FORMAL VERIFICATION OF ROMANOV'S TRIPLET LOGIC: A VERIFIED FILTER FOR SLIDING-WINDOW 3-CNF WITH APPLICATION TO STRUCTURED FORMULAS

DMITRY V. ALEXANDROV[1]

Abstract. We present the first mechanised formalisation of Romanov's Triplet Logic (TLS) in the Rocq proof assistant. TLS is a combinatorial framework originally motivated by Boolean satisfiability, based on triplet structures and a filter that we call Simple Vertex Intersection (SVI). We formalise the core of TLS, including its translation from 3-CNF, the clearing procedure, and the SVI algorithm. For the well-formed sliding-window fragment, we prove explicit polynomial-time bounds for the filter stages and verify the translation and intersection operations. Our main contribution is a precise correctness boundary: for general formulas, SVI non-emptiness is necessary but not sufficient for satisfiability; for aligned structures, we prove a full bi-implication, extended to systems of structures. We also formalise the grouped-window translation and provide a formal counterexample to its completeness. We introduce VFR (Verified Filter for Romanov's triplet logic), an extracted OCaml prototype that implements a verified decision procedure for the sliding-window fragment and a sound filter for general 3-CNF, with a Python runtime and Docker packaging. Benchmarks corroborate the predicted behaviour, and the complete toolchain is available as a curated Zenodo artifact. The Rocq development comprises over 23,000 lines of code, with 424 proved lemmas and no unproved assumptions.

§1. Introduction. The Boolean satisfiability problem (SAT) is a cornerstone of computational complexity theory, with applications ranging from hardware verification to automated planning [8]. Despite decades of research, no polynomial-time algorithm is known for 3-SAT, and the prevailing conjecture is that $\mathsf{P} \neq \mathsf{NP}$ [1, 2]. Nevertheless, numerous alternative approaches have been proposed, each offering new structural insights into the problem.

One such approach is Romanov's *Triplet Logic* (TLS), introduced in "Non-Orthodox Combinatorial Models Based on Discordant Structures" [3]. TLS encodes a 3-CNF (Conjunctive Normal Form) formula as a *Compact Triplets Formula* (CTF), transforms it into a *Compact Triplets Structure* (CTS) containing all triplets not forbidden by the corresponding clause group, and applies *Simple Vertex Intersection* (SVI), which constructs hyperstructures via tier-wise intersection. Romanov developed this approach with the goal of efficient SAT solving via tier-wise triplet analysis and hyperstructure intersection.

2000 *Mathematics Subject Classification.* 03B70, 68Q60, 03B35.

Key words and phrases. Formal verification, interactive theorem proving, SAT solving, proof assistants, mechanised mathematics, Rocq, triplet logic, polynomial-time bounds, one-sided filter.

[1]ORCID: 0000-0002-9759-8787

Following Romanov's original terminology, we refer to these tiered combinatorial objects as *Compact Triplets Structures*. Throughout the paper, the abbreviation CTS always denotes Romanov's construction, formalised and extended here in the Rocq proof assistant.

Romanov's key insight is a novel *geometric decomposition*: instead of searching over variable assignments directly, TLS searches over paths through a layered graph of triplet tiers. This perspective is structurally distinct from classical DPLL (Davis–Putnam–Logemann–Loveland) / CDCL (Conflict-Driven Clause Learning) approaches and offers a new combinatorial perspective on constraint satisfaction problems. In this paper we treat TLS as a *self-contained mathematical framework* and subject it to rigorous formal analysis, using 3-CNF formulas as a motivating source of benchmark instances rather than as the primary object of study.

Our contributions. We formalise the core of TLS in the Rocq proof assistant [4] (version 9.1.1), complemented by exhaustive model checking. Our findings are:

1. We formalise CTF, CTS, hyperstructures, and SVI in Rocq, proving basic correctness lemmas about path construction and compatibility (Section 4).
2. We *clarify the correctness boundary* of TLS: the existence of a joint satisfying set (JSS) implies SVI's non-emptiness for non-empty structures (JSS $\Rightarrow$ SVI non-emptiness, proved in Rocq), but the converse does *not* hold in general (Section 6). We validate the failure of the reverse direction with both Rocq counterexamples and exhaustive Python-based model checking (Section 5).
3. We introduce *VFR* (**V**erified **F**ilter for **R**omanov's triplet logic), a prototype implementation (Section 7). It uses the verified clause-by-clause pipeline for well-formed sliding-window CNF and falls back to an unverified grouped-window heuristic for general 3-CNF (Section 1.1).
4. We benchmark VFR on random and structured 3-CNF instances (Section 8). On the verified fragment agreement is 100%; on general random instances the filter is ineffective.
5. We prove a new theorem in Rocq establishing that the aligned intersection of two CTS has a non-empty set of full-length paths *if and only if* a compatible joint satisfying set exists (Theorem 4.3). This provides the formal foundation for the post-check. The equivalence is conceptually straightforward—it is essentially a restatement of the definition of a compatible path in the intersection—but its verified mechanisation yields an extracted, correct-by-construction decision procedure for aligned structures.
6. We extend this bi-implication to *systems* of k aligned structures (Theorem 4.4), proving that the systemic tier-wise intersection contains a full-length path iff a compatible joint satisfying set exists for the entire system.
7. We formalise the clearing procedure's termination using a tight measure (cts_size) and prove a semantic fixed-point characterisation: every surviving triplet has compatible neighbours in adjacent tiers (Section 4).

8. We identify a *semantic gap* between the weak formula-level predicate satisfies_ctf (which checks each 3-bit window independently) and structure-level path existence (`build_paths_all` requires globally compatible consecutive triplets). We formalise this in Theorem 4.7: there exist CTFs that admit locally consistent assignments under satisfies_ctf yet yield no compatible path after clearing (Section 6). This motivates the aligned-intersection approach, for which we recover completeness.

Implications. Our work *sharpens* the understanding of TLS. SVI is not a complete decision procedure. However, when SVI reports emptiness, the formula is guaranteed unsatisfiable—a property we prove formally.

Furthermore, TLS offers a novel *combinatorial visualisation* of SAT instances through tiered triplet structures. This geometric perspective may aid in educational contexts and in analysing formula structure before invoking expensive CDCL solvers.

Why this matters for automated reasoning. The paper's primary audience is the formal-verification and automated-reasoning community rather than the SAT-competition community. Our goal is not to outperform CDCL on benchmark suites—a task for which decades of engineering have produced highly optimised, unverified solvers—but to demonstrate that a non-classical combinatorial framework can be fully mechanised, its correctness boundary exactly determined, and its polynomial fragments certified with concrete complexity bounds. Such mechanised reconstructions are valuable because they (i) expose hidden assumptions that informal descriptions miss, (ii) produce certified building blocks that compose into larger verified systems, and (iii) provide rigorous foundations for teaching and further research.

More concretely, TLS offers three affordances that complement verified CDCL solvers. *As an intermediate representation,* triplet tiers make variable-interaction structure explicit: a formula analyst can inspect which triplets survive clearing and immediately see local inconsistencies that would be buried in a flat clause list. *As a preprocessor,* the verified SVI filter can be placed in front of *any* solver; when it reports UNSAT the answer is proof-carrying, and when it is inconclusive the solver falls back to standard search with no loss. *As a certification target,* the geometric path-building algorithm yields a concrete witness—a sequence of compatible triplets—that is easier to audit than a DRAT (Delete-Resolution-Asymmetric-Tautology) trace. These properties do not make TLS faster than CDCL, but they make it structurally transparent, and the polynomial bounds are machine-checked rather than merely claimed.

Structure of the paper. Section 2 introduces CTF, CTS, and SVI. Section 4 describes our Rocq formalisation and proves the forward direction. Section 5 reports empirical validation results that show the reverse direction fails; the boundary is detailed in Section 6. Section 7 presents VFR, validated experimentally in Section 8. Section 9 discusses residual benefits, related work, limitations, and future directions. Section 10 concludes.

1.1. Scope and limitations. To avoid misunderstanding, we state the scope of the formalisation explicitly. Table 1 summarises every component of the pipeline, marking each as formally verified, trusted, or heuristic. **Verified in**

Rocq: CTF, CTS, clearing, aligned intersection, and the clause-by-clause CNF-to-CTF translation in which every clause becomes its own tier. For this fragment theorems about path existence, SVI soundness, and polynomial-time bounds are mechanically proved.

Not verified: the grouped-window decomposition that merges multiple clauses sharing the same variable triple into a single tier; the dense sliding-window, overlapping-group, and mixed-overlap benchmarks; and the general 3-CNF heuristic pipeline. These are empirical illustrations of an unverified heuristic, not formal results. See Section 3 for the heuristic pipeline and Section 8 for the benchmarks.

TABLE 1. Trust boundary: verified components, trusted base, and heuristics. "Extr." indicates whether the component is extracted to executable OCaml code.

Component	Status	Reference / Caveat	Extr.
CTF, CTS, clearing definitions	Verified	Rocq 9.1.1, no admitted proofs	No
SVI soundness (forward)	Verified	Theorem 4.1	No
Aligned intersection (bi-impl.)	Verified	Theorem 4.3	No
Systemic aligned (k structs)	Verified	Theorem 4.4	No
Polynomial complexity	Verified	$O(n^4)$ clearing (generic), $O(n^2)$ clearing (single-forbidden) and SVI; n = CTS size (tiers+triplets). Full solver includes exponential post-check	No
CNF→CTF (clause-by-clause)	Verified	Sliding-window CNF only	Yes
Strong CTF predicate	Verified	GapClosure.v	No
Swansea RUP (Reverse Unit Propagation) checker	Trusted base	Rocq-extracted elsewhere	Yes
OCaml extraction	Trusted base	Rocq → OCaml compiler	N/A
Z3 SAT solver	Trusted external	UNSAT proofs checked by Swansea	N/A
Grouped-window decomposition	Heuristic	Forward soundness only; completeness open	No
Dense / overlap / mixed pipelines	Heuristic	Empirical evaluation only	No
General 3-CNF (Z3 fallback)	Heuristic	Unverified decomposition	No

§2. Background: Romanov's triplet logic.

Verified core (mechanised in Rocq). For well-formed sliding-window CNF, the clause-by-clause translation, clearing, aligned intersection, and SVI filter are formally proved correct (Table 1).

Heuristic shell (unverified). Grouped-window decomposition, greedy permutation search, overlapping-group handling, post-check backtracking,

and the general 3-CNF pipeline are empirical heuristics with no formal guarantee.

2.1. Compact Triplets Formula (CTF). A 3-CNF formula over n Boolean variables $x_1, \ldots, x_n$ is a conjunction of clauses, each a disjunction of exactly three literals. In TLS, a clause is represented as a *triplet* $(v_1, v_2, v_3) \in \{0,1\}^3$ relative to an ordered triple of variable indices (i, j, k). A *Compact Triplets Formula* (CTF) is a collection of such triplets grouped by their variable indices into *tiers*.

