CLAUSE-COMPOSITIONAL INFERENCE AND ALIGNED TRAINING FOR TARGET-HIDDEN LOOP VERIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Loop invariants reduce unbounded executions to finite proof obligations. For legacy programs with unknown verification goals, target-specific verifier feedback is unavailable, making invariant generation especially challenging. Meanwhile, LLM-generated invariants often fail as complete responses even when they contain useful clauses, and complementary clauses may be scattered across different responses. We introduce CRAFT, a framework that aligns invariant generation and training with clause composition under hidden verification targets. At inference, CRAFT decomposes responses into clauses, pools and jointly filters clauses across responses, and composes the survivors into an invariant. This recovers useful proof material without requiring any response to prove the hidden target independently. The same filtered compositions supply SFT labels and determine which clauses receive RL credit. Because target assertions and postconditions remain hidden throughout generation and training, CRAFT estimates invariant strength by measuring how many candidate negative traces constructed from executions an invariant rejects. Supervised fine-tuning (SFT) distills accepted compositions from multiple responses into labels for one response, while reinforcement learning (RL) credits coverage contributed by retained clauses even when their original responses fail as wholes. On 832 C programs, SFT raises Qwen3-8B's compose@1 (the verification rate after filtering and composing one response) from 37.86% to 63.94%, exceeding the untrained model's 57.81% compose@32. Subsequent RL raises compose@1 to 69.23%. Reward ablations show that whole-response credit can improve standalone success while degrading compositional verification, whereas clause-level credit preserves useful contributions from failed responses.

1 INTRODUCTION

Loop invariants summarize loop behavior and reduce unbounded executions to finite proof obligations. However, target-driven invariant generation assumes a given postcondition or assertion, which may be absent from legacy code. Even when a final postcondition exists, weakest-precondition reasoning through subsequent loops and function calls typically relies on intermediate invariants and contracts that are difficult to discover automatically. Moreover, invariants tailored to a particular downstream goal may be less reusable for subsequent analyses with different objectives. We therefore study *target-hidden loop verification*, generating loop properties from the loop context without exposing a particular verification target.

Existing learning-based approaches, including Code2Inv, CLN2Inv, and LIPuS (Si et al., 2020; Ryan et al., 2020; Yu et al., 2023), and LLM-based methods such as LaM4Inv, Clause2Inv, and LORIS (Wu et al., 2024a; Cao et al., 2025b; Li et al., 2026a), often use a given verification target to guide invariant generation and refinement. With a visible target, whether a candidate invariant helps prove the desired property provides direct feedback for generation and refinement; when the target is hidden, this feedback is unavailable. Inductiveness alone cannot distinguish useful candidates, since `true` is an invariant of every loop. A useful invariant should cover reachable loop states while excluding as many unreachable states as possible. Because the exact reachable set is generally unknown and difficult to express precisely as a logical formula, we need a computable, target-independent proxy for invariant strength.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

An LLM response may contain both useful and incorrect clauses. Accepting or rejecting it as a whole can discard its useful parts. Responses must therefore be decomposed into clauses so their useful parts can inform both inference and training. Useful clauses may also be scattered across responses: conjoining valid invariants can yield a stronger invariant, while candidate clauses that fail preservation checks in isolation may become jointly inductive with supporting constraints. Under target hiding, where strength feedback is scarce, pooling and composing clauses across responses helps build stronger invariants.

We introduce CRAFT to align inference and training with the same compositional verification procedure. At inference, it decomposes responses into clauses, pools them for joint verification, repeatedly removes unproved clauses, and conjoins the survivors. During training, we construct candidate negative traces by perturbing observed states and continuing execution beyond observed loop exits. The rate at which verified invariants reject these traces provides a computable, target-independent proxy for strength, rather than an exact measure of reachable-set precision. SFT distills verified multi-response compositions that meet a coverage threshold into single-response labels. RL further aims to strengthen the invariants obtained from a single response under this inference procedure. It applies the same pooled filtering to groups of current responses and computes response-level advantages from the negative-trace coverage of each response's retained clauses and their complementary contributions across the group. Useful clauses thus provide training credit even when the complete response fails. All supervision comes from open-weight model generations, program execution, and verification; no closed model, human labeler, or external reward model is used.

On 832 C programs, compositional inference substantially improves base-model verification, and aligned training yields further gains. Our primary metric, compose@1, measures verification after filtering and composing one response. We use pass@1 as a standalone-response baseline. After SFT and RL, Qwen3-8B reaches 69.23% compose@1, exceeding its untrained compose@32 (57.81%) with a single response. At the same one-response budget, it also exceeds GPT-5 (51.32%) and Claude Sonnet-4.6 (56.97%) in compose@1 under our evaluation protocol. Across sampling budgets, compositional rewards also help mitigate sharpening, preserving verification gains at larger budgets.

Contributions.

- We introduce CRAFT, aligning target-hidden compositional inference with supervision and reward. A shared pooled filtering procedure recovers useful clauses from imperfect responses, constructs SFT labels, and identifies the contributions rewarded by RL. With one response, base Qwen3-8B reaches 37.86% compose@1 versus 6.37% pass@1.
- We turn multi-response discoveries into single-response learning signals. SFT learns accepted compositions, raising Qwen3-8B compose@1 to 63.94%; RL rewards retained clauses' coverage and group contributions, including those from failed responses, further raising it to 69.23%.
- Ablations show why aligning training with compositional inference matters: SFT on multi-response compositions provides more effective supervision than SFT on independently filtered responses, while rewarding retained clause contributions improves compositional verification more than whole-response rewards. Whole-response success alone does not capture the value of generated clauses for composition.

2 METHOD

CRAFT aligns training with compositional inference: verified multi-response compositions become SFT labels, and retained clauses determine each response's RL credit, including useful parts of failed responses (Figure 1). Appendix A gives the complete formal definition.

Definition 1 (Target-hidden loop-verification task)**.** Let $\mathcal{L} = (\mathsf{Init}, B, S)$ describe loop-entry states, the guard, and one loop step. A task $\mathcal{T} = (\mathcal{L}, \mathcal{G})$ additionally specifies verification targets $\mathcal{G}$ at their program locations. For a clause set C, the composed invariant is $I(C) = \bigwedge_{c \in C} c$, with $I(\varnothing) = \mathsf{true}$. Its establishment and preservation conditions are

$$\mathrm{Est}_{\mathcal{L}}(I(C)) \iff \forall \sigma : \mathsf{Init}(\sigma) \Rightarrow I(C)(\sigma),$$
$$\mathrm{Pres}_{\mathcal{L}}(I(C)) \iff \forall \sigma, \sigma' : I(C)(\sigma) \wedge B(\sigma) \wedge S(\sigma, \sigma') \Rightarrow I(C)(\sigma').$$

The invariant is valid exactly when both hold: $\mathrm{ValidInv}_{\mathcal{L}}(I(C)) \iff \mathrm{Est}_{\mathcal{L}}(I(C)) \wedge \mathrm{Pres}_{\mathcal{L}}(I(C))$. In the *target-hidden* setting, generation and intermediate scores see $\mathcal{L}$ but not $\mathcal{G}$; targets are used only

