Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

TensorEQC: Structure-Aware Equivalence Checking with GF(2) Algebra, Exact Tensor Networks, and Reset Analysis : a Multi-Algorithmic Study of Datapath Verification and Reset-State Dependence

Christophe Bianchi

christophe.bianchi@octinnov.com

École Centrale de Lyon https://orcid.org/0000-0003-2714-4031

Research Article

Keywords: equivalence checking, Boolean algebra, tensor networks, tensor trains, arithmetic circuits, reset, X-propagation

Posted Date: November 11th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11687153/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

TensorEQC: Structure-Aware Equivalence Checking with GF(2) Algebra, Exact Tensor Networks, and Reset Analysis

A Multi-Algorithmic Study of Datapath Verification and Reset-State Dependence

Christophe Bianchi, PhD | École Centrale de Lyon, France

21 October 2026

Abstract

Equivalence checking is sensitive to the representation of a circuit. XOR-rich networks, carry chains, finite-field multipliers, and integer multipliers expose different kinds of exact structure. We describe TensorEQC, an executable research workflow combining structural hashing, counterexample simulation, algebraic normal forms over GF(2), relational tensor-network elimination, ordered binary decision diagrams, and small-support exhaustive evaluation. A companion module checks reset dependence by explicit-state self-composition; exact tensor-train tools measure algebraic rank and test zero functions. The reported synthetic campaign includes 2,048-bit rewritten ripple adders and 128-bit polynomial-basis multipliers, as well as cases where the portfolio adds overhead or reaches its limits. A supplementary experiment performed for this manuscript reconstructs five complete 16-input functions from exact GF(2) tensor trains, validates all unfolding ranks, and checks a parity-cancellation negative control. The results support structure-dependent engine selection, not universal tensor superiority. Timing comparisons are against internal baseline engines, not commercial or SAT-based systems. The submission distinguishes newly reproduced evidence from previously reported campaign results whose original execution artifacts must accompany an archival release.

Keywords: equivalence checking; Boolean algebra; tensor networks; tensor trains; arithmetic circuits; reset; X-propagation.

1. Introduction

A useful equivalence checker must make progress on heterogeneous logic without weakening the statement being proved. For some regions, correspondence of gate structure is enough. For others, polynomial structure is more useful than topology. A carry chain can have a small elimination interface even when the input space is enormous. A network with many cross-bit dependencies may instead favour decision diagrams or a different exact solver.

TensorEQC treats representation choice as part of the verification workflow. Its contribution is an executable integration and accompanying case studies, rather than a new Boolean algebra, a new tensor decomposition, or the first use of multiple formal methods. The principal design rule is simple: an engine may discharge an obligation only according to its own exact semantics; otherwise the obligation remains available to another engine.

The work has three components. First, a combinational portfolio retains per-output coverage and replays negative witnesses. Second, exact tensor-train (TT) analytics measure how field and variable order affect rank. Third, a synchronous reset module distinguishes observable initial-state dependence from uncertainty that disappears before a declared observation boundary.

The scope is deliberately precise. The demonstrated input interface is a Boolean DAG, restricted BENCH, or scalar flat structural Verilog. A general RTL and cell-library frontend is under development and is outside the present experiments. The study does not include customer netlists, commercial checker comparisons, asynchronous timing semantics, or an independently verified proof-production chain. These are restrictions on the interpretation of the results, not implementation capabilities implied by a

product name.

2. Problem and result contract

Let $F,G:\{0,1\}^n \to \{0,1\}^m$ denote the reference and candidate circuits with matched named inputs and outputs. For output i, define

$$D_i(x)=F_i(x)\oplus G_i(x),\ M(x)=\vee_{i=1}^{m} D_i(x).$$

Equivalence means MEquiv 0. The experiments use unconstrained binary inputs. A legal-input predicate would create the distinct obligation A∧ MEquiv0; its satisfiability and justification would also require checking.

The implementation validates topological order, supported primitives, and output correspondence. Supported Boolean operators are AND, OR, XOR, XNOR, NAND, NOR, NOT, BUF, and MUX. The scalar structural-Verilog reader is not a SystemVerilog elaborator. It rejects behavioural constructs, vectors, and unsupported cell instances instead of guessing their semantics.