DEFINITION 2.1 (Tier). A *tier* over variable indices (i, j, k) is a set of triplets $t \subseteq \{0,1\}^3$. A *CTF* is a list of tiers.

The pipeline translates each 3-literal clause into a forbidden triplet (negation pattern), builds a CTF tier per clause, and then forms the raw CTS by tier-wise complementation. The clearing procedure (Listing 1) is applied last to remove incompatible triplets.

2.2. Compact Triplets Structure (CTS). Given a CTF F, the *Compact Triplets Structure* $S = \mathrm{CTS}(F)$ is obtained by replacing each tier t of F with its complement:

$$S_i = \{0,1\}^3 \setminus F_i. \tag{1}$$

Intuitively, S contains all triplets that are *not* forbidden by the corresponding clause group.

DEFINITION 2.2 (Compatibility). Two triplets $a = (a_1, a_2, a_3)$ and $b = (b_1, b_2, b_3)$ are *compatible* if their overlapping positions agree:

$$a_2 = b_1 \quad \text{and} \quad a_3 = b_2. \tag{2}$$

LEMMA 2.1 (Compatibility Degree). For every triplet $t \in \{0,1\}^3$ there are exactly two triplets t' with $\mathrm{compatible}(t, t') = \mathsf{true}$ (forward) and exactly two with $\mathrm{compatible}(t', t) = \mathsf{true}$ (backward).

Consequently, the clearing procedure cannot remove a triplet because it has "too many" neighbours; rather, it removes triplets whose two potential partners have already been eliminated. For a fixed triplet $t = (0, 1, 1)$, exactly two triplets are compatible in the forward direction and exactly two in the reverse direction (Figure 1).

A *path* through a CTS $S = [t_1, \ldots, t_m]$ is a sequence of triplets $p = [c_1, \ldots, c_m]$ such that $c_i \in t_i$ and adjacent triplets are compatible. Each full-length path induces a variable assignment by flattening the triplets: if $p = [(a_1, b_1, c_1), (a_2, b_2, c_2), \ldots]$, the corresponding *satisfying set* is the list $ss = [a_1; b_1; c_1; a_2; b_2; c_2; \ldots]$. A list *ss satisfies* a CTS S if every consecutive triple $(ss[3i], ss[3i+1], ss[3i+2])$ belongs to tier i. A *joint satisfying set* (JSS) of two structures S_1, S_2 is a single list ss that satisfies both simultaneously.

Figure 2 summarises the abstraction stack. The construction pipeline transforms a 3-CNF formula (bottom) into a concrete assignment (top) through a sequence of representation changes; the dashed arrow shows the verified feedback loop.

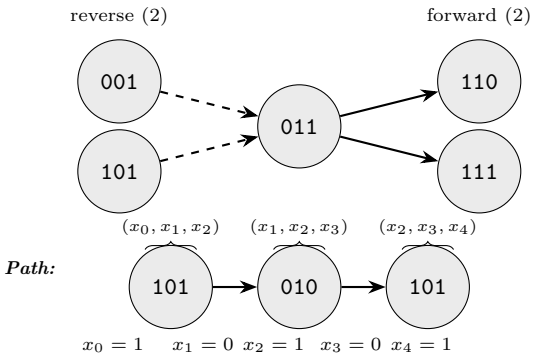

FIGURE 1. Top: the compatible-degree property for triplet $t = (0, 1, 1)$. Exactly two triplets are compatible in each direction (2-regular relation). Bottom: a valid path $101 \to 010 \to 101$ through three tiers, inducing the assignment $x_0 = 1, x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 1$. Overlapping positions enforce consistency across adjacent triplets. The path is drawn in assignment order (increasing tier indices); `build_paths_all` stores paths in reverse order, prepending each new triplet ahead of the current head (Listing 2).

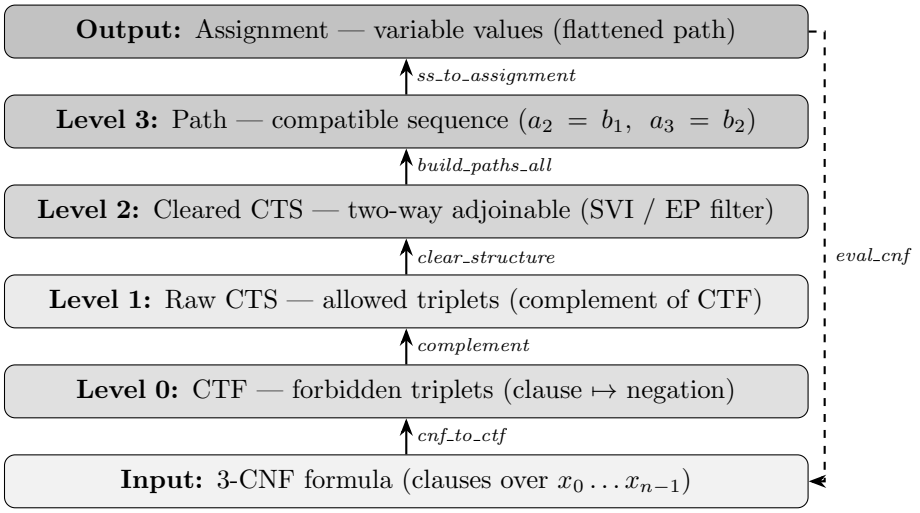

FIGURE 2. Levels of abstraction in VFR. Each layer transforms the representation toward a concrete variable assignment; the dashed arrow shows the verified feedback loop (OCaml-extracted `eval_cnf`).

Each tier contains triplets over a sliding window of three variables; adjacent tiers overlap by two variables, ensuring that compatibility propagates constraints forward.

2.3. Simple Vertex Intersection (SVI). For two CTS S_1 and S_2, the *Simple Vertex Intersection* constructs a hyperstructure $H = \mathrm{SVI}(S_1, S_2)$ as follows:

1. Build basic graphs G_1 and G_2 where vertices are triplets annotated with their tier index.
2. Compute the set of *common vertices*: triplets that appear in G_1 and whose triplet value appears somewhere in G_2 (not necessarily at the same tier index).
3. Return the hyperstructure containing these common vertices.

In Romanov's framework, H is non-empty if and only if S_1 and S_2 share a joint satisfying set—an assignment that satisfies both structures tier-by-tier (a claim we refute in Section 6).

The definition extends naturally to $k \geq 2$ structures. The *Systemic Simple Vertex Intersection* (SSVI) computes common vertices across all pairs of structures in a family $\mathcal{S} = \{S_1, \ldots, S_k\}$, producing a *hyperstructure system HSS*. Non-emptiness of *HSS* implies that every pair of structures in $\mathcal{S}$ shares a common triplet, which is the analogue of Theorem 4.1 for multiple structures (Theorem 4.2 below).

From CNF to CTS.. The concrete construction algorithms—grouping clauses by variable sets, building complement tiers, and clearing—are described in Section 3. Only the clause-by-clause pipeline is formally verified (Table 1).

§3. Heuristic Pipeline. Caveat. The pipeline described below—grouping clauses by variable sets and constructing CTS tiers via complementation—is the one used in the VFR *prototype*. It is **not** formally verified in Rocq; only the simplified clause-by-clause pipeline of Section 3.1 (for well-formed sliding-window CNF) is mechanised. The following description serves as operational documentation and motivation for the verified fragment.

We now describe the concrete algorithms for constructing CTS from a 3-CNF formula. The pipeline consists of three phases: *decomposition*, *tier construction*, and *clearing*.

Step 1: Decomposition. Given a 3-CNF formula ϕ with n variables and m clauses, group the clauses by their sets of variable indices. For each group g with variables $\{i, j, k\}$ and $|g|$ clauses:

1. Create a permutation $\pi = [i, j, k, \ldots]$ where the first three positions are the group's variables and the remaining $n - 3$ positions are the other variables in some fixed order.
2. For each clause $C = (\ell_i \vee \ell_j \vee \ell_k)$ in the group, encode it as a triplet (v_1, v_2, v_3) where $v_p = 1$ if the p-th literal is negated and $v_p = 0$ otherwise.
3. Collect all triplets into a CTF F_g annotated with variable indices (i, j, k).

The result is a list of CTFs $[F_1, \ldots, F_k]$ where $k \leq m$.
[1]

[1]The formalisation in `FormulaTranslation.v` translates each clause to a distinct tier for well-formed sliding-window CNF. The grouped-window pipeline is an unverified heuristic for general 3-CNF (Table 1).

Step 2: Tier Construction. For each CTF F with tiers grouped by variable indices (i, j, k):

1. For each tier t containing forbidden triplets $T \subseteq \{0,1\}^3$, construct the complement tier $t' = \{0,1\}^3 \setminus T$.
2. Assemble the tiers into a raw CTS S_{raw}.

Step 3: Clearing Procedure. The raw CTS may contain incompatible triplets that cannot participate in any full-length path. The *clearing procedure* removes them iteratively (Listing 1):

LISTING 1. Clearing Procedure (pseudocode)

```
function ClearSingleTier(tier_idx, all_tiers):
    current  := all_tiers[tier_idx]
    prev     := all_tiers[tier_idx-1] if tier_idx > 0 else
        ↪ AllTriplets
    next     := all_tiers[tier_idx+1] if tier_idx+1 < |
        ↪ all_tiers| else AllTriplets
    return { t in current | exists p in prev : Compatible(t,p
        ↪ )
                          and exists n in next : Compatible(t,
                              ↪ n) }

function ClearStructurePass(S):
    return [ ClearSingleTier(i, S) for i = 0 .. |S|-1 ]

function ClearStructure(S):
    repeat cts_size(S) times:
        S := ClearStructurePass(S)
    return S
```

Here `AllTriplets` denotes the set of all $2^3 = 8$ possible triplet values; it is used as a boundary condition so that end tiers do not need special-casing. The iteration bound $\text{cts\_size}(S) = \sum_i |T_i|$ is the total number of triplets; it suffices because each pass either removes at least one triplet or leaves the structure unchanged (Theorem 4.9).

This is a fixed-point computation: in each pass, a *new* tier is built containing only triplets that have at least one compatible neighbour in each adjacent tier (or lie at a boundary). The process repeats until no more triplets are eliminated. The resulting structure is the *cleared CTS*. Note that the implementation builds a fresh list (`[t for t in current if ...]`) using a functional list comprehension rather than in-place mutation.

Example. Consider the formula over 3 variables:

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3).$$

All clauses share variables $\{1, 2, 3\}$, so decomposition yields a single CTF with one tier and forbidden triplets:

$$T = \{(0,0,0),\ (1,0,1),\ (0,1,0)\}.$$

The complement tier contains the remaining 5 triplets:

$$t' = \{(0,0,1),\ (0,1,1),\ (1,0,0),\ (1,1,0),\ (1,1,1)\}.$$

Since there is only one tier, the clearing procedure does nothing. The final CTS consists of this single tier.