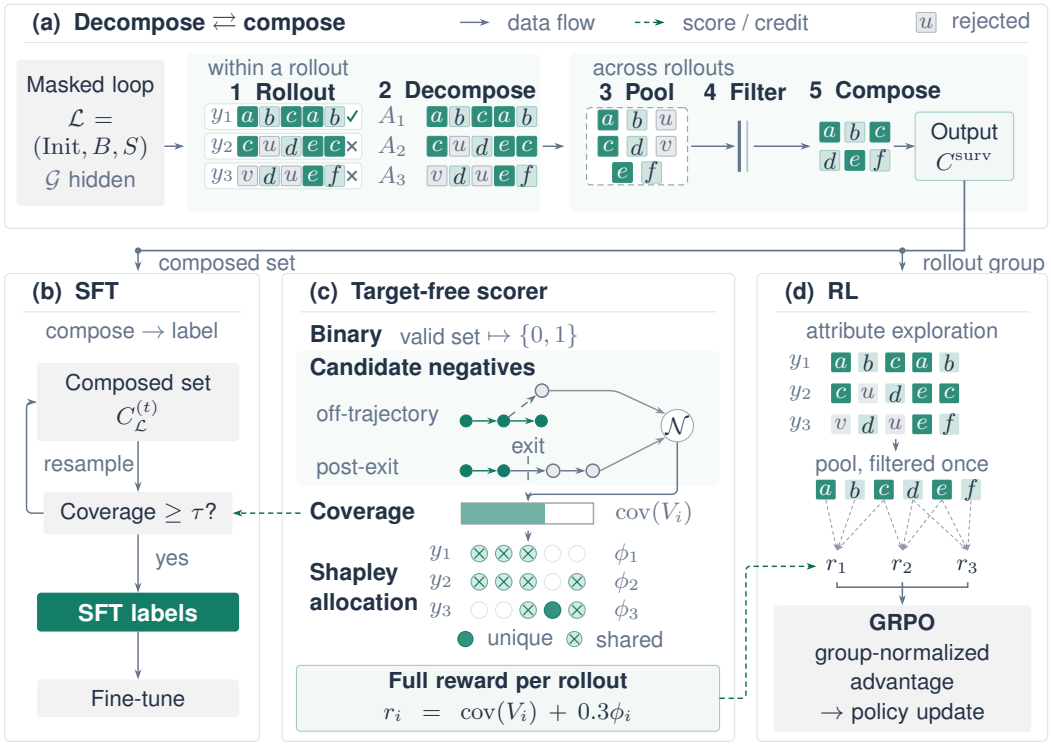

Figure 1: **Decompose–compose is the organizing principle.** (a) Pooling and joint filtering retain useful clauses from failed responses; letters track clause identity. (b) SFT distills coverage-accepted compositions. (c) Execution-derived candidate negatives supply coverage; Shapley allocation splits shared rejection credit. (d) Retained clauses credit their proposing rollouts for GRPO. The sampler inset shows off-trajectory perturbations and post-exit body continuations. The hidden target $\mathcal{G}$ is restored only for final verification.

for final verification. Appendix A.1 gives the full proof obligations, and Appendix B illustrates the numerical-loop setting through case studies.

We use the notation of Definition 1 throughout the inference, SFT, and RL procedures below. Clause pools and filter outputs are ordered sequences in the algorithms; set notation denotes their members, and conjunction ignores order. We abbreviate the bounded filter as $\mathrm{Filter}_{\mathcal{L}}$ and the final verifier as TaskVerify, suppressing the fixed budgets $\delta, h_{\max}$ defined in Appendix A.2.

2.1 Target-Hidden Composable Inference

Clauses from different responses can provide complementary target constraints and support one another during inductive verification. We therefore pool them before filtering. Let $\mathrm{Admit}(y)$ extract the first $m_{\max}$ invariant clauses from response y and remove duplicates, with $m_{\max}$ specified in Table 4. Algorithm 1 gives the inference procedure; $\|$ denotes concatenation and Unique removes duplicates.

Algorithm 1 Rollout–decompose–pool–filter–compose

Input: task $\mathcal{T}$ from Definition 1, policy π, number of responses n
Output: filtered clause set C, composed invariant $I(C)$, final verdict v

$x \leftarrow \mathcal{L}$ by masking $\mathcal{G}$ in $\mathcal{T}$ ▷ *hide target*
$y_{1:n} \sim \pi(\cdot \mid x)$ ▷ *rollout*
$A_i \leftarrow \mathrm{Admit}(y_i)$ for $i = 1, \ldots, n$ ▷ *decompose and cap*
$P \leftarrow \mathrm{Unique}(A_1 \| \cdots \| A_n)$ ▷ *pool in response order*
$C \leftarrow \mathrm{Filter}_{\mathcal{L}}(P)$ ▷ *filter*
form $I(C)$ as in Definition 1 ▷ *compose*
$v \leftarrow \mathrm{TaskVerify}(\mathcal{T}, C)$ ▷ *restore target*
return $(C, I(C), v)$

Joint verification and deletion. Houdini checks each clause for establishment at loop entry and preservation under the current conjunction. It repeatedly deletes unproved clauses and rechecks the survivors, since they may have depended on removed clauses. Only a proved fixed point is returned, ensuring an inductive conjunction under a sound verifier. Budget and failure handling are specified in Appendix A.2.

Why pooling precedes filtering. Let $\mathsf{H}_{\mathcal{L}}$ denote ideal Houdini with exact logical decisions. Filtering the union retains every clause obtainable by separately filtering the responses:

$$\bigcup_i \mathsf{H}_{\mathcal{L}}(A_i) \subseteq \mathsf{H}_{\mathcal{L}}\left(\bigcup_i A_i\right). \tag{1}$$

The inclusion can be strict when supporting clauses occur in different responses; its derivation is in Appendix A.10. Finite prover budgets can break this inclusion, so measured compose@k need not be monotonic (Appendix K.6).

2.2 Target-Independent Strength Signal

Inductive validity alone cannot distinguish useful invariants from weak ones such as true. We execute the masked loop and construct candidate negative traces $\mathcal{N}$: off-trajectory perturbations probe constraints among variables, continuations beyond witnessed exits probe boundary constraints, and local perturbations fill unused capacity. Sampling and cleaning are detailed in Appendix C.2.

For a verified clause set C, the state evaluator returns $\mathrm{Eval}(c, s) \in \{\mathsf{true}, \mathsf{false}, \mathsf{unknown}\}$. A negative trace is rejected only by a definite false evaluation:

$$\mathrm{rej}_{\mathcal{N}}(C) = \{\nu \in \mathcal{N} : \exists s \in \nu, \exists c \in C, \mathrm{Eval}(c, s) = \mathsf{false}\}. \tag{2}$$

An unknown evaluation contributes no rejection. Perturbations form singleton traces; each cleaned post-exit continuation remains one trace. The trajectory-level coverage is

$$\mathrm{cov}_{\mathcal{N}}(C) = \frac{|\,\mathrm{rej}_{\mathcal{N}}(C)|}{|\mathcal{N}|}, \qquad |\mathcal{N}| > 0. \tag{3}$$

Each trace contributes one unit regardless of its length.

Empirical alignment. The verifier supplies soundness, while coverage provides an empirical strength signal, achieving a within-program macro AUROC of 0.842 for final verification (Appendix K.5).

2.3 Distilling Multi-Response Search

SFT distills accepted multi-response compositions into single-response supervision, aiming to make a richer set of useful clauses available with fewer samples. Starting with $P_0 = \langle\rangle$, each round samples g_{sft} responses from a fixed generator policy π_0 (instantiated in Section 3.1), adds their clauses to the accumulated pool, and jointly filters it:

$$P_t = \mathrm{Unique}\Big(P_{t-1} \| \,\mathrm{Admit}(y_1^{(t)}) \| \cdots \| \,\mathrm{Admit}(y_{g_{\mathrm{sft}}}^{(t)})\Big), \qquad C_t = \mathrm{Filter}_{\mathcal{L}}(P_t). \tag{4}$$

Accumulating all admitted clauses in P_t allows a removed clause to return when a later response supplies support. The label can therefore combine discoveries across responses and rounds.

