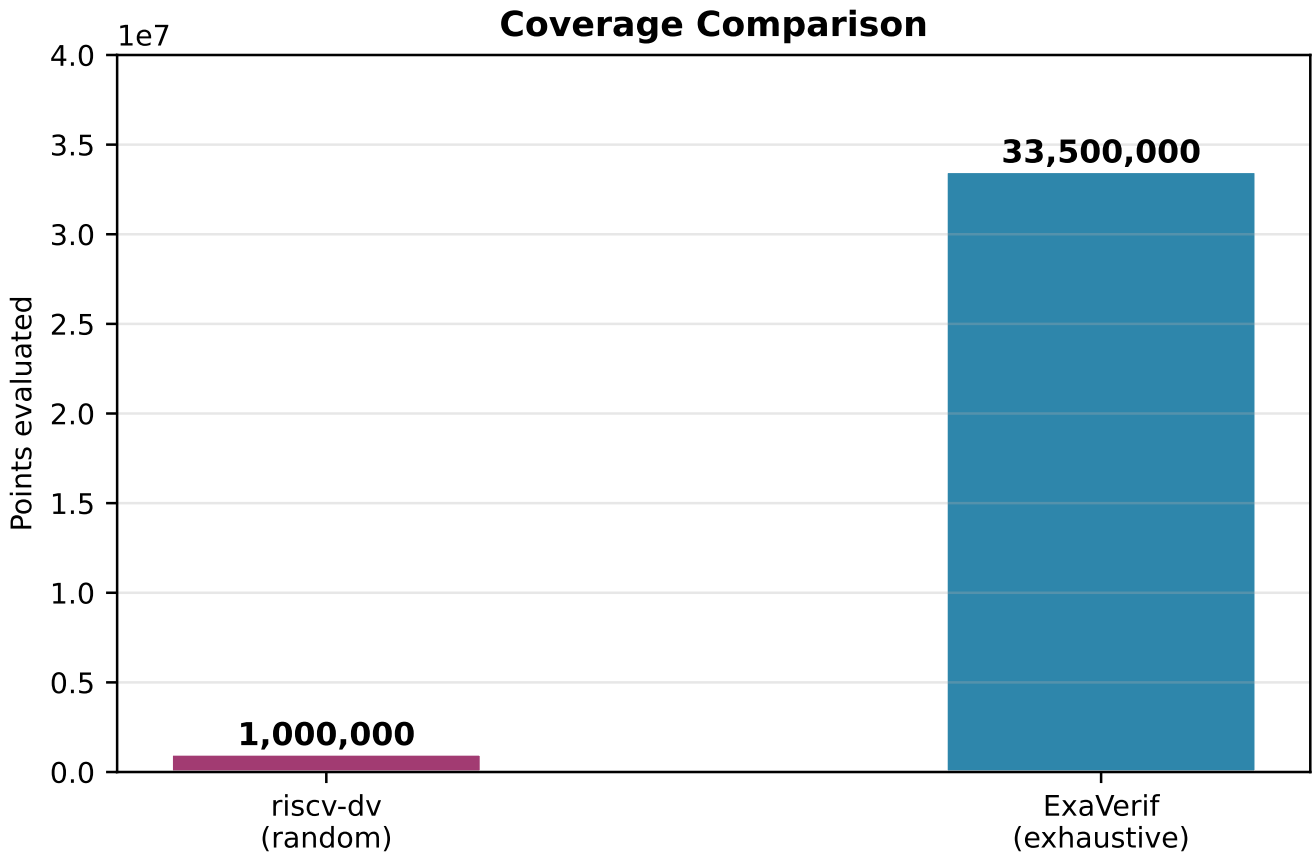

Figure 9: Unique encoding coverage: exhaustive enumeration evaluates the full 33.5M-point space, where riscv-dv's random sampling budget is a configuration choice, not a ceiling.