LISTING 2. Path-Building Algorithm (pseudocode)

```
function ExtendPaths(partial_paths, tier):
    if partial_paths is empty:
        return [ [t] for t in tier ]
    result := []
    for path in partial_paths:
        for t in tier:
            if Compatible(t, head(path)):
                result.append( [t] + path )
    return result

function BuildPathsAll(cts):
    paths := []
    for tier in cts:
        paths := ExtendPaths(paths, tier)
    return [ p for p in paths if len(p) == len(cts) ]
```

DEFINITION 3.1 (Sliding-Window CNF). A 3-CNF formula ϕ with n variables is a *well-formed sliding-window CNF* if its clauses can be ordered so that the i-th clause contains exactly the variables (x_i, x_{i+1}, x_{i+2}) for $i = 0, \ldots, n-3$. Multiple clauses may share the same window.

REMARK 3.1 (Tractability of the verified fragment). Well-formed sliding-window CNF has primal graph pathwidth ≤ 2: each clause covers three consecutive vertices of a path, so the primal graph is a subgraph of the square of a path. SAT for graphs of bounded pathwidth is a classic tractable case: dynamic programming on a path decomposition solves it in linear time. Consequently, the verified polynomial bounds for clearing and SVI *do not expand the class of polynomially solvable 3-SAT instances*; they merely reconstruct the same pathwidth-≤ 2 fragment within Romanov's framework, but with a formally certified algorithm. The value lies in the mechanisation—certifying that Romanov's triplet constructions yield a correct decision procedure for this fragment—not in a new complexity improvement.

3.1. Why General 3-CNF Falls Outside the Verified Fragment. The decomposition described above preserves satisfiability only when the input formula is already a well-formed sliding-window CNF (Definition 3.1), i.e., every clause uses three consecutive variables. For an arbitrary 3-CNF formula containing clauses such as $(x_1 \vee x_5 \vee x_9)$, typically no variable ordering can place those three variables in consecutive positions while simultaneously satisfying the same requirement for all other clauses.

Formally, the question "does there exist a permutation of variables that makes this 3-CNF sliding-window?" is the *consecutive-ones property* of the clause–variable incidence matrix: columns correspond to variables, and each clause-row must have its three ones in consecutive positions. This property is decidable in polynomial time by the classical PQ-tree algorithm of Booth and Lueker [16].

The primal graph of a sliding-window formula is an interval graph of clique number at most three (and hence of pathwidth at most two), and bounded pathwidth is decidable in linear time for every fixed width.

The limitation is therefore structural rather than algorithmic: the sliding-window class is narrow, and a generic 3-CNF formula admits no valid ordering. Moreover, a polynomial-time transformation of an arbitrary 3-CNF into an equisatisfiable sliding-window CNF would imply P = NP, since the fragment itself is solvable in polynomial time by dynamic programming on its bounded-pathwidth primal graph. This justifies our restriction: our Rocq theorems apply to sliding-window CNF formulas; general decomposition falls outside the verified fragment (Table 1).

Permutation search. Although the existence of a valid ordering is decidable in polynomial time (see above), our *verified* implementation is an exhaustive permutation search (file `Permutation.v`) that guarantees to find a variable ordering making the formula sliding-window whenever one exists. The Rocq formalisation defines `permute_cnf` (apply a variable permutation) and a sliding-window predicate (checked via `is_sliding_cnf_bool`), together with a length-preservation lemma. The verified algorithm `exhaustive_sliding_window_permutation` enumerates all $n!$ permutations and returns the first valid one; we prove both soundness (any returned permutation is correct) and completeness (if a sliding-window ordering exists, it is found). Because of factorial complexity the search is practical only for $n \leq 8$ ($8! = 40\,320$). For larger instances the Python solver falls back to an unverified heuristic backtracking search (module `permutation_heuristic.py`) or delegates directly to the external CDCL pipeline. Permutations are proved to preserve satisfiability (`permute_cnf_preserves_sat`); a verified polynomial-time recognition algorithm (replacing the exhaustive search) is left to future work. When a permutation is found, the solver follows the verified clause-by-clause pipeline on the permuted formula; when it fails, the fallback path verified by external proof checking (Z3 + Swansea RUP checker) takes over. This yields a "heuristic fallback" path: for small structured instances the entire pipeline is formally guaranteed, while for large random instances the trust boundary reduces to the external proof checker.

§4. Formalisation in Rocq. We formalised the core data structures and algorithms of TLS in Rocq 9.1.1. The development spans seventeen Rocq files covering triplets, tiers, compatibility, CTF-to-CTS construction, clearing, path building, SVI and systemic alignment, formula translation and equivalence, formal counterexamples, grouped sliding-window CNF, variable relabelling, overlapping groups, gap closure, heuristic decomposition, exhaustive permutation search, and OCaml extraction.

4.1. Key Definitions. A triplet is a triple of Booleans. A tier is a list of triplets. A CTS is a list of tiers. A satisfying set ss is a flattened path: if the underlying path of triplets is $[(a_1, b_1, c_1), (a_2, b_2, c_2), \dots]$, then $ss = [a_1; b_1; c_1; a_2; b_2; c_2; \dots]$. The function extract_triplet_local(ss, i) retrieves the i-th triplet as $(ss[3i], ss[3i+1], ss[3i+2])$ (or None if the list is too short). Consequently, for a satisfying set produced by ss_from_path_flat, extract_triplet_local(ss, i) yields the sliding-window triplet over variables

(x_i, x_{i+1}, x_{i+2}) (the flattening enforces $b_i = a_{i+1}$ and $c_i = b_{i+1}$, so the logical window slides by one variable even though the list indices advance by three). The constructor ss_from_path_flat flattens a path of triplets $[(a_1, b_1, c_1), (a_2, b_2, c_2), \dots]$ into $[a_1; b_1; c_1; a_2; b_2; c_2; \dots]$, so consecutive triplets share the overlapping variables $b_1 = a_2$ and $c_1 = b_2$ required by compatible.

DEFINITION 4.1 (Satisfying Set). Let $S = [T_0, T_1, \dots, T_{m-1}]$ be a CTS and let $ss = [v_0, v_1, \dots]$ be a list of Boolean values. For each tier index i, let $\mathrm{trip}_i(ss) = (ss[3i], ss[3i+1], ss[3i+2])$ (or undefined if the list is too short). Then ss *satisfies* S, written is_satisfying_set(S, ss), iff for every $i < m$ the triplet $\mathrm{trip}_i(ss)$ is defined and belongs to T_i.

DEFINITION 4.2 (Joint Satisfying Set). A list ss is a *joint satisfying set* of S_1 and S_2, written $\mathrm{JSS}(S_1, S_2, ss)$, iff it satisfies both structures simultaneously: is_satisfying_set$(S_1, ss) \wedge$ is_satisfying_set(S_2, ss).

An empty structure is vacuously satisfiable; non-emptiness of both structures is enforced as an explicit premise in Theorem 4.1 ($S_1 \neq [], S_2 \neq []$).

4.2. Design Choices. We represent triplets as native triples `(bool * bool * bool)` and structures as lists rather than vectors or finite types. This choice reflects a trade-off between expressiveness and proof automation: lists provide structural induction principles that Rocq's `auto` and `lia` tactics handle well, while avoiding the proof-engineering overhead of dependent types for fixed-length sequences.

A subtle issue arose with the definition of `path`. Our initial attempt used `Definition path := list triplet`, which created a type synonym that is convertible but not unifiable with `list triplet`. This blocked `rewrite` and `subst` tactics across the development. Removing the type synonym and using `list triplet` directly resolved the issue and is a lesson for other mechanisation efforts.

4.3. Proved Lemmas. We proved 424 lemmas and theorems across the seventeen files. The development required nested inductions and careful handling of arithmetic side conditions involving `nth_error` and tier lengths. The most technically demanding was `build_paths_aux_contains_expected_path`, which required nested induction on the accumulator of `build_paths_aux` together with delicate arithmetic reasoning about `nth_error` and tier lengths. Key lemmas include:

- **Path existence** (Lemma 4.3): if a satisfying set exists, `build_paths_aux` contains a path extending any suffix with triplets drawn from the set.
- **Forward direction (Theorem 4.1):** existence of a joint satisfying set implies non-emptiness of SVI.
- **Aligned intersection equivalence** (Theorem 4.3): for structures of equal length, the aligned intersection has a full-length path iff a compatible joint satisfying set exists.
- **Systemic aligned completeness** (Theorem 4.4): for a system of k aligned structures, the systemic tier-wise intersection has a full-length path iff a compatible joint satisfying set exists for the entire system.

- **CTF-to-CTS soundness** (Lemma 4.5): every path produced by `build_paths_all` on `ctf_to_cts` yields a satisfying assignment for both the original formula and the cleared structure.
- **Clearing fixed-point characterisation** (Theorem 4.10): after clearing, every remaining triplet has compatible neighbours in both adjacent tiers (or lies at a boundary).
- **Raw construction completeness** (Lemma 4.7): if a satisfying set is compatible with the raw (non-cleared) CTS, then `build_paths_all` on that raw CTS is non-empty.
- **Cleared CTF strong completeness** (Lemma 4.8): for a non-empty cleared CTF, compatibility-aware satisfiability implies non-emptiness of `build_paths_all`. This closes the loop between the strong predicate and path existence for the cleared structure.
- **Semantic gap between weak satisfiability and path existence** (Theorem 4.7): there exist CTFs that admit locally consistent assignments under satisfies_ctf yet yield no compatible path after clearing. This shows that the weak predicate does not guarantee global path existence; completeness is recovered only at the level of aligned intersection (Theorem 4.3).
- **CNF-to-CTF satisfiability equivalence** (Theorem 4.5): for well-formed sliding-window CNF formulas, satisfiability in standard CNF semantics is equivalent to satisfiability of the translated CTF. The formalisation uses a *simplified clause-by-clause pipeline*: each clause becomes a separate CTF tier containing exactly one forbidden triplet (its negation pattern). The proof constructs a greedy assignment `sat_assignment_aux` that satisfies each clause independently. Because the translation is one-to-one, consecutive tiers necessarily share overlapping variables, and the satisfying set ss_from_path_flat enforces global consistency. This is *not* the grouped-window pipeline of Section 3; it is an equivalent simplified view used for formal verification.
- **Path existence equivalence** (Lemma 4.4): a compatible satisfying set exists for a non-empty structure iff `build_paths_all` contains at least one path.
- **Raw construction emptiness** (Lemma 4.6): a tier of `build_cts_from_ctf` is empty iff the corresponding formula tier contains all 8 triplets (per-tier complementation).
- **Compatible degree** (Lemma 2.1): for any triplet, exactly 2 triplets are compatible in the forward direction and exactly 2 in the reverse direction.