Acceptance and serialization. For nonempty $\mathcal{N}$, the first composition satisfying $\mathrm{cov}_{\mathcal{N}}(C_t) \geq \tau$ within t_{sft} rounds is serialized as the SFT label; otherwise no training pair is produced. Standard autoregressive cross-entropy trains the model on this label, conditioned on the masked loop. Algorithm 3 gives the complete procedure.

2.4 Rewarding Decomposed Contributions

A failed response should not erase the contributions of its useful clauses. RL therefore assigns credit to the parts retained by joint filtering. For a group sampled from the current policy, let $A_i = \mathrm{Admit}(y_i)$. Executions yield reachable loop-head states $\mathcal{E}$. The order-preserving screen $\mathrm{Screen}_{\mathcal{E}}$ removes malformed or out-of-scope clauses and clauses that evaluate to false on any $s \in \mathcal{E}$; unknown evaluations cause no removal. We screen the pooled clauses, filter once, and map survivors back to their proposing responses:

$$\begin{aligned} P^* &= \mathrm{Unique}(A_1 \| \cdots \| A_{g_{\mathrm{rl}}}), \\ P_{\mathcal{E}} &= \mathrm{Screen}_{\mathcal{E}}(P^*), & C^* &= \mathrm{Filter}_{\mathcal{L}}(P_{\mathcal{E}}), \\ V_i &= \langle c \in A_i \mid \mathrm{key}(c) \in \mathrm{keys}(C^*) \rangle, & R_i &= \mathrm{rej}_{\mathcal{N}}(V_i). \end{aligned} \tag{5}$$

Canonical keys identify which clauses each response proposed; V_i preserves their order in A_i. Thus V_i retains its surviving contributions even when other clauses fail, with mutually supporting clauses checked in the pooled context used at inference.

For nonempty $\mathcal{N}$, Full, our default reward, adds Shapley credit to Clause-decomposed coverage:

$$r_i = \frac{|R_i|}{|\mathcal{N}|} + 0.3\phi_i. \tag{6}$$

With the survivor bundles fixed, let $f(\nu) = |\{j : \nu \in R_j\}|$ count the responses rejecting trace ν. The Shapley credit is

$$\phi_i = \frac{1}{|\mathcal{N}|} \sum_{\nu \in R_i} \frac{1}{f(\nu)}. \tag{7}$$

Each rejected trace's credit is divided among the responses covering it. At equal coverage, Full rewards responses more when their rejected traces are covered by fewer other responses, encouraging complementary effective coverage within the group. The coverage game and proof are in Appendix A.6.1; Appendix K.9 analyzes the empirical effect.

GRPO standardizes rewards within each group and applies each response's scalar advantage to its tokens; clauses determine credit, while policy updates remain response-level. A clause that only supports another clause receives no direct coverage credit. Update details and the empty-negative fallback are in Appendix A.7 and Algorithm 4.

3 Experiments

We organize the evaluation around four questions: how much do compositional inference and aligned training improve verification (RQ1); can CRAFT improve verification while reducing token use relative to specialized tools (RQ2); do SFT and RL contribute differently (RQ3); and which design choices enable these gains (RQ4)? Appendix J collects the complete raw results; the independent analyses are in Appendix K.

3.1 Experimental Setup

Data and judge. Training data comprise 9,024 RL records and 33,346 SFT records (31,865 distinct sources) of target-masked scalar-integer single-loop programs. The raw training-program pool is drawn primarily from SV-COMP and SyGuS (Beyer, 2023; Alur et al., 2013); Appendix G describes program selection and verified SFT-label construction. The SFT corpus is synthesized once with Qwen3-8B as the generator in the procedure of Section 2.3 and is shared unchanged by all trained

Backbone	Training	$k = 1$		$k = 8$	
		compose	pass	compose	pass
Qwen3-4B	Base	32.93	5.29	47.60	12.50
	SFT	62.62	40.38	76.20	59.98
	RL	48.80	1.68	62.50	4.45
	SFT+RL (CRAFT)	67.55	35.10	78.00	53.25
Qwen3-8B	Base	37.86	6.37	54.21	17.31
	SFT	63.94	41.95	76.20	61.18
	RL	51.80	2.88	64.18	7.81
	SFT+RL (CRAFT)	69.23	36.78	**78.73**	56.49
Qwen3-14B	Base	43.51	7.69	61.54	17.91
	SFT	66.35	**45.79**	78.37	**63.58**
	RL	65.38	10.34	72.24	18.63
	SFT+RL (CRAFT)	**69.35**	35.82	77.04	53.00
API models					
GPT-5-nano	–	33.89	3.61	55.53	16.11
GPT-5	–	51.32	32.57	72.00	51.68
Claude Sonnet-4.6	–	56.97	36.30	66.95	44.95
DeepSeek V4-Flash	–	51.44	28.13	74.40	48.20

Table 1: Verification across models and training stages (%). Bold marks each column's maximum.

backbones; RL rewards are computed online from each trained model's own rollouts. Program execution and Frama-C/WP verification supply every acceptance and scoring decision; no closed model, human labeler, or external reward model is involved at any stage. The evaluation combines 832 programs from Code2Inv, SyGuS, SV-COMP, LIPuS, Clause2Inv, and Loopy: 316 linear, 50 nonlinear-arithmetic, and 466 Loopy instances (Si et al., 2020; Alur et al., 2013; Beyer, 2023; Yu et al., 2023; Cao et al., 2025b; Kamath et al., 2024). No training program matches an evaluation program under exact, alpha-renamed, or constant-abstracted matching (Appendix F). The judge is Frama-C/WP (Kirchner et al., 2015). Success requires it to prove induction and every restored target; all other outcomes count as zero.

Inference protocol. All CRAFT evaluation runs share target masking, the canonical prompt, clause processing, Houdini, stochastic decoding, and no re-roll or target feedback; external tools retain their own target-hidden orchestration. Each probe curve reuses one saved pool of n_{probe} responses per task. Appendix H gives decoding, serving, prover, and training details.

Metrics. Throughout evaluation, success means $\mathcal{G}$-sufficiency under the final verifier. Our primary metric is compose@k: success under CRAFT's filtering-and-composition pipeline on the first k saved responses. We report pass@k as a standalone baseline: success when at least one of the same responses independently verifies the targets. Appendix H gives the definitions.

3.2 RQ1: How Much Do Compositional Inference and Aligned Training Improve Verification?

Table 1 compares three local backbones across training stages, with four API models as references. It reports the primary compose metric and the standalone pass baseline at one- and eight-response budgets. RL denotes training from Base; SFT+RL starts from SFT. API models receive no additional training.

Compositional inference alone substantially improves verification. For Base Qwen3-8B, compose@1 reaches 37.86%, versus pass@1 at 6.37%, or $5.94\times$ the standalone rate. Both rates concern a one-response inference budget, showing that the framework can recover clauses sufficient for a proof from an answer that fails as a whole.

Two-stage training aligned with inference further improves verification. SFT constructs supervision from composed valid clause sets, while RL assigns credit based on the contributions of clauses retained after filtering. Qwen3-8B compose@1 rises from 37.86% to 63.94% after SFT and then

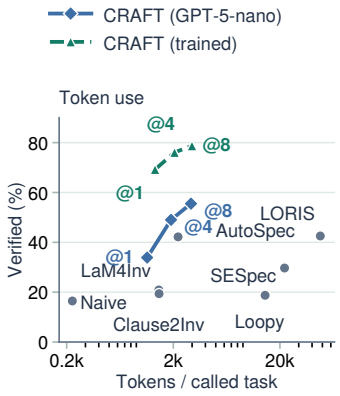

Figure 2: Verification versus token use. Dashed: trained CRAFT reference.