An overall Eq\ result requires all named outputs to be discharged. One replayed mismatch suffices for Neq. A mixture of discharged and unfinished outputs is Unresolved, not equivalence. Execution errors and resource limits are recorded separately in engine logs. This paper uses ``proof'' for exact satisfaction of the encoded Boolean or finite-state obligation; it does not imply that the program itself has been formally verified.

3. An exact multi-engine workflow

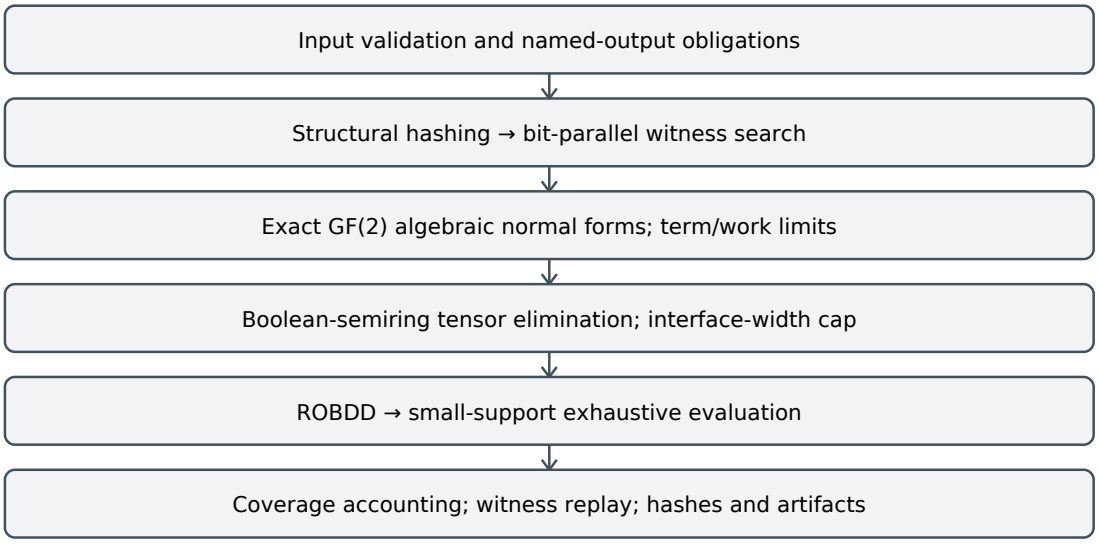


Figure 1. Implemented combinational portfolio. Unfinished obligations proceed to the next stage. TT rank analytics and sequential reset analysis are companion modules, not hidden stages of this proof path.

3.1. Preprocessing and negative evidence

Structural hashing establishes identical expression DAGs under the supported operators, including commutative-input normalization. Packed simulation then evaluates random inputs together with directed all-zero and all-one patterns. Simulation can establish a negative result with a replayed witness; an absence of mismatches is never used as an equivalence proof.

3.2. Algebraic normal forms

A Boolean function has a unique multilinear polynomial representation over GF(2), after reduction by $x_i^2=x_i$. A polynomial is stored as a set of monomial masks. XOR is symmetric difference. Multiplication unions variable masks and cancels repeated monomials modulo two. Operator encodings include

$$\neg\, a=1+a,\ a\vee b=a+b+ab,\ \mathrm{mux}(s,t,e)=e+s(t+e).$$

The representation is exact but may grow exponentially. The release uses term-count and multiplication-work budgets.

Lemma 1 (Witness from a nonzero ANF). Let P be a nonzero multilinear polynomial over GF(2). Choose a monomial of minimum cardinality in its support, with variable set S. Set exactly the variables in S to one. Then P evaluates to one.