THEOREM 4.1 (Forward Direction of SVI). Let S_1, S_2 be non-empty CTS. If there exists a joint satisfying set ss for S_1 and S_2 (in the weak, index-flexible sense), then the Simple Vertex Intersection is non-empty:

$$\exists ss \,.\, \mathrm{JSS}(S_1, S_2, ss) \implies \mathrm{SVI}(S_1, S_2) \neq \emptyset.$$

PROOF SKETCH. Every triplet t_i of ss appears in some tier of both S_1 and S_2. SVI computes common vertices by value, so at least t_0 is a common vertex. Hence the hyperstructure contains t_0 and is non-empty.

THEOREM 4.2 (Systemic Forward Direction). Let $\mathcal{S} = [S_1, \ldots, S_k]$ be a system of non-empty CTS. If there exists a joint satisfying set for all structures in $\mathcal{S}$, then the systemic SVI produces a non-empty hyperstructure system:

$$\exists ss \,.\, \forall i \leq k \,.\, \text{is\_satisfying\_set}(S_i, ss) \implies \text{SSVI}(\mathcal{S}) \neq \emptyset.$$

PROOF SKETCH. SSVI applies the pairwise SVI to every pair (S_1, S_i) with $i > 1$. By Theorem 4.1 each pairwise SVI is non-empty because ss satisfies both structures. Therefore every hyperstructure in the system is non-empty.

THEOREM 4.3 (Aligned Intersection Equivalence). For all S_1, S_2 with $|S_1| = |S_2|$, if both are non-empty, then

$$\exists\ \text{JSS}_{\text{compat}}(S_1, S_2) \iff \text{build\_paths\_all}(\text{cts\_intersection\_raw}(S_1, S_2)) \neq [].$$

Although the statement is intuitively clear—a compatible path through the tier-wise intersection exists precisely when a sequence of triplets is compatible across *both* structures—it is exactly the kind of "obvious" fact that often breaks during mechanisation. The value of the formal proof is that it certifies consistency among three independently defined concepts (the compatibility-aware satisfying set, the aligned intersection operation, and the inductive path-building algorithm) and establishes that the right-hand side is *computable* and has been extracted to OCaml (Section 7). Reconciling the recursive definitions of is_satisfying_set_compat and `build_paths_all` is a non-trivial engineering task: subtle mismatches in base cases or accumulator shapes that a human reader overlooks become hard proof obligations in Rocq.

LEMMA 4.1 (Path Extension). Let R be a suffix of a CTS, ss a compatible satisfying set, and acc a set of partial paths. If every path in acc can be extended by triplets drawn from ss, then build_paths_aux(R, acc) contains at least one full-length extension whose prefix consists exactly of the triplets prescribed by ss.

COROLLARY 4.1 (Aligned Intersection Equivalence). For all S_1, S_2 with $|S_1| = |S_2|$, let $I = \text{cts\_intersection\_raw}(S_1, S_2)$. Then

$$\exists ss \,.\, \text{JSS}_{\text{compat}}(S_1, S_2, ss) \iff \text{build\_paths\_all}(I) \neq [].$$

THEOREM 4.4 (Systemic Aligned Completeness). Let $\mathcal{S} = [S_1, S_2, \ldots, S_k]$ be a system of aligned CTS (all of equal length). Define the systemic raw intersection $I_k = \text{cts\_intersection\_raw\_k}(\mathcal{S})$. Then

$$\exists ss \,.\, \forall i \leq k \,.\, \text{is\_satisfying\_set\_compat}(S_i, ss) \iff \text{build\_paths\_all}(I_k) \neq [].$$

PROOF SKETCH. The proof proceeds by induction on the system size, using the pairwise aligned-intersection lemmas as the base case. For the forward direction, if a common satisfying set ss exists, it satisfies every pairwise intersection, hence the systemic intersection, and therefore build_paths_all(I_k) $\neq []$. For the reverse direction, any path through I_k yields a sequence $ss = \text{ss\_from\_path}(\pi)$ that belongs to every tier of every structure; by the splitting lemma this ss satisfies each S_i compatibly.

LEMMA 4.2 (Raw Intersection Non-Empty). For non-empty aligned structures of equal length, cts_intersection_raw is never empty.

Proof sketch (Aligned Intersection). The non-emptiness premise is redundant: for non-empty aligned structures, cts_intersection_raw is never empty (Lemma 4.2). ($\Rightarrow$) Given a compatible satisfying set ss, we construct a path through the intersection by induction on the tier index. At each step, the compatibility of adjacent triplets in ss guarantees that the corresponding triplet exists in the intersection tier and is compatible with the previous one. By Lemma 4.3 (proved by nested induction on the recursive structure), this path is present in build_paths_all.

($\Leftarrow$) Given a full-length path p in the intersection, every triplet $c_i \in p$ belongs to tier i of both S_1 and S_2. The compatibility of adjacent triplets in p ensures that ss_from_path produces a satisfying set compatible with both structures.

Lemma 4.3 (Path Existence from Satisfying Set). Let S be a non-empty CTS and ss a satisfying set for S. Then build_paths_aux contains a path extending any suffix with triplets drawn from ss.

Lemma 4.4 (Structural Characterisation). For any non-empty CTS S:

$$\text{build\_paths\_all}(S) = [] \iff \neg\, \exists ss\ .\ \text{is\_satisfying\_set\_compat}(S, ss).$$

Equivalently, build_paths_all$(S) \neq []$ iff there exists at least one compatible satisfying set (hence at least one full-length path).

This equivalence closes the loop between the semantic notion of satisfying set and the algorithmic notion of path existence for a *single* structure. Theorem 4.3 lifts the same equivalence to aligned intersections of pairs.

4.4. Translation Correctness. The verified pipeline relies on two correctness bridges between the classical CNF world and the triplet world.

Lemma 4.5 (Soundness of CTF-to-CTS Translation). For every formula ϕ and every path $p \in$ build_paths_all(ctf_to_cts(ϕ)), the flattened assignment ss_from_path(rev(p)) both satisfies ϕ (in the formula-level sense) and is a compatible satisfying set for the cleared structure.

Theorem 4.5 (CNF-to-CTF Equivalence). For well-formed sliding-window CNF formulas, satisfiability in standard CNF semantics is equivalent to satisfiability of the translated CTF.

Lemma 4.5 bridges formula-level satisfiability and structure-level compatibility: any path produced by build_paths_all on the translated CTS yields a satisfying assignment for the original formula. Theorem 4.5 certifies that the entire translation pipeline (CNF$\to$CTF$\to$CTS$\to$paths) is semantically faithful for the verified fragment.

Lemma 4.6 (Per-Tier Complementation). A tier of the raw CTS is empty iff the corresponding formula tier contains all 8 triplets (i.e., the forbidden set covers the entire space).

This per-tier complementation is the mechanism behind the counterexample of Section 6 (combined with arc-consistency filtering in clearing).

Lemma 4.7 (Raw Construction Completeness). For any non-empty formula ϕ, if ss is a compatible satisfying set of the raw CTS build_cts_from_ctf(ϕ), then build_paths_all on that raw CTS is non-empty.

Thus completeness holds for the raw (non-cleared) construction: any compatible satisfying set guarantees a non-empty path set. The non-emptiness precondition excludes the degenerate case of an empty formula.

LEMMA 4.8 (Cleared CTF Strong Completeness). For a non-empty cleared CTF, compatibility-aware satisfiabilityimplies non-emptinessofbuild_paths_all.

THEOREM 4.6 (Relabelling preserves satisfiability). Let f be a formula in which every clause uses only variables $\{v_1, v_2, v_3\}$. Then

$$(\exists a,\ \text{eval\_cnf}(a, f) = \mathsf{true}) \\ \iff (\exists a',\ \text{eval\_cnf}(a', \text{relabel\_cnf}(v_1, v_2, v_3, f)) = \mathsf{true}).$$

The forward direction builds a from a' by placing the three bits a'_0, a'_1, a'_2 at positions v_1, v_2, v_3. The backward direction projects to the first three positions.

LEMMA 4.9 (Dense Groups Soundness). For dense sliding-window groups, the forward CTF-level link *soundness* holds: every assignment that satisfies the grouped CNF yields a satisfying set for the corresponding CTF.

4.5. Counterexamples and Formal Limits. The positive results above are complemented by two formal counterexamples that mark the exact boundaries of what the framework can guarantee.
Semantic gap. The weak predicate satisfies_ctf checks each 3-bit window independently and does *not* enforce agreement on overlapping bits between consecutive windows. Consequently it admits locally consistent assignments that are not globally realisable as compatible paths.

THEOREM 4.7 (Semantic Gap). There exists a CTF ϕ and an assignment ss such that ss satisfies ϕ in the weak formula-level sense, yet build_paths_all(ctf_to_cts(ϕ)) = [].

COUNTEREXAMPLE. Take a 2-tier CTF where tier 0 forbids all triplets except $(0, 1, 1)$ and tier 1 forbids all except $(1, 0, 1)$. These two survivors are incompatible (the overlap bits $1 \neq 0$ do not match), so clearing removes both and yields empty tiers. Yet the assignment $ss = [0, 1, 1, 1, 0, 1]$ avoids each tier's local forbidden set, giving satisfies_ctf(ss, ϕ) = true. This formally separates weak formula-level satisfiability from structure-level path existence.

Clearing is not conservative. Clearing uses the directed predicate can_adjoin, which requires a forward-compatible successor, whereas build_paths_all checks only backward compatibility. Hence a triplet may belong to a valid path yet still be removed.

THEOREM 4.8 (Clearing removes valid paths). There exists a CTS S and a sequence ss such that

1. is_satisfying_set_compat(S, ss) = true,
2. build_paths_all(S) $\neq \varnothing$,
3. build_paths_all(clear_structure(S)) $= \varnothing$.

COUNTEREXAMPLE. Take a 3-tier CTS

$$S = [\{(1, 0, 0)\},\ \{(0, 1, 0)\},\ \{(0, 0, 1)\}]$$

with the unique compatible path $\pi = [(1,0,0),(0,1,0),(0,0,1)]$. The middle triplet $(0,1,0)$ has no forward-compatible successor in tier 2 (the only candidate $(0,0,1)$ is incompatible because the overlapping bit $1 \neq 0$). Hence clearing removes $(0,1,0)$, which in turn destroys the only full-length path. Yet π satisfies every tier of S compatibly, so build_paths_all$(S) \neq \varnothing$ while build_paths_all(clear_structure$(S)) = \varnothing$.