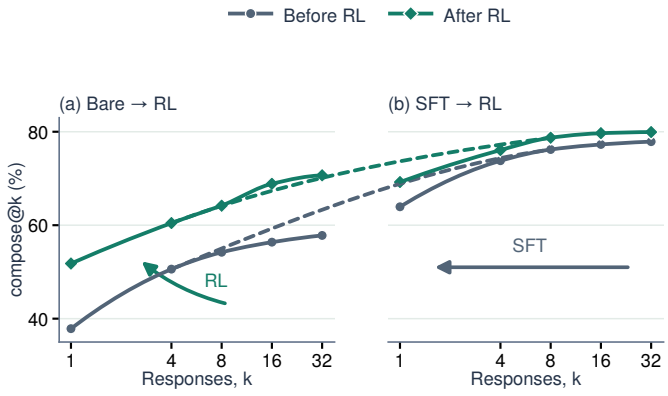

Figure 3: Qwen3-8B verification across sampling budgets, with RL initialized from (a) Base and (b) SFT. SFT reduces the sampling budget needed for verification; RL exhibits a sharpening effect, with larger gains at smaller k.

to 69.23% after RL. All three local backbones show this progressive improvement in compose@1; Qwen3-4B SFT+RL reaches 67.55%, exceeding Sonnet-4.6's 56.97% under the evaluated protocol.

Finding. Compositional inference improves verification both before and after training. Aligning SFT labels and RL rewards with this inference process further improves verification, while trained models continue to benefit from inference-time composition.

3.3 RQ2: Can CRAFT Verify More While Spending Fewer Tokens?

We compare CRAFT with AutoSpec, SESpec, Clause2Inv, LORIS, LaM4Inv, Loopy, and Daikon (Wen et al., 2024; Yang et al., 2025; Cao et al., 2025b; Li et al., 2026a; Wu et al., 2024a; Kamath et al., 2024; Ernst et al., 2007). LLM-based comparisons hold GPT-5-nano, disabled reasoning, target hiding, and the final judge fixed. Comparison protocols, supported subsets, and matched-subset results are reported in Appendix K.8.2.

Clause reuse improves the verification–cost frontier. With GPT-5-nano fixed, all three CRAFT budgets lie on the verification–token Pareto frontier (Figure 2). On all 832 tasks, four responses verify **49.04%** using **1,905 tokens per task**, exceeding all compared tools in verification rate while costing less than four of the six LLM-based baselines. Eight responses raise verification to 55.53% at 2,929 tokens. The trained model further improves the verification–token trade-off.

Finding. For invariant generation with hidden verification targets and limited informative feedback, reusable clauses improve the verification–token trade-off. Compared with iterative repair pipelines, parallel sampling and composition verify more programs with fewer tokens.

3.4 RQ3: How Do SFT and RL Reshape Compositional Search?

Prior work suggests that RLVR can increase the sampling frequency of existing behaviors (Yue et al., 2025). Figure 3 compares Qwen3-8B from Base and SFT initializations to examine whether this pattern also appears in compositional verification.

SFT expands coverage and compresses the sampling budget. SFT reaches 63.94% compose@1, exceeding Base compose@32 (57.81%). After RL, SFT initialization raises compose@1 from 51.80% to 69.23% under the same one-response budget. It also raises compose@32 from 70.67% to 79.93%: SFT both makes useful candidates accessible with fewer responses and expands verification coverage within the measured range.

After SFT, RL mainly makes useful candidates available earlier. From Base, RL improves compose@1 and compose@32 by 13.94 and 12.86 points, respectively, retaining a substantial gain even at the largest budget. After three SFT epochs, its gains shrink to **5.29 points at compose@1 and 2.05 at compose@32**. After SFT, RL mainly improves sampling efficiency: useful candidates

appear with fewer responses, while coverage at the largest tested budget improves more modestly. Appendix K.10 further examines this sharpening pattern and shows that finer credit assignment through compositional rewards helps mitigate it.

RL reveals a tension between pass and compose. Table 1 shows that RL's compositional gains can coincide with lower standalone success. For Qwen3-8B, pass@1/8 falls from 6.37%/17.31% to 2.88%/7.81% when RL starts from Base, and from 41.95%/61.18% to 36.78%/56.49% when RL starts from SFT. The appendix clause-cap ablation shows the same pattern: admitting more clauses improves compose@1/8 while lowering pass@1 (Appendix K.4).

Finding. SFT broadens verification coverage and compresses compositional search into fewer responses. Post-SFT RL mainly improves sampling efficiency, making useful candidates available earlier even after substantial supervised training.

3.5 RQ4: Why Do Composed SFT Labels and Decomposed Rewards Help?

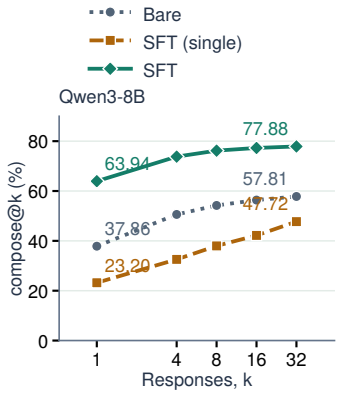

Figure 4: Effect of SFT label construction on Qwen3-8B compose@k.

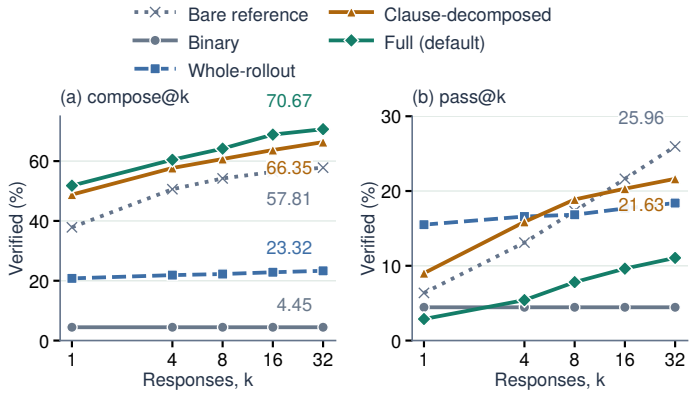

Figure 5: Qwen3-8B reward ablation from Base: (a) compose@k; (b) pass@k. Vertical scales differ.

Having examined how SFT and RL reshape compositional search, we now isolate the effects of SFT label construction and RL credit assignment through ablations.

Composition turns broader exploration into supervision. SFT (single) uses independently filtered responses; composed labels pool valid clauses across responses, giving supervision access to more candidate invariants and complementary constraints. With approximately equal numbers of training examples, composed-label SFT raises Qwen3-8B compose@1 from 23.20% to 63.94% and pass@1 from 19.95% to 41.95%, relative to SFT (single) (Figure 4). Composition exposes SFT to the results of multi-response exploration, which the trained model can exploit with a single response.

One rejected clause should not erase the others' credit. With verification targets hidden, RL uses inductiveness and negative-trace coverage as its reward signal (Figure 5). Binary reward reaches only 4.45% compose@32. Whole-rollout adds a coverage reward, but pays it only when every clause in a response survives filtering. Thus, one rejected clause can withhold credit from other useful surviving clauses, making the signal sparse. Consistent with this, Whole-rollout reaches only 23.32% compose@32, below Base's 57.81%. Clause-decomposed removes this all-or-nothing gate and rewards retained clause contributions, reaching 66.35% compose@32. Relative to Base, this gain comes with a decrease in pass@32 from 25.96% to 21.63%. This reflects our training alignment: RL rewards clause contributions that serve CRAFT's composition pipeline, rather than standalone pass.

Group credit adds most from Base. Full adds Shapley credit with weight 0.3. From Base, it raises compose@32 from 66.35% to 70.67%; after SFT, the selected Full model reaches 69.23% compose@1 versus Clause-decomposed's 67.91%, while compose@32 improves by 1.68 points. Thus, the compose@32 gain from Shapley credit is 4.32 points from Base but only 1.68 points

after SFT. This smaller marginal benefit may indicate that SFT already enables sufficiently broad exploration of useful clauses. The full comparison is in Appendix K.9.