Proof. Only monomials whose variable sets are subsets of S can contribute. By minimum cardinality there is no strict-subset monomial. The chosen monomial is the only contributing monomial, so the value is one. The empty monomial is covered by the same argument. □

This yields a simple exact negative witness when two output ANFs differ. The implementation still replays it on the original gate models. Linear maps and carry-free polynomial-basis multiplication are favourable ANF workloads. Carry propagation can create high-degree terms, so successful finite-field results do not predict integer-multiplier performance.

3.3. Relational tensor elimination

A gate is represented by a Boolean indicator tensor of its legal input/output assignments. Shared wires are shared variables. Joining factors is conjunction; eliminating a variable is disjunction over its binary values. Thus the contraction operates over the Boolean semiring and answers an existence question, as in established tensor formulations of Boolean problems [3].

The implementation plans a minimum-degree elimination order and refuses a plan once its intermediate scope exceeds the configured width. It retains elimination buckets for witness back-substitution. An aborted planning width is a threshold crossing, not a completed estimate of the best possible width.

Proposition 1 (Semantics of elimination). Eliminating an internal variable by disjunction preserves the existence of a satisfying completion of the remaining factors. Eliminating all variables from a mismatch relation returns false exactly when that relation is unsatisfiable.

Proof. For a chosen variable v, collect all factors that contain it. Replacing their conjunction by ∃ v of that conjunction is an equivalent projection for the remaining variables; factors not containing v can be retained unchanged. Iteration proves the claim. □

The important qualification is the semiring. Summing mismatches in GF(2) computes their parity: two mismatches can produce zero. This is not a sound equivalence criterion. TensorEQC's existence engine uses Boolean elimination, while its GF(2) TT module uses a distinct zero-function test.

3.4. ROBDD and exhaustive fallback

The ROBDD engine follows canonical ordered-function reasoning [1], using the circuit input order and a node cap. Small-support enumeration is an exact final fallback. Neither a favourable TT representation nor a good elimination order is assumed for every region. In the present implementation, routing is an attempt-based sequence with budgets, not a learned predictor. Failed attempts are charged to total runtime.

4. Tensor trains and field-sensitive structure

An order-d Boolean tensor admits a TT representation

$$f(x_1,\ldots,x_d)=G_1(x_1)\cdots G_d(x_d),\ r_0=r_d=1,$$

with storage $\Sigma_k\ 2r_{k-1}r_k$ for binary physical dimensions [2]. Over a field, exact ranks are the ranks of the corresponding unfoldings. Rank over GF(2) can be much smaller than rank over the reals. Real-valued SVD and finite-field elimination therefore address different structure.

The implemented TT analysis builds exact cores from a complete truth table using successive GF(2) row-space factorizations. It reconstructs the original table and independently compares cut ranks with direct elimination. This is an exponential-input diagnostic for small functions, not a scalable netlist-to-TT compiler. These TT tools do not supply the large-netlist proofs reported later; those use relational elimination or ANF.

Proposition 2 (Exact TT zero test). For an exact TT over a field, propagate a basis for the span of all prefix row products through both slices of each core. The represented function is identically zero iff the final reachable scalar span is zero.

Proof. At the first boundary the span is generated by the scalar one. Matrix multiplication is linear in the prefix row. Consequently, multiplying a basis by each slice and retaining a basis of their combined span preserves the span of all extended prefix products. At the final scalar boundary this span is zero exactly when every full index assignment gives zero. □

This proposition concerns testing an available exact representation. It does not establish a cheap construction algorithm or a general low-rank bound.

5. Dedicated datapath treatment

The relevant distinction is the mathematical domain, not simply whether a region contains arithmetic-looking gates. Integer addition modulo 2^n is not XOR. Polynomial multiplication modulo an irreducible binary polynomial is not integer multiplication. Saturation, truncation, signed interpretation, and pipeline latency introduce separate obligations.

Word-level and finite-field verification have substantial prior art [4, 5]. TensorEQC does not claim that incumbent methods ignore datapaths. Its engineering objective is to preserve useful structure in a heterogeneous workflow and expose which representation discharges which region.