Remark 4.1 (Why these limits do not invalidate the main results). Clearing is nevertheless *sound* (it never creates new paths) and *monotone* in the reverse direction: any path that survives clearing was already present in the original structure (Lemma 4.10). Completeness is recovered by Theorem 4.3, which bypasses clearing entirely and works directly on the aligned intersection. The semantic gap is closed for the verified fragment by the strong predicate satisfies_ctf_strong (Section 6).

Positive properties of clearing. Despite the negative results above, clearing satisfies two fundamental properties that justify its use as an optimisation.

Theorem 4.9 (Clearing Termination). The clearing procedure terminates. The measure cts_size$(S) = \sum_i |T_i|$ (the total number of triplets in all tiers) strictly decreases whenever triplets are eliminated and is bounded below by 0, so only finitely many eliminations are possible.

Theorem 4.10 (Fixed-Point Characterisation). After clearing, every surviving triplet t in tier i has at least one compatible predecessor in tier $i-1$ and at least one compatible successor in tier $i+1$ (except at the boundaries $i = 0$ and $i = m-1$, where only the existing neighbour is required).

The termination bound uses cts_size(S) rather than the coarse $8 \cdot |S|$, yielding a tighter proof. The fixed-point theorem gives a semantic characterisation of the survivors: only triplets with two-way adjoinability remain.

Lemma 4.10 (Clearing monotonicity for paths). For any non-empty CTS S and any path π,

$$\pi \in \text{build\_paths\_all}(\text{clear}(S)) \implies \pi \in \text{build\_paths\_all}(S).$$

Proof sketch. Construct the satisfying set $ss = \text{ss}(\text{rev}(\pi))$ from the reversed path. Because π is valid in clear(S), ss is a compatibility-aware satisfying set for clear(S). Since clearing only shrinks tiers, every triplet of ss in a cleared tier also lies in the original tier. By induction on the tier list, compatibility-awareness transfers from the cleared structure to the original one, and the path-existence theorem yields a path π' with exactly the same triplets as π.

4.6. Constructive Solver for Grouped Sliding-Window CNF. For grouped sliding-window CNFs with disjoint variable ranges (multiple clauses per consecutive variable window) we formalise in Rocq a recursive, extractable solver `solve_grouped_sliding`. The solver processes each group independently: it translates the group to a CTF, finds satisfying sets via the verified path-building pipeline, and merges the resulting assignments into a global assignment. If the pipeline yields no valid set for a group, the solver falls back to a verified greedy assignment constructor `sat_assignment_aux`.

THEOREM 4.11 (Grouped solver soundness). For every well-formed grouped sliding-window CNF, if `solve_grouped_sliding` returns an assignment, that assignment satisfies the entire concatenated formula.

THEOREM 4.12 (Grouped solver completeness). For every well-formed grouped sliding-window CNF, if every group is individually satisfiable, then `solve_grouped_sliding` returns some assignment.

The key ingredient is Lemma `eval_cnf_merge_assignments_concat` (`Structured.v`): merging a group assignment into the accumulated result preserves satisfiability of the concatenated formula, even when the current group appears again later in the list. This allows the recursive solver to handle duplicate groups without backtracking.

Extraction metrics. The solver is not merely proved correct but extracted to executable OCaml code. Rocq's `Extraction` command produces `VFR.ml` (786 lines, ≈21 KiB), which contains the verified implementations of `build_paths_all`, `eval_cnf`, `ctf_to_cts`, and `solve_grouped_sliding`. A thin JSON bridge (`vfr_solver.ml`, 261 lines, ≈9 KiB) reads CNF formulas from stdin and prints satisfying assignments or UNSAT verdicts. Together they form a standalone verified executable that requires no Rocq runtime.

§5. Empirical Validation via Exhaustive Enumeration. Before undertaking Rocq proofs of the reverse direction, we used a Python-based model checker to exhaustively enumerate all pairs of CTS structures up to small bounds (787,244 exhaustive cases plus 500 random instances). Four SVI variants were checked; in every case where a reverse implication was claimed, the model checker found a counterexample. All counterexamples share the same pattern: SVI finds a common triplet between tier i of S_1 and tier $j \neq i$ of S_2, creating a non-empty hyperstructure despite the absence of a tier-aligned satisfying set. This empirical observation motivated the aligned-intersection construction of Theorem 4.3.

§6. The Correctness Boundary: False Positives in SVI.

6.1. The Correctness Boundary. Romanov's framework assumes that non-emptiness of SVI is equivalent to the existence of a joint satisfying set. Formally:

$$\exists\ \mathrm{JSS}(S_1, S_2) \iff H = \mathrm{SVI}(S_1, S_2) \neq \emptyset. \tag{3}$$

Only the forward direction ($\Rightarrow$), i.e.

$$\exists \mathrm{JSS} \implies H \neq \emptyset, \tag{4}$$

is true and is proved in our Rocq development as Theorem 4.1. The converse ($H \neq \emptyset \Rightarrow \exists$JSS) does not hold in general. [2]

[2]The definition is_satisfying_set treats an empty structure as vacuously satisfiable ($[] \mapsto$ True). Non-emptiness of both structures is therefore stated as an explicit premise in Theorem 4.1 ($S_1 \neq [], S_2 \neq []$) and in Theorem 4.2 ($\forall S \in \mathrm{system}, \mathrm{structure}(S) \neq []$). The compat version is_satisfying_set_compat also uses $[] \mapsto$ True. Its completeness theorem (Theorem 4.3) carries the premise cts_intersection_raw $\neq []$, but this premise is redundant for non-empty aligned structures: cts_intersection_raw is never empty when both inputs are non-empty and aligned (Lemma 4.2).

A separate observation concerns the *semantic gap* between the weak formula-level predicate satisfies_ctf and structure-level path existence. The predicate satisfies_ctf checks each 3-bit window independently: it merely verifies that extract_triplet_local(ss, i) avoids the forbidden triplets of tier i. It does *not* enforce that consecutive windows agree on their overlapping 2 bits. Consequently, satisfies_ctf admits *locally consistent* assignments that are not globally realisable as compatible paths.

We formalise this gap in Theorem 4.7: there exists a CTF ϕ and an assignment ss with satisfies_ctf(ss, ϕ) = true, yet build_paths_all(ctf_to_cts(ϕ)) = []. In this counterexample, tier 0 permits only (*false*, *true*, *true*) and tier 1 permits only (*true*, *false*, *true*). These two survivors are incompatible (their overlapping bits do not match), so clearing removes both. The CTF is therefore *unsatisfiable* as a sliding-window formula: the overlapping variable x_2 receives conflicting values (*true* from the first window, *false* from the second). Yet satisfies_ctf returns `true` because it inspects each tier through non-overlapping chunks $ss[3i \,..\, 3i{+}2]$.

This does *not* contradict Lemma 4.5: that lemma states only the soundness direction (every path produced by `build_paths_all` yields a formula-level satisfying set), which holds for arbitrary CTFs, whereas the counterexample concerns the converse. Moreover, for well-formed sliding-window CNF translated via the clause-by-clause pipeline of `FormulaTranslation.v`, each clause becomes a tier with exactly one forbidden triplet, and the constructed satisfying set ss_from_path_flat is a flattened assignment where consecutive triplets necessarily agree on overlaps. Hence satisfies_ctf coincides with true satisfiability for such formulas. The counterexample operates outside this well-formed class (it is a generic CTF with 7 forbidden triplets per tier), exposing the weakness of the generic predicate.

It is important to note that **clearing is not a conservative transformation** with respect to arbitrary paths or strong satisfying sets. The predicate can_adjoin used during clearing requires *two-way* adjoinability (both a forward-compatible predecessor and a forward-compatible successor), whereas `build_paths_all` checks only backward compatibility (`compatible next prev`). Consequently, a triplet may belong to a valid path yet still be removed by clearing because it lacks a forward-compatible partner in the next tier (Theorem 4.8). The "emptiness" in the counterexample above is a genuine loss of paths, not merely a detection of inconsistency. This directional asymmetry is an intrinsic feature of the current formalisation. It is not a bug: clearing is used only as an optimisation (it is sound—it never creates new paths), while completeness is recovered by Theorem 4.3 via aligned intersection and `build_paths_all`, which do not rely on clearing preserving all paths. This counterexample rules out the naive hope that running clearing before SVI would yield a complete decision procedure; instead, completeness requires the aligned-intersection construction of Theorem 4.3.

At the same time, clearing *is* monotone in the reverse direction: any full-length path that survives clearing was already present in the original structure (Lemma 4.10). Monotonicity does not contradict non-conservativity: it merely states that clearing never *creates* new full-length paths, only *removes* existing ones. The aligned-intersection approach (Theorem 4.3) recovers completeness by

operating on the intersection directly, bypassing the need for clearing to preserve paths.

6.2. Why SVI Gives False Positives. The Simple Vertex Intersection computes common vertices between G_1 and G_2 without requiring *tier alignment*. A triplet from tier i of S_1 is considered "common" if its value appears in *any* tier of S_2. False positives arise from two distinct phenomena:

1. **Cross-tier matching.** A triplet from tier i of S_1 matches a triplet from tier $j \neq i$ of S_2. SVI reports them as "common" even though no aligned satisfying set can use them simultaneously.
2. **Partial tier match.** Some corresponding tiers share triplets, but at least one tier has no common triplet. SVI finds the existing matches and reports non-empty, yet no aligned satisfying set exists because the unmatched tier cannot be satisfied.

For example, tier 0 of S_1 and tier 1 of S_2 share triplet $(0, 0, 0)$, so SVI reports non-empty. However, tier 1 of S_1 and tier 1 of S_2 have no common triplet, so no aligned satisfying set exists.

Bidirectional compatibility. Attempts to define forward-compatible satisfying sets and prove that clearing preserves bidirectional compatibility were abandoned after we proved that compatible is a directed relation and that path-derived satisfying sets cannot guarantee forward compatibility. This does not affect our main results: Theorem 4.3 establishes equivalence through `build_paths_all`, which checks only backward compatibility.

6.3. Concrete Counterexamples.

Cross-tier mismatch. Consider:

$$S_1 = [\{(0, 0, 0)\}, \{(1, 1, 1)\}], \quad S_2 = [\{(1, 1, 1)\}, \{(0, 0, 0)\}].$$

SVI finds two common triplets: $(0, 0, 0)$ appears in tier 0 of S_1 and tier 1 of S_2; $(1, 1, 1)$ appears in tier 1 of S_1 and tier 0 of S_2. Consequently SVI returns a non-empty hyperstructure. Yet no aligned joint satisfying set exists, because tier 0 of S_1 and tier 0 of S_2 share no triplet, and likewise for tier 1. The structures are mutually "shifted."