Appendix K reports the complete ablations, including inference, clause cap, filtering order, and target visibility.

Finding. Composed labels turn useful clauses from broader exploration into supervision; clause-level rewards preserve useful contributions within failed responses. Together, they let training exploit more of the candidates that compositional verification can use.

4 Related Work

Invariant generation and verified filtering. Classical invariant synthesis uses abstract interpretation, algebraic relations, constraint solving, interpolation, recurrences, syntax-guided search, or counterexamples (Cousot & Cousot, 1977; Cousot & Halbwachs, 1978; Karr, 1976; Colón et al., 2003; Sankaranarayanan et al., 2004; Gulwani et al., 2008; Gupta & Rybalchenko, 2009; McMillan, 2003; Kincaid et al., 2017; Alur et al., 2013; Garg et al., 2014); dynamic approaches mine likely invariants from executions (Ernst et al., 2007). Verified filtering retains an inductive subset of proposed clauses (Flanagan & Leino, 2001). Neural and LLM systems learn construction policies or combine generation with solver feedback, static analysis, filtering, and repair (Si et al., 2018; 2020; Ryan et al., 2020; Yao et al., 2020; Yu et al., 2023; Kamath et al., 2024; Chakraborty et al., 2023; Liu et al., 2026; Wu et al., 2024b). Baldur instead generates and repairs whole proofs (First et al., 2023). LaM4Inv conjoins BMC-filtered predicates from candidate invariants (Wu et al., 2024a); Clause2Inv searches conjunctions and disjunctions of clauses (Cao et al., 2025b). Both iteratively refine candidates using verification feedback; LORIS repairs local reasoning errors (Li et al., 2026a). Quokka uses UAutomizer to evaluate invariant correctness and verification speedups (Wei et al., 2026). These workflows rely on visible targets or target-derived feedback unavailable under target hiding. CRAFT jointly filters clauses from independent responses without verifier feedback to the generator, and aligns SFT and RL with this composition.

Broader specification-generation frameworks also synthesize loop invariants by combining LLM generation with program analysis or symbolic verification (Wen et al., 2024; Yang et al., 2025; Ma et al., 2025).

Training from formal and behavioral feedback. Verified data and execution or prover feedback have been used to train invariant, code, and proof models (Wei et al., 2026; Pinto et al., 2026; Le et al., 2022; Yan et al., 2026). Wonda curates verifier-generated invariants through AST normalization, LLM-driven semantic rewriting, and augmentation to train small language models (Pinto et al., 2026). COINS and SpecRL further use positive and negative behaviors to distinguish weak valid specifications from stronger ones (Yang et al., 2026; Huang et al., 2026); SpecRL rewards verified candidates according to the fraction of negative tests their specifications reject. CRAFT instead uses target-hidden behavioral evidence to train on verified multi-response compositions and reward retained clauses from failed responses.

Credit assignment in RLVR. Related RLVR and RLHF methods study group-relative advantages and fine-grained credit (Shao et al., 2024; Li et al., 2026b; Cao et al., 2025a). Recent methods optimize pass@k or up-weight rare correct responses (Walder & Karkhanis, 2025; Chen et al., 2025; He et al., 2025), and per-response RL can improve pass@1 without expanding large-k coverage (Yue et al., 2025; He et al., 2025). CRAFT can combine clauses from multiple failed responses into a proof. Its clause-based coverage and Shapley credit produce response-level advantages for policy updates.

5 Conclusion and Limitations

CRAFT recovers proofs from clauses in responses that fail to prove targets alone. Decomposing, filtering, and composing those clauses raises Base Qwen3-8B verification from 6.37% to 37.86%. Composed SFT labels broaden coverage, while subsequent RL makes useful clauses available in fewer samples. Reward ablations show that whole-response credit can improve standalone pass but reduce compositional verification. CRAFT currently handles one loop over scalar integers and combines clauses by conjunction. We evaluate it with Frama-C/WP; nonlinear relations remain challenging for the evaluated models. Future work will extend CRAFT to richer program states and control flow and explore its transfer to specification and test-case generation.

REFERENCES

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, pp. 1–8, 2013.

Dirk Beyer. Competition on software verification (SV-COMP). `https://sv-comp.sosy-lab.org/`, 2023.

Meng Cao, Shuyuan Zhang, Xiao-Wen Chang, and Doina Precup. SCAR: Shapley credit assignment for more efficient RLHF. *arXiv preprint arXiv:2505.20417*, 2025a.

Weining Cao, Guangyuan Wu, Tangzhi Xu, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. Clause2Inv: A generate-combine-check framework for loop invariant inference. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1009–1030, 2025b. doi: 10.1145/3728920.

Saikat Chakraborty, Shuvendu K. Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. Ranking LLM-generated loop invariants for program verification. In *Findings of the Association for Computational Linguistics: EMNLP*, pp. 9164–9175, 2023. doi: 10.18653/v1/2023.findings-emnlp.614.

Zhipeng Chen, Xiaobo Qin, Youbin Wu, Yue Ling, Qinghao Ye, Wayne Xin Zhao, and Guang Shi. Pass@k training for adaptively balancing exploration and exploitation of large reasoning models. *arXiv preprint arXiv:2508.10751*, 2025.

Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. Int. Conf. on Computer Aided Verification (CAV)*, pp. 420–432. Springer, 2003.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL)*, pp. 238–252, 1977.

Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL)*, pp. 84–96, 1978.

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, 2007.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proc. 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE)*, pp. 1229–1241, 2023.

Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proc. Int. Symp. of Formal Methods Europe (FME)*, pp. 500–517. Springer, 2001.

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *Proc. Int. Conf. on Computer Aided Verification (CAV)*, pp. 69–87. Springer, 2014.

Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 281–292, 2008.

Ashutosh Gupta and Andrey Rybalchenko. InvGen: An efficient invariant generator. In *Proc. Int. Conf. on Computer Aided Verification (CAV)*, pp. 634–640. Springer, 2009.

Andre Wang He, Daniel Fried, and Sean Welleck. Rewarding the unlikely: Lifting GRPO beyond distribution sharpening. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 25548–25560, 2025. doi: 10.18653/v1/2025.emnlp-main.1298.

Zhechong Huang, Zhao Zhang, Zeyu Sun, Huifeng Sun, and Yingfei Xiong. SpecRL: Reinforcement learning with test-based completeness rewards for formal specification synthesis. *arXiv preprint arXiv:2604.05820*, 2026.

Adharsh Kamath, Nausheen Mohammed, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Leveraging LLMs for program verification. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, pp. 107–118, 2024. doi: 10.34727/2024/isbn.978-3-85448-065-5_16.

Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.

Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. Compositional recurrence analysis revisited. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 248–262, 2017.

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Tianchi Li, Zhenyu Yan, Junhao Liu, Peng Di, and Xin Zhang. Guiding LLM-based loop invariant synthesis via feedback on local reasoning errors. *ACM Transactions on Programming Languages and Systems*, 48(2):8:1–8:30, 2026a. doi: 10.1145/3806652.

Yugu Li, Zehong Cao, Jianglin Qiao, and Siyi Hu. SSVPO: Effective step-level credit assignment for RL training of language models. In *Proc. Int. Conf. on Learning Representations (ICLR)*, 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/4da4dd613cc9e63932d643b9c6e8f86f-Abstract-Conference.html.