The 33.5x ratio compares a 1M-sample budget against the full space. Random generators can raise coverage by increasing the budget or by directing samples with coverage feedback; the point of exhaustive enumeration is determinism and completeness, not a fixed budget.

Relation to the Standard CVA6 Verification Environment

The CVA6 project verifies the core with a UVM testbench (`vcs-uvm`), the riscv-dv random instruction generator, and tandem simulation that compares the RTL model trace against the Spike trace (`DV_SIMULATORS= veri-testharness,spike`, summary in `iss_regr.log`). Regression suites include riscv-arch-test, riscv-compliance, riscv-tests, and riscv-dv; runs are driven by `verif/sim/cva6.py`. ExaVerif is complementary to this flow:

- The standard flow checks execution behavior against Spike and accumulates coverage statistically. ExaVerif classifies the entire encoding space deterministically, which the random flow can never prove complete.
- The custom-3 encodings this report verifies are the ones the CVA6 hardware decoder accepts; the DV class also generates encodings the decoder rejects (see DV class divergence), which is a finding for the CVA6 verification team.
- ExaVerif's Spike backend is a C/Rust codegen cross-check, distinct from CVA6's tandem Spike simulation. Running the accepted encodings through the standard `veri-testharness,spike` flow in the CVA6 repository is the natural next step to close the loop.

R4 vs Full Fixture

Two fixtures model the same decoder at different granularities:

Aspect	Full (`xif_ref.xif.yaml`)	R4 (`xif_ref_r4.xif.yaml`)
Encoding	flat R-type (funct7 = bits[31:25])	R4 format (func2 = bits[26:25], rs3 = bits[31:27])
Registers	rs1, rs2, rd all in 0..31	rs1, rd in 0..3; rs2 in 0..31
Raw space	33,554,432	16,384
Valid combinations	196,608	2,560
funct3=0	funct7=0 (NOP)	func2=00 (NOP)
funct3=1	funct7 in 0..4 (ADD family)	func2 in 0..3 (ADD family)

ADD_RS3_R (funct7=4) sets rs3[0]; the R4 fixture fixes rs3 to 0, so this encoding is not representable in R4 and is verified only in the full fixture.

Limitations

What ExaVerif does not yet model

- **ADD_RS3_R in the R4 fixture**: the hardware distinguishes ADD_RS3_R via rs3[0]=1 (bit[27]); the R4 fixture fixes rs3 to 0, so ADD_RS3_R is verified only in the full fixture.
- **DV class encodings**: CUS_U_ADD, CUS_S_ADD, CUS_ADD_MULTI (funct3=000, funct7 in {2,6,8}) and CUS_EXC (funct3=010) are generated by the DV class but rejected by the reference decoder. This report verifies the decoder acceptance; the DV encodings are reported as a divergence.
- **Weighted register distribution**: the DV class biases certain register values. ev treats all register values uniformly.
- **Pipeline hazards**: write-after-read hazards (rs1==rd) are not modelled at the encoding level.

References

- CVA6 hardware decoder (authoritative reference, commit `6544a714c`): `cva6/core/cvxif_example/include/cvxif_`
  `cva6/core/cvxif_example/instr_decoder.sv`, `cva6/core/cvxif_example/copro_alu.sv`
- DV class: `cva6/verif/env/corev-dv/custom/cvxif_custom_instr.sv`
- CVA6 project overview and verification environment: https://deepwiki.com/openhwgroup/cva6 https://deepwiki.com/openhwgroup/cva6/Verification_Environment
- CVA6 simulation tutorial (bsc-loca fork): https://raw.githubusercontent.com/bsc-loca/cva6/refs/heads/master/tutorials/running_sim.md
- Fixtures: `tests/fixtures/cva6/xif_ref.xif.yaml`, `tests/fixtures/cva6/xif_ref_r4.xif.yaml`
- Benchmark suite: `benches/bench.rs`
- Regression tests: `tests/structural_enum.rs`
- Structural enumeration: `src/verify/compose.rs`
- Spike backend: `src/synth/backends/spike.rs`