A ripple adder illustrates a narrow interface:

$$a_i+b_i+c_i=s_i+2c_{i+1}.$$

Only a binary carry connects adjacent bit groups. This motivates chain-like elimination. Prefix adders and multipliers create wider dependencies, so the same heuristic need not work. For GF multiplication, output ANFs remain degree two in operand bits, whereas integer carries can raise degree and polynomial size.

The implemented flow does not depend on unverified operator recognition: all engines act on the gate semantics. This avoids a mistaken word-level abstraction but also gives up the potential benefits of specialized arithmetic identities. Additional frontend and arithmetic integration are under development; the measurements below do not include those modules.

6. Sequential reset and X reasoning

The synchronous model specifies initial state, next-state functions, a reset prefix, and an observation boundary. Output checks use the post-transition state and the current cycle's inputs. The reset input is one for a fixed number of transitions and then permanently zero.

For reset dependence, self-composition gives two copies identical input sequences but independently chosen permitted initial states. A difference at a protected output after the observation boundary produces a trace. Each transition and observed difference is replayed. Exhausting the complete finite product establishes independence within that declared model; a depth-limited run gives only a bounded pass. An incompatible initial relation is vacuous, not equivalent.

The combinational X diagnostic preserves identity across repeated unknown inputs by enumerating their binary completions. Thus $q \oplus q$ is known zero even when a local three-valued gate abstraction reports X. A separate one-bit RTL-if micro-model illustrates optimism. These diagnostics relate to established formal and simulation X methods [6]. They are not a generic RTL-to-gate four-state simulator, nor do they prove absence of analog metastability or asynchronous reset timing failures. A general frontend is under development.

7. Experimental protocol and provenance

Two evidence sets are separated throughout this paper.

Reported portfolio campaign.

The earlier version-1.0.0 campaign recorded 89 isolated case/engine runs, nine rank-profile functions, twelve reset configurations, and six additional runs after correcting a multiplier-fault construction. The supplied project record contains timing and result summaries, including favourable and unfavourable cases. The original full-flow source archive and raw logs were not available when preparing this manuscript. Accordingly these are reported historical results, not a claim that the complete campaign was independently rerun for this submission.

The reported environment was a two-core Linux container, Python 3.12.8, NumPy 1.26.4, with a 60-second case timeout, one-million-node BDD cap, tensor scope cap 22, 200,000 ANF monomials, and three million ANF multiplication operations. Single-run timings have no confidence intervals. Internal BDD and tensor-only engines were the baselines; no commercial checker, external SAT engine, or industrial netlist was tested.

New reproducible supplement.

For this manuscript we implemented and ran a self-contained exact-GF(2) TT validation script on five full 16-input truth tables. Each factorization was timed three times; input generation, reconstruction, and direct-rank checks are outside the reported construction time. Every reconstructed table was compared entry-for-entry, and all 15 cut ranks were checked by a separate rank routine. The supplement and all three timing samples accompany this manuscript.

The new experiment used Python 3.12.8 and NumPy 1.26.4 on Linux. Timings are local process measurements, not hardware-independent performance guarantees. The five functions are parity, bit 7 of eight-bit addition under two variable orders, bit 4 of byte multiplication modulo $x^8+x^4+x^3+x+1$, and a seeded random function. GF multiplication truth values are generated directly by shift-and-reduce arithmetic, independently of the earlier gate generator.

8. Results

8.1. Reported equivalence results

Case	Width	Gates (ref./cand.)	Full flow	BDD only
Ripple/NAND rewrite	256	1,280/3,840	Eq\ 0.52 s	Eq\ 3.89 s
Ripple/NAND rewrite	1,024	5,120/15,360	Eq\ 1.38 s	Unresolved
Ripple/NAND rewrite	2,048	10,240/30,720	Eq\ 2.87 s	Unresolved
Ripple/prefix	128	640/2,690	Eq\ 0.84 s	Eq\ 0.61 s
Integer multiplier	10	1,100/1,100	Eq\ 5.98 s	Eq\ 5.55 s
GF multiplier	128	33,021/33,021	Eq\ 0.80 s	Unresolved
Affine network	256	4,146/4,146	Eq\ 0.07 s	Eq\ 2.00 s
Affine network	1,024	65,700/65,700	Eq\ 5.93 s	Unresolved

Table 1. Selected historical portfolio results, not rerun for this manuscript. Width is operand width except for affine networks, where it is input count. Unresolved\ denotes exhaustion of a configured budget, not a timed-out equivalence proof.