Chang Liu, Xiwei Wu, Yuan Feng, Qinxiang Cao, and Junchi Yan. Towards general loop invariant generation: A benchmark of programs with memory manipulation. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Ruibang Liu, Minyu Chen, Ling-I Wu, Jingyu Ke, and Guoqiang Li. Enhancing automated loop invariant generation for complex programs with large language models. *Science of Computer Programming*, 248:103387, 2026. doi: 10.1016/j.scico.2025.103387.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. SpecGen: Automated generation of formal program specifications via large language models. In *Proc. Int. Conf. on Software Engineering (ICSE)*, 2025.

Kenneth L. McMillan. Interpolation and SAT-based model checking. In *Proc. Int. Conf. on Computer Aided Verification (CAV)*, pp. 1–13. Springer, 2003.

Ido Pinto, Yizhak Yisrael Elboher, Haoze Wu, Nina Narodytska, and Guy Katz. Not all invariants are equal: Curating training data to accelerate program verification with SLMs. In *Proc. Int. Conf. on Machine Learning (ICML)*, 2026. URL https://icml.cc/virtual/2026/poster/62571.

Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. CLN2INV: Learning loop invariants with continuous logic networks. In *Proc. Int. Conf. on Learning Representations (ICLR)*, 2020.

Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In *Proc. 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pp. 318–329, 2004.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pp. 7751–7762, 2018.

Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. Code2Inv: A deep learning framework for program verification. In *Proc. Int. Conf. on Computer Aided Verification (CAV), Part II*, pp. 151–164. Springer, 2020.

Christian Walder and Deep Tejas Karkhanis. Pass@k policy optimization: Solving harder reinforcement learning problems. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Anjiang Wei, Tianran Sun, Tarun Suresh, Haoze Wu, Ke Wang, and Alex Aiken. Quokka: Accelerating program verification with LLMs via invariant synthesis. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=YvyAcGrpcJ.

Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *Proc. Int. Conf. on Computer Aided Verification (CAV)*, pp. 302–328. Springer, 2024.

Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. LLM meets bounded model checking: Neuro-symbolic loop invariant inference. In *Proc. IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, pp. 406–417, 2024a. doi: 10.1145/3691620.3695014.

Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *Proc. Int. Conf. on Learning Representations (ICLR)*, 2024b.

Chuanhao Yan, Fengdi Che, Xuhan Huang, Xu Xu, Xin Li, Yizhi Li, Xingwei Qu, Jingzhe Shi, Chenghua Lin, Yaodong Yang, Binhang Yuan, Hang Zhao, Yu Qiao, Bowen Zhou, and Jie Fu. Re:Form—reducing human priors in scalable formal software verification with RL in LLMs: A preliminary study on Dafny. *Transactions on Machine Learning Research*, 2026.

Fanpeng Yang, Xu Ma, Shuling Wang, Xiong Xu, Qinxiang Cao, Naijun Zhan, Xiaofeng Li, and Bin Gu. Integrating symbolic execution with LLMs for automated generation of program specifications. *arXiv preprint arXiv:2506.09550*, 2025.

Fanpeng Yang, Xing Li, Shuling Wang, Jie An, Zeyu Sun, Shenghua Feng, Wenhan Wang, Weiyi Wang, Naijun Zhan, and Fanjiang Xu. How powerful are LLMs in generating formal program specifications? In *Proc. Int. Conf. on Machine Learning (ICML)*, 2026.

Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 106–120, 2020.

Shiwen Yu, Ting Wang, and Ji Wang. Loop invariant inference through SMT solving enhanced reinforcement learning. In *Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA)*, pp. 175–187, 2023. doi: 10.1145/3597926.3598047.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? In *Advances in Neural Information Processing Systems*, volume 38, 2025.

AI Use Statement

The authors used generative AI tools to polish the manuscript's language and organize experimental data for presentation in tables and figures. They checked the AI-assisted text and data presentations against the underlying records and take responsibility for the final manuscript and reported results.

Reproducibility Statement

The method and objective are specified in Section 2; Appendices C–H give the implementation defaults, prompt, training interface, data-overlap audit, and evaluation protocol. Appendix J organizes the available measurements.

A Complete Formalization of CRAFT

This section formalizes the task, invariant obligations, verifier, and the inference and training procedures introduced in Section 2.

A.1 Task Semantics, Loop Invariants, and Verification

We use the establishment, preservation, and validity conditions from Definition 1. Let Σ be the program-state space and $S \subseteq \Sigma \times \Sigma$ one normally completing loop step.

For each target $(\ell, g) \in \mathcal{G}$, let $R_\ell(\sigma, \sigma')$ mean that execution from loop-head state σ can reach target location ℓ in state σ'. This relation follows the program paths to the target, including the loop exit path for a post-loop target. The separate target obligation is

$$\mathrm{Term}_{\mathcal{L}}(I(C), \ell, g) \iff \forall \sigma, \sigma' : \ I(C)(\sigma) \wedge R_\ell(\sigma, \sigma') \Rightarrow g(\sigma'). \tag{8}$$

It expresses what follows if execution reaches the target; it is a partial-correctness condition and does not prove that the loop terminates. The task is valid when the invariant is valid and all target obligations hold:

$$\mathrm{TaskValid}_{\models}(\mathcal{T}, C) = \mathrm{ValidInv}_{\mathcal{L}}(I(C)) \wedge \bigwedge_{(\ell,g)\in\mathcal{G}} \mathrm{Term}_{\mathcal{L}}(I(C), \ell, g). \tag{9}$$

Generation and training observe only $\mathcal{L}$; the target set $\mathcal{G}$ is restored only for final verification.

For a proof goal g generated by the configured verifier, define its portfolio status by

$$\mathrm{GoalStatus}_\delta(g) \in \{\mathsf{valid}, \mathsf{timeout}, \mathsf{unknown}\}, \tag{10}$$

where valid means that at least one configured prover returned *Valid*, timeout means no prover proved the goal and at least one exhausted budget δ, and unknown covers other non-valid outcomes.

Let $\mathcal{S}_{\mathrm{ver}} = \{\mathsf{valid}, \mathsf{timeout}, \mathsf{unknown}\}$. For an ordered clause sequence $D = \langle d_1, \ldots, d_m \rangle$, one resource-bounded loop-verifier call returns

$$\mathrm{LoopVerify}_{\mathcal{L},\delta}(D) = \{d_k \mapsto v_{D,k} : 1 \le k \le m\} \in \mathcal{S}_{\mathrm{ver}}^{D}, \tag{11}$$

where $v_{D,k} = \mathsf{valid}$ means that establishment and preservation for clause d_k are proved in the current conjunction $I(D)$. The interface is sound if

$$\big(\forall c \in D : \ \mathrm{LoopVerify}_{\mathcal{L},\delta}(D)[c] = \mathsf{valid}\big) \Rightarrow \mathrm{ValidInv}_{\mathcal{L}}(I(D)). \tag{12}$$

Timeout and unknown are both rejected by filtering. The final task verifier checks syntax, the loop obligations, and the original target goals:

$$\begin{aligned} \mathrm{TaskVerify}_\delta(\mathcal{T}, C) = \mathbf{1}\big[& \mathrm{SyntaxOK}(\mathcal{T}, C) \wedge \\ & \forall c \in C : \ \mathrm{LoopVerify}_{\mathcal{L},\delta}(C)[c] = \mathsf{valid} \wedge \\ & \forall g \in \mathrm{TargetGoals}(\mathcal{T}, C) : \ \mathrm{GoalStatus}_\delta(g) = \mathsf{valid}\big]. \end{aligned} \tag{13}$$

A.2 Ideal and Resource-Bounded Houdini Filtering

Ideal Houdini starts from a clause set C, retaining clauses whose establishment and context-dependent preservation obligations hold:

$$\begin{aligned} \mathrm{Est}_{\mathcal{L}}(c) &\iff \forall \sigma : \ \mathsf{Init}(\sigma) \Rightarrow c(\sigma), \\ \mathrm{Pres}_{\mathcal{L},C}(c) &\iff \forall \sigma, \sigma' : \ I(C)(\sigma) \wedge B(\sigma) \wedge S(\sigma, \sigma') \Rightarrow c(\sigma'). \end{aligned} \tag{14}$$

It iterates

$$C_{j+1} = \mathsf{F}_{\mathcal{L}}(C_j) = \{c \in C_j : \mathrm{Est}_{\mathcal{L}}(c) \wedge \mathrm{Pres}_{\mathcal{L},C_j}(c)\} \tag{15}$$

until a fixed point $\mathsf{H}_{\mathcal{L}}(C)$ is reached.