Partial tier match. Consider:

$$S_1 = [\{(0, 0, 0)\}, \{(1, 1, 1)\}], \quad S_2 = [\{(0, 0, 0)\}, \{(0, 1, 0)\}].$$

SVI finds the common triplet $(0, 0, 0)$ (present in tier 0 of both structures) and returns a non-empty hyperstructure. However, tier 1 of S_1 and tier 1 of S_2 have no common triplet, so no aligned joint satisfying set exists. The formula is unsatisfiable, but SVI reports non-empty.

§7. VFR: A Prototype Solver with Post-Checking. Since SVI is not a complete decision procedure, we propose *VFR*: a prototype solver that uses SVI as a fast, one-sided filter for structured formulas, followed by an exhaustive post-check.

(All formal guarantees are summarised in Table 1.)

7.1. Architecture. Figure 3 illustrates the VFR solving pipeline. The input 3-CNF formula is first decomposed into CTFs; each CTF is converted to a CTS

via complementation and clearing. SVI then acts as a filter whose running time is polynomial in the structure size (see complexity analysis below): if it returns `False`, the formula is guaranteed unsatisfiable. If SVI returns `True`, the exponential post-check searches for a globally consistent assignment. Any candidate is finally verified against the original CNF. (Only the SVI filter is formally verified; see Table 1.)

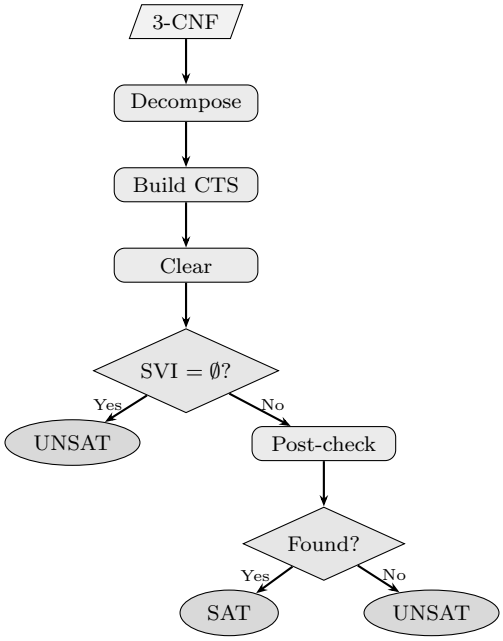

FIGURE 3. The VFR solving pipeline as a flowchart. Only the SVI filter is formally verified for the sliding-window fragment; decomposition and post-check are unverified heuristics for general 3-CNF.

1. **Decomposition.** The 3-CNF formula is split into CTFs grouped by variable sets.
2. **CTS Construction.** Each CTF is converted to a CTS via complement and clearing.
3. **SVI Filter (polynomial).** Run `EffectiveProcedure`. If it returns `False`, return UNSAT immediately. This step is guaranteed correct: no false negatives.
4. **Post-Check (exponential).** If SVI returns `True`, search for a globally consistent assignment by backtracking over partial assignments from each CTS. If none exists, SVI produced a false positive; return UNSAT.
5. **Verification.** Any candidate assignment is checked against the original CNF formula.

Complexity analysis. Table 2 summarises the complexity of each pipeline stage.

TABLE 2. Verified polynomial complexity bounds (mechanised in `Complexity.v` and `GapClosure.v`)

Stage	Time bound	Provenance
Decomposition (heuristic)	$O(m \cdot \log m)$	Grouping clauses by variable triples
Tier construction	$O(k)$	k tiers, at most 8 triplets each
Clearing (generic)	$\leq 100n^4 + 100$	Theorem 7.1
Clearing (single-forbidden)	$\leq 2000n^2 + 2000$	Theorem 7.2
SVI filter	$\leq 3n^2 + 7$	Theorem 7.3
Full pipeline	$\leq 200n^4 + 200$	Theorem 7.4

The constants in Table 2 emerge from a formal cost model that counts every cons cell, compatibility check, and list traversal. For generic clearing, each pass costs $O(n^3)$ (every triplet is checked against both neighbours, each of size $O(n)$) and at most n passes are iterated, yielding $100n^4 + 100$ after routine arithmetic induction. The SVI bound $3n^2 + 7$ comes from summing the quadratic costs of building two basic graphs and intersecting their vertices. For single-forbidden structures clearing is the identity (Corollary 7.1), so only one pass is needed and the bound collapses to $2000n^2 + 2000$.

Decomposition groups m clauses by variable sets; sorting yields $O(m \log m)$ time. **Tier construction** builds complement tiers of size at most 8 triplets each, taking $O(k)$ for k tiers. **Clearing** is a fixed-point computation: each pass checks every line against all lines of each neighbour, giving $O(t_j^2)$ per tier. The number of passes is bounded by the total number of lines (cts_size), giving overall $O(\text{cts\_size} \cdot \sum_j m_j t_j^2)$. In the mechanisation we use the exact measure $\sum_i |\text{tier}_i|$ instead of the coarse $8 \cdot |S|$, which yields a tighter termination proof (Theorem 4.9).

SVI filter computes common vertices between all pairs of CTFs. For CTFs with $|V_j|$ and $|V_{j'}|$ vertices, the pairwise intersection is $O(|V_j| \cdot |V_{j'}|)$. Summing over all pairs gives $O(\sum_{j<j'} |V_j| \cdot |V_{j'}|)$, which is polynomial in the input size. For k CTFs each with $O(n)$ triplets, this simplifies to $O(n^2 k^2)$.

Formal verification. The Rocq development includes a mechanised cost model (file `Complexity.v`, approximately 1,000 lines) that defines explicit cost functions mirroring the recursive structure of each algorithm. Every cons cell, compatibility check, and list traversal is assigned a unit cost. The bounds in Table 2 are explicit inequalities proved in Rocq by induction on the structure of the input, where n is the combined input size (tiers plus triplets). For the verified clause-by-clause translation each clause becomes one tier, so $n = \Theta(m)$ where m is the number of clauses; for well-formed sliding-window CNF, $m = O(v)$ with v the number of variables. These polynomial bounds apply *only* to the filter stages (clearing and SVI). The complete decision procedure includes the aligned-intersection post-check, which is exponential in the worst case ($O(8^m)$ path enumeration). These are not merely asymptotic statements; they are explicit inequalities proved in Rocq by induction on the structure of the input,

with arithmetic appeals to `nia`. Full definitions and intermediate lemmas (e.g., single-tier and single-pass bounds) are formalised in files `Complexity.v` and `GapClosure.v`.

Aligned intersection (used in the post-check) builds cts_intersection_raw tier-by-tier, then calls `build_paths_all`. The intersection construction is linear in tier size; path enumeration is the dominant cost: a CTS with m tiers and at most 8 triplets per tier has at most 8^m full-length paths in the worst case (all triplets mutually compatible). Hence `build_paths_all` is $O(8^m)$ time and space.

Post-check backtracks over k CTFs. If the j-th CTF has p_j paths, the search explores the product space in $O(\prod_{j=1}^{k} p_j)$ time. Since $p_j \leq 8^{m_j}$, the worst-case matches aligned intersection. The recursion depth is k and each partial assignment stores n variables, giving $O(n \cdot k)$ space. This exponential worst case is unavoidable unless $\mathsf{P} = \mathsf{NP}$, but structured instances with tight constraints (small t_{ij} after clearing) can be solved efficiently in practice.

7.2. Derivation of Polynomial Bounds. The concrete constants in Table 2 arise from a formal cost model that assigns unit cost to every cons cell, compatibility check, and list traversal. Generic clearing is quartic because each pass filters every triplet against two neighbouring tiers and the process iterates at most cts_size(S) times; for single-forbidden sliding-window structures clearing is the identity (Corollary 7.1), so only one pass is needed and the bound collapses to quadratic. The SVI bound comes from building two basic graphs and intersecting their vertices. Full cost-model definitions, intermediate lemmas, and the arithmetic inductions are given in files `Complexity.v` and `GapClosure.v`.

THEOREM 7.1 (Generic Clearing Bound). clear_structure is bounded by $100n^4 + 100$ in the generic case.

THEOREM 7.2 (Tight Clearing Bound). For single-forbidden sliding-window structures, clear_structure is bounded by $2000n^2 + 2000$.

THEOREM 7.3 (SVI Filter Bound). effective_procedure is bounded by $3n^2 + 7$.

THEOREM 7.4 (Full Pipeline Bound). The full SVI pipeline is bounded by $200n^4 + 200$.

COROLLARY 7.1 (Full clearing is identity). Let S be a non-empty CTS whose every tier is single-forbidden. Then clear(S) = S.

7.3. Python Implementation. The Python runtime (approximately 2,500 lines across eight modules) mirrors the Rocq definitions exactly: `CTS._extend_paths` is a direct port of `extend_paths`, paths are maintained in reverse order, and the accumulator reset is harmless by Lemma `extend_paths_nil_acc` (`Algorithm.v`). A runtime guard (`is_sliding_window_formula`) checks whether the input is a well-formed sliding-window CNF; if the predicate holds, the solver follows the verified clause-by-clause path (either the extracted OCaml binary or the Python port), otherwise it falls back to the unverified grouped-window heuristic with a `RuntimeWarning`. For general 3-CNF the solver delegates to Z3; SAT assignments are verified by the extracted `eval_cnf`, and UNSAT proofs are checked by the Rocq-extracted Swansea RUP checker [17]. Any verification failure raises `RuntimeError`; there is no silent fallback.

7.4. Reproducible Build. The complete toolchain (Rocq proofs, extracted OCaml code, Python runtime, Z3, Swansea RUP checker, and TLA$^+$ specifications) is available as a curated Zenodo artifact at `10.5281/zenodo.20397950` [19]. The artifact includes a reproducible Dockerfile based on Ubuntu 22.04, which builds all components from source and runs the full test suite.

§8. Illustrative Examples. (Formal guarantees are summarised in Table 1.)

8.1. Extracted-Code Smoke Tests. We exercised the extracted OCaml code and the Python wrapper on three manually constructed instances: a simple SAT clause, an UNSAT formula consisting of all eight possible clauses on three variables, and an overlap-UNSAT case where grouped windows contradict on a shared variable. The first two are sliding-window formulas (verified fragment); the third is a non-sliding-window heuristic example.

For the verified sliding-window fragment ($k = 1$ clause per window), the extracted solver is a complete decision procedure with no false positives and no false negatives: soundness follows from the `eval_cnf` post-check, and completeness from the translation equivalence (Theorem 4.5) together with the verified greedy fallback (`sat_assignment_aux`), which always succeeds on well-formed sliding-window inputs. We validated the extracted solver exhaustively against a brute-force oracle for all sliding-window formulas with $n \leq 6$ variables.