The tensor-only path reportedly proved the 2,048-bit rewritten ripple pair in 2.60 s with a planned interface width of 18. The full flow included unsuccessful ANF work. At 256 bits the tensor-only result was 0.27 s, whereas BDD took 3.89 s. These observations show a capacity difference under the chosen implementation and limits, not superiority over optimized commercial arithmetic solvers.

For the prefix adder, tensor planning crossed the width cap and BDD solved the remaining problem. For integer multiplication, ANF ceased to fit at eight bits and the tensor path also exceeded its scope cap; the ten-bit result came from BDDs. These cases are essential to interpreting the portfolio: the full flow may be slower than choosing the successful engine directly.

The reported GF multiplier pair uses schoolbook and Horner-like constructions of the same polynomial-basis map. At 128 bits the ANF-only run took 0.61 s; the full flow took 0.80 s. Reported BDD runs stopped at the node cap from 16 bits. Finite-field results must not be extrapolated to integer multiplication.

The 1,024-input affine example reportedly used about 1.2 GB of resident memory. That cost is a material limitation, not evidence of negligible-memory linear reasoning. Runtime is not the only capacity metric.

8.2. New exact TT validation

Function/order	Max. rank	TT entries	Dense/TT	Median build (ms)
Parity	2	120	546.13	0.544
Addition, interleaved	3	168	390.10	0.581
Addition, split	129	77,492	0.846	8.976
GF byte product, split	8	928	70.62	1.009
Random control	256	174,760	0.375	26.558

Table 2. New measurements: five 16-input functions, each with 65,536 truth-table entries. Counts are logical entries, not a process-memory compression ratio. Times exclude truth-table generation and reconstruction. Three construction runs per function; all complete reconstructions and cut-rank checks passed.

Table 2 confirms that exact rank depends strongly on ordering. Interleaving adder operands yields rank at most three; splitting operands produces rank 129 and more core entries than a dense table. The random control is also larger in TT form. No claim of universal compression is supported.

The previously reported rank profile additionally compared real and GF(2) ranks. For the split-order GF byte-product output, it reported real rank 255 and GF(2) rank eight. The new supplement reproduces the finite-field rank and core count but does not independently remeasure real rank. The field-comparison value remains historical evidence.

The new negative control has exactly two nonzero assignments. Its scalar GF(2) sum is zero, while the TT zero-function test correctly returns false. An all-zero control returns true. These controls directly test the distinction between parity and existence.

8.3. Reported correctness and negative controls

The historical unit suite reported 20 passing tests. One test compared 150 small circuit pairs across exhaustive, ANF, tensor, BDD, and full-flow configurations against exhaustive truth-table comparison. The engines shared primitive semantics, so this is differential validation within the implementation, not independent proof of correctness.

Reported negative controls included carry faults, affine gate faults, and a rare finite-field output fault triggered by an AND of 16 or 24 primary inputs. All reported negative witnesses were replayed. For integer multipliers the first mutation was functionally redundant and correctly returned equality; a replacement partial-product AND-to-OR fault produced replayable mismatches at six, eight, and ten bits, with exhaustive agreement at six and eight bits. A syntactic mutation alone is not a ground-truth inequivalence label.

8.4. Other recorded project artifacts

The earlier 0.3.0 reference reported 22 tests, including a differential check on 100 four-input circuits, and a 128-bit ripple-adder BDD resource limit under a different cap. These results use different software and budgets and are not pooled with version 1.0.0 timings.

The generated ARITH128 stress family contained 180,225 reference gates and 720,900 rewritten gates; the RECONV512_R24 family contained 61,440 and 245,760 gates. Each also included a rare-trigger faulty candidate. Their recorded validation was 256 random vectors plus all-zero/all-one and directed fault detection; the arithmetic reference additionally matched integer multiplication on those vectors. These are simulation checks, not full formal results. Raw gate count and local rewrite similarity do not

establish industrial proof difficulty. The present manuscript does not upgrade those artifacts to a full-chip proof-capacity claim.