The implemented deletion round uses the clause-level loop verifier and preserves the relative order of retained clauses:

$$\widehat{\mathsf{F}}_{\mathcal{L},\delta}(D) = \big\langle c \in D \ \big|\ \mathrm{LoopVerify}_{\mathcal{L},\delta}(D)[c] = \mathsf{valid} \big\rangle. \tag{16}$$

Thus timeout and unknown both cause conservative deletion. Every status in one round is computed under the same pre-deletion context D. A clause removed at the end of that round may therefore have helped another clause receive valid. The survivors must be verified again without the removed hypotheses; this is why a single deletion pass is insufficient. For an ordered input sequence D, let $D_0 = D$ and $D_{j+1} = \widehat{\mathsf{F}}_{\mathcal{L},\delta}(D_j)$. Iteration stops at the first proved fixed point or at the symbolic round limit $h_{\max}$. Let

$$j^* = \min\{j < h_{\max} : \widehat{\mathsf{F}}_{\mathcal{L},\delta}(D_j) = D_j\}, \tag{17}$$

when this set is nonempty. The implemented filter is

$$\mathrm{Filter}_{\mathcal{L}}^{\delta,h_{\max}}(D) = \begin{cases} D_{j^*}, & j^* \text{ exists}, \\ \langle\rangle, & \text{the round limit is reached first.} \end{cases} \tag{18}$$

At a returned fixed point, every surviving clause has just been proved in the context of that same sequence. Reaching the limit returns the empty sequence. Under a sound loop verifier, the result is sound and may be smaller than the ideal fixed point. In the remainder, $\mathrm{Filter}_{\mathcal{L}}$ abbreviates Equation (18), and TaskVerify abbreviates $\mathrm{TaskVerify}_{\delta}$.

A.3 The Shared Composition Operator

Let $\mathrm{parts}(y)$ extract the ordered sequence of invariant clauses in response y, let $\mathrm{First}_m(L)$ be the prefix of sequence L containing at most m clauses, and let $\|$ denote sequence concatenation. Define the clauses admitted from one response by

$$\mathrm{Admit}(y) = \mathrm{Unique}(\mathrm{First}_{m_{\max}}(\mathrm{parts}(y)))\,. \tag{19}$$

Unique removes duplicates using the finite-rule canonical normalization $\mathrm{key}(c)$. Equal keys identify the clauses merged by those rules. Provenance is retained by recording which responses supplied each key. It keeps the first occurrence; concatenations are traversed in response order and then clause order, which fixes the sequence supplied to LoopVerify. For a clause sequence D, let $\mathrm{keys}(D) = \langle \mathrm{key}(c) : c \in D \rangle$.

For any response sequence $y_{1:n}$, define

$$A_i = \mathrm{Admit}(y_i), \qquad P(y_{1:n}) = \mathrm{Unique}(A_1 \| \cdots \| A_n)\,, \tag{20}$$

$$C(y_{1:n}) = \mathrm{Filter}_{\mathcal{L}}(P(y_{1:n})), \qquad I(C(y_{1:n})) = \bigwedge_{c \in C(y_{1:n})} c. \tag{21}$$

This is the shared rollout–decompose–pool–filter–compose operator. The filter checks preservation under the complete current conjunction, so clauses from different responses may support one another; standalone preservation is not required.

A.4 Complete Inference Algorithm

Algorithm 2 CRAFT inference

Input: task $\mathcal{T}$ from Definition 1, policy π, number of responses n
Output: filtered clause set C, composed invariant $I(C)$, final verdict v

$x \leftarrow \mathrm{Mask}(\mathcal{T}) = \mathcal{L}$ ▷ *hide goals*
sample $y_{1:n} \sim \pi(\cdot \mid x)$ ▷ *rollout*
$A_i \leftarrow \mathrm{Admit}(y_i)$ for every i ▷ *decompose and cap*
$P \leftarrow \mathrm{Unique}(A_1 \| \cdots \| A_n)$ ▷ *pool*
$C \leftarrow \mathrm{Filter}_{\mathcal{L}}^{\delta,h_{\max}}(P)$ ▷ *filter*
form $I(C)$ as in Definition 1 ▷ *compose*
$v \leftarrow \mathrm{TaskVerify}_{\delta}(\mathcal{T}, C)$ ▷ *restore goals*
return $(C, I(C), v)$

For evaluation, our primary metric compose@n is the percentage of tasks for which $\mathrm{TaskVerify}(\mathcal{T}, C(y_{1:n}))$ succeeds on the first n saved responses. We use pass@n as a standalone baseline: the percentage of tasks for which at least one of these same n responses independently verifies the targets (Appendix H).

A.5 Complete SFT Label Construction

SFT distills the same multi-response map into one response. It accumulates raw candidates so that a clause discarded in one round can later acquire a supporting companion. The released corpus instantiates π_0 with Qwen3-8B and shares the resulting labels across all trained backbones; π_0 is the only learned generator in label construction. Executions of $\mathcal{L}$ produce the trace signal, and $\mathrm{Filter}_{\mathcal{L}}$ supplies the verifier decision; no other model supplies candidates, labels, or acceptance judgments.

Algorithm 3 Multi-response SFT synthesis

Input: masked loop $\mathcal{L}$, policy π_0, negatives $\mathcal{N}$, group size g_{sft}, round limit t_{sft}
Output: one serialized label, or reject

if $\mathcal{N} = \varnothing$ **then return** reject
$P \leftarrow \langle\rangle$
for $t = 1, \ldots, t_{\mathrm{sft}}$ **do**
  sample a response group $y_{1:g_{\mathrm{sft}}} \sim \pi_0(\cdot \mid \mathcal{L})$
  $P \leftarrow \mathrm{Unique}(P \| \mathrm{Admit}(y_1) \| \cdots \| \mathrm{Admit}(y_{g_{\mathrm{sft}}}))$
  $C \leftarrow \mathrm{Filter}_{\mathcal{L}}(P)$
  if $\mathrm{cov}_{\mathcal{N}}(C) \geq \tau$ **then return** $\mathrm{Serialize}(C)$
return reject

The SFT input and label contain no element of $\mathcal{G}$. The parameter values appear in Appendix C.

A.6 Complete RL Group-Scoring Algorithm

The sampled policy is the same small open-weight model updated by RL. Given its rollouts, every reward component below is computed from program executions, verifier outcomes, and deterministic clause provenance; no external model enters the scoring function.

Executions of the masked loop yield reachable loop-head states $\mathcal{E}$ and target-independent negative traces $\mathcal{N}$. The lightweight evaluator is three-valued:

$$\mathrm{Eval}(c, s) \in \{\mathsf{true}, \mathsf{false}, \mathsf{unknown}\}. \tag{22}$$

An unknown evaluation neither rejects a negative trace nor removes a candidate during reachable-state screening; surviving candidates are passed to the verifier for syntax and induction checks. A trace counts as rejected only when at least one retained clause evaluates definitively to false at one of its states:

$$\mathrm{rej}_{\mathcal{N}}(C) = \{\nu \in \mathcal{N} : \exists s \in \nu, \exists c \in C,\ \mathrm{Eval}(c, s) = \mathsf{false}\}, \qquad \mathrm{cov}_{\mathcal{N}}(C) = \frac{|\,\mathrm{rej}_{\mathcal{N}}(C)|}{|\mathcal{N}|}. \tag{23}$$

Coverage is trajectory level and is defined only for $|\mathcal{N}| > 0$. Instances without retained negatives are rejected during SFT construction. For RL, the explicit fallback below uses a binary survivor signal and never evaluates this ratio.