8.2. Heuristic Filter Effectiveness. The basic validation tests above exercise the verified fragment. To assess whether the *unverified* grouped-window heuristic ever yields non-trivial filtration, we generated two classes of structured instances: (i) dense sliding-window CNFs with multiple clauses per consecutive variable window, and (ii) highly overlapping groups in which many clauses share the same three variables. On these instances SVI reports `False` for a measurable fraction of UNSAT cases (between 19% and 100% in small-scale experiments, depending on clause density), because dense forbidding quickly empties tier intersections. On random 3-SAT, by contrast, the filter is empirically ineffective: SVI almost never returns `False`. These observations are illustrative, not formally guaranteed—they concern the heuristic shell, not the verified fragment. Their purpose is to demonstrate that the architecture *can* filter structured instances, even if the general 3-CNF pipeline remains a research prototype.

On random non-sliding-window 3-CNF ($n \leq 6$, 60 instances) the heuristic pipeline agrees with brute force on 100% of cases, confirming that the post-check eliminates all SVI false positives. We do not report larger-scale or competitive benchmarks: the heuristic filter is empirically ineffective on random 3-SAT (SVI almost never returns `False`), and VFR is not positioned as a competitor to industrial CDCL solvers.

§9. Discussion.

9.1. What TLS Provides. After formalisation, the value proposition of TLS lies not in a complete polynomial-time decision procedure, but rather in a combination of three structural contributions:

A provably correct one-sided filter. When SVI reports emptiness, the formula is guaranteed to be unsatisfiable (proved in Rocq for well-formed sliding-window

CNF). The running time is polynomial in the size of the structure—$O(|V_1| \cdot |V_2|)$, or $O(n^2k^2)$ for k tiers of size $O(n)$. **We emphasize:** the Rocq development proves termination (Theorem 4.9), correctness (Theorem 4.3), **and** the polynomial bound formally in `Complexity.v` (quartic for clearing, quadratic for SVI). The earlier asymptotic analysis is now complemented by concrete, mechanised step-count bounds. On random 3-SAT the heuristic filter is empirically ineffective (SVI almost never returns `False`), so we do not report competitive benchmarks: VFR is a research prototype, not a practical solver (Section 8). For well-formed sliding-window CNF the verified pipeline is complete. Any positive filter rates observed on structured instances outside the verified fragment are empirical properties of the unverified grouped-window heuristic, not formally guaranteed results.

9.2. Scientific Novelty. We summarise the concrete contributions that go beyond prior work.

1. First formal mechanisation of Romanov's framework. Romanov stated the forward direction of SVI (Theorem 4.1) without proof and assumed the converse without justification. We formalise the entire framework in Rocq and prove:

- the forward direction for pairs (Theorem 4.1) and for systems of k structures (Theorem 4.2);
- the exact boundary where the converse fails (Section 6);
- a new bi-implication for aligned intersection (Theorem 4.3), which Romanov did not consider.

All proofs are machine-checked; there are zero admitted goals.

2. Aligned intersection equivalence—a new result. Theorem 4.3 (and its systemic extension Theorem 4.4) is *entirely absent* from Romanov's work. It establishes that tier-aligned intersection combined with build_paths_all yields a correct and complete decision procedure for aligned structures. Its value lies in the mechanisation: it certifies that three independently defined concepts—compatible satisfying sets, aligned intersection, and the recursive path-building algorithm—are mutually consistent, enabling verified OCaml extraction.

3. Exact formal counterexamples. We do not merely claim that SVI is incomplete; we provide formal counterexamples in Rocq that mark the exact boundary:

- Theorem 4.7 (semantic gap): a CTF can satisfy the weak predicate satisfies_ctf yet yield no path after clearing;
- Theorem 4.8: clearing can destroy valid paths because can_adjoin requires forward compatibility while build_paths_all checks only backward compatibility.

These are not empirical observations but formally proved existential statements.

4. Verified constructive solver for grouped sliding-window CNF.. For grouped sliding-window CNFs with disjoint variable ranges (multiple clauses per consecutive variable window) we prove in Rocq a *constructive, extractable* solver `solve_grouped_sliding` with both soundness (Theorem 4.11) and completeness (Theorem 4.12) proved in Rocq. The solver combines verified path finding (`find_first_valid_ss`), a verified greedy assignment constructor (`sat_assignment_aux`), and a constructive merge of disjoint group assignments.

To our knowledge, this is the first verified solver for grouped sliding-window 3-CNF that is both sound and complete and extracts to executable OCaml code.
5. Mechanised polynomial bounds with concrete constants. The Rocq development includes a formal cost model (`Complexity.v`, approximately 1,000 lines) that assigns unit cost to every cons cell, compatibility check, and list traversal. We prove closed-form inequalities with explicit constants—$100n^4 + 100$ for generic clearing, $2000n^2 + 2000$ for the single-forbidden fragment, $3n^2 + 7$ for SVI—by induction on the input structure, not merely asymptotic analysis. For the single-forbidden fragment the bound collapses from quartic to quadratic because clearing is the identity (Corollary 7.1), a fact we also prove formally.
Positioning. These contributions do *not* expand the class of polynomially solvable 3-SAT instances: well-formed sliding-window CNF has primal graph pathwidth ≤ 2, which is already solvable in linear time by classical dynamic programming. Our novelty lies in *reconstructing* this tractable fragment inside Romanov's triplet framework with full formal certification and executable extraction.

9.3. Relation to Bounded-Treewidth SAT. Well-formed sliding-window CNF has primal graph pathwidth ≤ 2 (each clause covers three consecutive vertices of a path, so the primal graph is a subgraph of the square of a path). SAT for graphs of bounded pathwidth is a classic FPT (Fixed-Parameter Tractable) result: dynamic programming on a path decomposition of width w solves it in time $2^{O(w)} \cdot n$ [15]. Our verified polynomial bounds for clearing and SVI therefore *do not expand the class of polynomially solvable 3-SAT instances*; they reconstruct the same pathwidth-≤ 2 fragment within Romanov's framework.

The connection to the FPT literature is worth making explicit. In a standard path decomposition each bag contains the variables active at that position, and the DP table stores all satisfying assignments of the bag (size 2^{w+1}). In Romanov's construction each tier corresponds to a bag of size 3, but instead of enumerating $2^3 = 8$ assignments directly, the tier stores the *forbidden* triplets—those ruled out by the clauses—and the DP step is replaced by a local compatibility check (compatible(t_1, t_2)) between adjacent bags. Thus a CTS is essentially a *nice path decomposition* in which

- **Introduce/forget** steps are implicit (the path is uniform, each bag contains exactly three consecutive variables);
- **Join** steps are absent (the graph is a path, not a tree);
- the DP table is replaced by an explicit triplet set, and the transition function is replaced by tier-wise intersection and adjacency checking.

This equivalence explains why the single-forbidden fragment (where each tier contains exactly one forbidden triplet) admits a quadratic bound: the DP table has constant size 8, and the transition is a simple table lookup. Our contribution is not a new complexity improvement but a *formally certified reformulation*: we prove in Rocq that Romanov's triplet constructions yield a correct decision procedure for the pathwidth-≤ 2 fragment, with explicit polynomial bounds and executable extraction. The practical interest of VFR lies not in solving sliding-window formulas (which is already tractable classically), but in using SVI as a

fast, auditable filter for harder structured instances that lie outside the verified fragment.

A novel structural decomposition. TLS decomposes a formula into independent CTFs based on variable overlap. This natural partitioning supports parallel solving and reveals structural properties of the instance (e.g., tightly coupled vs. loosely coupled variable groups).

A geometric interpretation of SAT.. Unlike the abstract implication graphs of CDCL, TLS provides a concrete picture: triplets as nodes, compatibility as edges, and satisfying assignments as paths through a layered graph (Figure 1). This makes TLS a valuable *pedagogical tool* for teaching SAT and constraint satisfaction.

9.4. Comparison with Classical SAT Solving. Unlike classical DPLL/CDCL, which operate directly on CNF clauses and rely on clause learning and implication graphs, VFR translates a formula into tiered triplet structures and reduces satisfiability to finding a compatible path. This yields a concrete, visualisable representation and a polynomial one-sided filter (SVI), but completeness requires an exponential post-check and the prototype does not implement clause learning. The main gain is a smaller, mechanically checked trust base for the verified sliding-window fragment.

9.5. Related Work. Our work complements a growing body of verified SAT solvers. Maric [9] verified a modern DPLL solver in Isabelle/HOL, proving correctness of unit propagation, conflict analysis, and clause learning. Blanchette et al. [10] extended this to a verified CDCL solver with proof generation, watched literals, and incremental solving. Subsequent work refined this into competitive verified solvers: Fleury et al. [11] formalised IsaSAT, an imperative CDCL solver with watched literals that approaches the performance of unverified solvers on some benchmarks. Complementing solver verification, Lammich [12] developed the GRAT toolchain, a formally verified certificate checker for DRAT proofs that outperforms the unverified reference implementation. Heule et al. [13] verified UNSAT proofs with extended resolution. These works establish a high standard for verified SAT solving; our work is the first to explore an alternative combinatorial foundation, using Romanov's triplet logic.

Structural differences. In DPLL/CDCL, the formula remains a flat set of clauses; the solver reasons via implication graphs, watched literals, and conflict-driven clause learning. A proof of correctness must therefore maintain global invariants over the trail, the learned-clause database, and the watched-literal indices. In contrast, TLS decomposes the formula geometrically into tiers of variable triplets, and satisfiability reduces to finding a compatible path through a layered graph. There is no implication graph, no watched literals, and no learned clauses—the only "conflict" is the emptiness of a tier intersection. Reasoning is local: each tier can be analysed as an independent set of triplets, and global correctness follows from pairwise compatibility. This locality makes TLS proofs compositional (tier-by-tier) rather than trace-based (execution-by-execution).

Insights from the alternative mechanisation. The TLS formalisation suggests three lessons that differ from the CDCL experience. First, *different data structures yield different proof localities*: whereas CDCL reasons about global implication graphs and watched literal indices whose correctness depends on intricate trail invariants, TLS decomposes the formula into tiers of triplets whose adjacency can be checked pairwise. The 424 proved statements in our Rocq development are overwhelmingly lemmas about tier-wise inclusion, compatibility, and intersection—concepts that have direct geometric meaning. Second, *a filter architecture offers a different trade-off*: the polynomial filter (SVI) can be verified in isolation, but completeness requires an exponential post-check. CDCL solvers are monolithic yet complete; VFR sacrifices completeness for a clean verified boundary (Table 1). Third, *concrete counterexamples are easy to construct*: because the reasoning is geometric, one can draw a three-tier structure and readily see why clearing is non-conservative (Section 4.5).