8.5. Reported reset results

For pipelines of depth two, four, six, and eight, resetting every stage removed initial-state dependence under the chosen observation contract. Resetting only the first stage left dependence visible immediately after reset, but it disappeared after sufficient flushing cycles. At depth eight, the reported complete product exploration visited 65,793 states for the fully reset case and 98,432 for the flush-complete case, taking 2.5 s and 3.2 s respectively. Early dependence was found in 22 ms with a replayed trace.

These are small-state diagnostic results. Their exponential growth is visible in the state counts; they do not demonstrate chip-wide reset analysis.

9. Interpretation and limitations

The experiments support a representation-sensitive strategy. GF algebra fits low-degree finite-field and affine functions. Relational elimination fits certain carry-chain miters. ROBDDs remain useful where the other representations grow too large. Exact TT analytics explain some of these differences but are not the large-circuit proof engine.

Three boundaries prevent stronger claims. First, baseline comparisons are internal, and single-run timings do not estimate variability. Second, historical execution summaries are not a replacement for archived source, raw logs, and hashes. Their recovery is an artifact requirement before treating the full campaign as independently reproducible. Third, language translation, technology-cell interpretation, proof-checker trust, and state correspondence are part of the overall verification problem. A correct backend cannot certify an unsupported frontend.

A general frontend, SAT integration, and independent certificate handling are under development. They were not exercised in this study. Replacing that statement with a performance claim would change the meaning of the evidence. The present implementation covers binary combinational primitives and small explicitly specified synchronous machines; it does not include general SystemVerilog scheduling, power-aware semantics, arbitrary retiming/state recoding, asynchronous reset timing, or analog metastability.

10. Conclusion

TensorEQC combines exact engines behind an explicit output-coverage and witness-replay contract. The reported portfolio campaign identifies both favourable structures and important counterexamples to a universal tensor advantage. The new reproducible supplement confirms exact GF(2) TT construction, field-specific rank measurements, ordering sensitivity, and a zero-test safeguard against parity cancellation.

For a logic and verification audience, the central result is not that tensors replace all existing methods. It is that exact algebra, relational elimination, decision diagrams, and temporal reasoning can be integrated while keeping their proof contracts distinct. The available results justify scoped technical assessment of this approach, with language and artifact boundaries stated explicitly.

Appendix A. Supplementary reproduction

The fresh supplement consists of `anc/validate_tt.py` and `anc/new_validation.json`. It requires Python and NumPy. From the anc directory:

```
python validate_tt.py
```

This regenerates all five truth tables, performs three exact TT constructions for each, compares every reconstructed entry, checks every cut rank, and runs the two zero-test controls. Historical values are transcribed in `anc/historical_results.csv` with provenance labels. The original portfolio source archive and execution logs are not reconstructed by this supplement.

References

[1] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986. doi:10.1109/TC.1986.1676819.

[2] I. V. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011. doi:10.1137/090752286.

[3] J. D. Biamonte, J. Morton, and J. W. Turner. Tensor network contractions for #SAT. *Journal of Statistical Physics*, 2015. arXiv:1405.7375.

[4] N. Shekhar, P. Kalla, and F. Enescu. Equivalence verification of arithmetic datapaths with multiple word-length operands. In *DATE*, 2006. https://my.ece.utah.edu/ kalla/papers/date06.pdf.

[5] T. Pruss, P. Kalla, and F. Enescu. Equivalence verification of large Galois field arithmetic circuits using word-level abstraction via Gröbner bases. In *DAC*, 2014. doi:10.1145/2593069.2593134.

[6] S. Zhao, S. Yan, and Y. Feng. Practical approach using a formal app to detect X-optimism related RTL bugs. *DVCon Proceedings*, 2014. https://dvcon-proceedings.org/wp-content/uploads/practical-approach-using-a-formal-app-to-detect-x-optimism-related-rtl-bugs.pdf.