Why a verified one-sided filter matters. Verified CDCL solvers are complete decision procedures, but they are also heavy: thousands of lines of proof, complex imperative invariants, and aggressive performance engineering. Not every application needs a full solver. A verified one-sided filter answers a different but useful question: "Is this instance definitely outside the easy fragment?" with a proof-backed guarantee. For well-formed sliding-window CNF, the filter is not merely one-sided—it is exactly complete (Theorem 4.12), yet its proof is orders of magnitude smaller than a full CDCL proof. In program analysis or hardware verification, formulas often exhibit bounded pathwidth or sliding structure; a lightweight verified filter can discharge these instances quickly without invoking a heavy solver. The two approaches are complementary: a verified filter can be placed in front of *any* solver (even unverified) to obtain a sound architecture in which the polynomial stage is proof-carrying and the exponential stage is a standard fallback. Where CDCL verifies "the solver always returns the correct answer," VFR verifies "the polynomial filter never produces false negatives on the fragment." The fragment (pathwidth ≤ 2) is classically tractable; our contribution is a geometric reinterpretation with machine-checked polynomial bounds and an extractable implementation, not a new complexity result.

Triplet-based representations have appeared in various SAT contexts, although not as a complete solving paradigm. Romanov's TLS [3] is the first systematic framework built on tier-wise triplet structures and hyperstructure intersection. His insight—decomposing a formula into geometric layers and searching for compatible paths rather than raw assignments—is distinct from both DPLL search and CSP propagation. Our work is the first to subject TLS to mechanised formal verification, identifying the precise conditions under which its Simple Vertex Intersection is correct and where it requires supplementation.

The clearing procedure in TLS resembles arc consistency enforcement in constraint satisfaction problems (CSPs) [14]. The difference is that TLS operates on explicit triplet sets rather than general relations.

9.6. Threats to Validity.

The fundamental gap: grouped-window translation is a one-sided filter. The most serious threat is that our Rocq theorems are proved for a *clause-by-clause* translation (`FormulaTranslation.v`), whereas the VFR prototype uses

a *grouped-window* decomposition that merges multiple clauses sharing the same variable triple into a single tier. We have now proven (the group-equivalence and dense-groups lemmas in `FormulaTranslation.v`) the full forward equivalence: if an assignment satisfies the concatenated CNF, then the grouped CTF is satisfied. The converse, however, is **false**: satisfies_ctf checks each tier independently and does not enforce consistency between overlapping windows. A concrete counterexample: group 0 consists of the seven clauses on variables (x_0, x_1, x_2) whose only satisfying pattern is $(0, 0, 0)$, and group 1 consists of the seven clauses on (x_1, x_2, x_3) whose only satisfying pattern is $(1, 1, 1)$. The grouped CTF has two tiers, forbidding all triplets except $(0, 0, 0)$ and $(1, 1, 1)$ respectively; the sequence $ss = [0, 0, 0, 1, 1, 1]$ avoids both forbidden sets, so satisfies_ctf$(ss, \phi) =$ `true`. Yet the CNF is unsatisfiable: group 0 forces $x_1 = x_2 = 0$ while group 1 forces $x_1 = x_2 = 1$, so no assignment realises ss. This counterexample is formalised in `FormulaTranslation.v`. Consequently, the grouped-window heuristic is a *one-sided* (sound but incomplete) filter, not a complete decision procedure. The end-to-end guarantee holds only when the input is a well-formed sliding-window CNF translated clause-by-clause; for all other inputs the pipeline is heuristic.

Our empirical evaluation relies on instances derived from 3-CNF formulas, both randomly generated and from the SATLIB benchmark suite. Because the grouped-window translation is unverified, these benchmarks demonstrate operational behaviour of a heuristic, not formally verified correctness. A second threat concerns the scalability of VFR's post-check: our timeout-based evaluation for $n \geq 100$ does not establish an upper bound on the fraction of instances solvable within a practical time limit. Our Rocq formalisation proves termination and a fixed-point characterisation of clearing (Theorem 4.9 and Theorem 4.10), but we do not prove that clearing preserves all satisfying sets—in fact, we prove the opposite: Theorem 4.7 shows that the weak predicate satisfies_ctf can admit locally consistent assignments that do not correspond to any globally compatible path.

TLA$^+$ model-checking limitation. The TLA$^+$ (Temporal Logic of Actions) specifications provide finite-state sanity checks on small instances (MaxVars ≤ 3). The module `TLSSpec` exhausts TLC (the TLA$^+$ model checker) memory for MaxVars > 2 because explicit-state model checking of a 3-SAT solver is inherently exponential. This is expected: TLC serves as an auxiliary debugging tool for small illustrative instances, while the general correctness guarantees for arbitrary n are provided entirely by the Rocq proofs (424 lemmas and theorems). No fix is planned: switching to Apalache (a symbolic model checker for TLA$^+$) would extend the feasible bound only marginally and would not add scientific value, since the general theorem is already machine-checked in Rocq for arbitrary n. Users should treat TLC checks as illustrative rather than as formal verification.

9.7. Limitations and Future Work. The main practical limitation is the exponential post-check: when SVI returns `True`, the solver must enumerate partial assignments from each CTS. A second limitation is the absence of a verified CNF-to-CTF translation for grouped-window formulas; even though SVI and clearing are correct, the input may not faithfully represent the original formula. This is the single most important gap in turning the verified core into

an end-to-end verified solver. A third limitation is that verified permutation search is exhaustive and hence practical only for $n \leq 8$; a verified polynomial-time recognition algorithm for the sliding-window class (membership is decidable in polynomial time; cf. Section 3.1) would widen the verified fragment. A fourth, fundamental limitation is that arbitrary 3-CNF formulas typically admit no sliding-window ordering at all, so formal guarantees for general 3-SAT require external verified solvers or certificates.

An important research direction is to use the polynomial clearing procedure on single-forbidden structures as an alternative proof format: showing that CTS derivations map to short DRAT or RUP proofs would let the verified kernel *produce* certificates rather than merely *check* them. Engineering refinements such as lazy path generation, CDCL integration, and a native LRAT checker remain natural but secondary next steps for future work.

§10. Conclusion. We presented the first formal verification of Romanov's Triplet Logic in Rocq, complemented by empirical validation. Our central findings are:

- SVI is a *correct one-sided filter*: a joint satisfying set implies non-emptiness, but the converse fails when common triplets appear in non-aligned tiers (Section 6).
- Aligned intersection yields a *true bi-implication* (Theorem 4.3), extended systemically to k structures (Theorem 4.4).
- The weak predicate satisfies_ctf is incomplete with respect to global path existence; the strong predicate closes this gap for well-formed sliding-window CNF, where the clause-by-clause CNF-to-CTF translation preserves satisfiability (Section 4).
- Clearing and SVI have verified polynomial bounds (Table 2), and the extracted VFR prototype combines the verified filter with an exponential post-check.

To explore the structural potential of TLS, we proposed VFR: a hybrid architecture combining SVI's filter with an exponential post-check. We proved the filter's correctness formally and implemented the pipeline in Python, validating it against SAT and UNSAT instances, including cases where SVI gives false positives. We observed 100% agreement with a brute-force oracle on random sliding-window instances with $n \leq 6$. The entire toolchain is packaged in a reproducible Docker image (Section 7.4) that builds all components from source and passes the full automated test suite.

Our work positions TLS as a rigorous combinatorial framework for reasoning about compatible paths in tiered triplet structures, with a provably correct polynomial-time filter as a *theoretical building block*.

Formal Verification. All definitions, lemmas, and theorems described in this paper have been formally verified in Rocq 9.1.1. The development comprises more than 23,000 lines of Rocq code across

seventeen files with 424 proved lemmas and theorems and zero admitted goals. The source code is available in the curated Zenodo artifact [19].

Acknowledgments. This work is an output of a research project implemented as part of the Basic Research Program at the National Research University Higher School of Economics (HSE University).

During the preparation of this work, the author used large language models (DeepSeek, GLM, and Kimi Code) for generating and debugging Python, OCaml, TLA$^+$, and Rocq (Coq) code, creating tests and scripts, and optimising algorithms; for language polishing and stylistic refinement of the manuscript; for LaTeX code debugging; and for literature search, review and summarisation. The author reviewed and edited all outputs as needed and takes full responsibility for the content of the published article.

The author thanks arXiv for hosting the preprint version of this work (arXiv:2608.18445) [18].

The author is deeply indebted to the late Vladimir Romanov, whose pioneering ideas on triplet structures inspired this formal investigation.

REFERENCES

[1] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. STOC*, pages 151–158. ACM, 1971.

[2] L. A. Levin. Universal enumeration problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[3] V. F. Romanov. Non-Orthodox Combinatorial Models Based on Discordant Structures. 2011 (arXiv:1011.3944 [v2], revised Jan 2011; orig. Nov 2010).

[4] The Rocq Development Team. The Rocq Prover, version 9.1.1. `https://coq.inria.fr/`, 2026.

[5] N. Eén and N. Sórensson. An extensible SAT-solver. In *Proc. SAT*, pages 502–518. Springer, 2003.

[6] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proc. IJCAI*, pages 399–404, 2009.

[7] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC*, pages 530–535. ACM, 2001.

[8] A. Biere, M. Heule, H. van Maaren, T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[9] F. Maric. Formalisation and implementation of modern SAT solvers. *J. Automated Reasoning*, 43(1):81–119, 2009.

[10] J. C. Blanchette, M. Fleury, C. Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. *J. Automated Reasoning*, 61(1–4):333–365, 2018.

[11] M. Fleury, J. C. Blanchette, P. Lammich. A verified SAT solver with watched literals using imperative HOL. In *Proc. CPP*, pages 158–171. ACM, 2018.

[12] P. Lammich. Efficient verified (UN)SAT certificate checking. *J. Automated Reasoning*, 64(3):513–532, 2020.

[13] M. J. H. Heule, W. A. Hunt Jr., N. Wetzler. Verifying refutations with extended resolution. In *Proc. CADE*, pages 345–359. Springer, 2013.

[14] C. Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, pages 29–82. Elsevier, 2006.

[15] M. Alekhnovich and A. A. Razborov. Satisfiability, branch-width and Tseitin tautologies. In *Proc. FOCS*, pages 593–603. IEEE, 2002.

[16] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. System Sci.*, 13(3):335–379, 1976.

[17] H. Bryant, A. Lawrence, M. Seisenberger, and A. Setzer. Verification of Z3 RUP proofs in Coq-Rocq and Agda. `https://github.com/HarryBryant99/Verification-of-Z3-RUP-Proofs-in-Coq-Rocq-and-Agda`**, 2025.**

[18] D. V. Alexandrov. Formal verification of Romanov's triplet logic: A verified filter for sliding-window 3-CNF with application to structured formulas. *arXiv preprint arXiv:2608.18445*, 2026.

[19] D. V. Alexandrov. Formal verification of Romanov's triplet logic: artifacts and reproducible build. `https://doi.org/10.5281/zenodo.20397950`**, 2026.**

HSE UNIVERSITY
MOSCOW, RUSSIA
***E-mail*:** dvalexandrov@hse.ru