For an RL group $y_{1:g_{\mathrm{rl}}}$, first form the same globally deduplicated pool used at inference. Let $\mathrm{Screen}_{\mathcal{E}}(P)$ denote the order-preserving front end that removes malformed or out-of-scope clauses and any clause c with $\mathrm{Eval}(c, s) = \mathsf{false}$ for some $s \in \mathcal{E}$. Unknown evaluations do not remove a clause. Apply this screen to the pooled candidates, then invoke the verifier filter once:

$$A_i = \mathrm{Admit}(y_i), \qquad P^* = \mathrm{Unique}(A_1 \| \cdots \| A_{g_{\mathrm{rl}}}), \tag{24}$$

$$P_{\mathcal{E}} = \mathrm{Screen}_{\mathcal{E}}(P^*), \qquad C^* = \mathrm{Filter}_{\mathcal{L}}(P_{\mathcal{E}}),$$

$$V_i = \langle c \in A_i \mid \mathrm{key}(c) \in \mathrm{keys}(C^*) \rangle, \qquad R_i = \mathrm{rej}_{\mathcal{N}}(V_i). \tag{25}$$

Here V_i assigns each pooled survivor back to every response that proposed it. The verifier processes the screened pool $P_{\mathcal{E}}$. If $f(\nu) = |\{j : \nu \in R_j\}|$, the reward is

$$r_i = w_c \frac{|R_i|}{|\mathcal{N}|} + w_g \left(\frac{1}{|\mathcal{N}|} \sum_{\nu \in R_i} \frac{1}{f(\nu)} \right). \tag{26}$$

Here $w_c = 1.0$ and $w_g = 0.3$ weight coverage and group credit. The first term rewards each response by the negative coverage of its pooled survivors. The second is the response-level Shapley value of the fixed post-filter coverage game $u(A) = |\bigcup_{i \in A} R_i|/|\mathcal{N}|$: repeated coverage is shared among its proposers, while unique coverage receives full credit. Thus $1/f(\nu)$ is a group-local difficulty weight: negatives rejected by many responses receive less credit per response than rare negatives rejected by few. The negatives are nonuniformly weighted for each response, while their credits still sum to one unit per covered trace across the group. The per-response cap limits admission without penalizing excess output.

Algorithm 4 CRAFT reward for one RL group

Input: masked loop $\mathcal{L}$, responses $y_{1:g_{\mathrm{rl}}}$, reachable states $\mathcal{E}$, negative traces $\mathcal{N}$
Output: response rewards $r_{1:g_{\mathrm{rl}}}$

$A_i \leftarrow \mathrm{Admit}(y_i)$ for every i
$P^* \leftarrow \mathrm{Unique}(A_1 \| \cdots \| A_{g_{\mathrm{rl}}})$ ▷ *pool globally*
$P_{\mathcal{E}} \leftarrow \mathrm{Screen}_{\mathcal{E}}(P^*)$ ▷ *screen pooled clauses on reachable states*
$C^* \leftarrow \mathrm{Filter}_{\mathcal{L}}(P_{\mathcal{E}})$ ▷ *verify the screened pool once*
$V_i \leftarrow \langle c \in A_i \mid \mathrm{key}(c) \in \mathrm{keys}(C^*) \rangle$ for every i ▷ *project by provenance*
if $\mathcal{N} = \varnothing$ **then**
  $r_i \leftarrow \mathbf{1}[A_i \neq \langle\rangle \ \wedge \ \mathrm{keys}(V_i) = \mathrm{keys}(A_i)]$ for every i
  return $r_{1:g_{\mathrm{rl}}}$
$R_i \leftarrow \mathrm{rej}_{\mathcal{N}}(V_i)$ for every i
$f(\nu) \leftarrow |\{j : \nu \in R_j\}|$ for every $\nu \in \mathcal{N}$
$r_i \leftarrow w_c |R_i|/|\mathcal{N}| + w_g |\mathcal{N}|^{-1} \sum_{\nu \in R_i} f(\nu)^{-1}$ for every i
return $r_{1:g_{\mathrm{rl}}}$

A clause c *semantically supports* d in D when $c \neq d$, $\mathrm{Pres}_{\mathcal{L},D}(d)$ holds, but $\mathrm{Pres}_{\mathcal{L},D\setminus\{c\}}(d)$ does not. The implemented verifier exposes the broader, resource-dependent relation

$$\begin{aligned} \mathrm{VSupport}_{\mathcal{L},\delta}(c, d \mid D) \iff c \neq d \ \wedge \mathrm{LoopVerify}_{\mathcal{L},\delta}(D)[d] = \mathsf{valid} \ \wedge \\ \mathrm{LoopVerify}_{\mathcal{L},\delta}(D \setminus \{c\})[d] \neq \mathsf{valid}. \end{aligned} \tag{27}$$

This relation includes semantic support, generated-goal dependencies, and resource effects. Algorithm 4 can retain such a clause through joint filtering.

A.6.1 Shapley Credit: Definition and Equivalence

Fix the survivor bundles from one pooled filtering call and write $R_i = \mathrm{rej}_{\mathcal{N}}(V_i)$ and $f(\nu) = |\{j : \nu \in R_j\}|$. For nonempty $\mathcal{N}$,

$$\begin{aligned} r_i^{\text{Clause-decomposed}} &= \frac{|R_i|}{|\mathcal{N}|}, & \phi_i &= \frac{1}{|\mathcal{N}|} \sum_{\nu \in R_i} \frac{1}{f(\nu)}, \\ r_i^{\mathrm{Full}} &= r_i^{\text{Clause-decomposed}} + 0.3\phi_i. \end{aligned} \tag{28}$$

Equivalence to the standard Shapley value. Equation (28) gives a closed form for the standard response-level Shapley value. For response i, let V_i be its projected bundle of clauses that survive filtering of the full response group, and let $R_i = \mathrm{rej}_{\mathcal{N}}(V_i)$. A coalition S contributes the projected bundles of its responses, whose rejected-trace set is $\bigcup_{i \in S} R_i$.

Proposition 2 (Equivalence to standard Shapley)**.** *Let $\mathcal{J} = \{1, \ldots, g\}$ index the responses above, and define their coverage game by*

$$u(S) = \frac{1}{|\mathcal{N}|} \left| \bigcup_{j \in S} R_j \right|, \qquad S \subseteq \mathcal{J}.$$

The standard Shapley value of response i,

$$\mathrm{Sh}_i(u) = \sum_{S \subseteq \mathcal{J} \setminus \{i\}} \frac{|S|!\,(g - |S| - 1)!}{g!} \big(u(S \cup \{i\}) - u(S)\big), \tag{29}$$

is exactly

$$\mathrm{Sh}_i(u) = \frac{1}{|\mathcal{N}|} \sum_{\nu \in R_i} \frac{1}{f(\nu)} = \phi_i, \qquad f(\nu) = |\{j : \nu \in R_j\}|.$$

Consequently, $\sum_{i \in \mathcal{J}} \phi_i = u(\mathcal{J})$.

Proof. The coefficient in Equation (29) is the fraction of permutations of $\mathcal{J}$ in which exactly the members of S precede i: there are $|S|!$ orders before i, $(g - |S| - 1)!$ orders after it, and $g!$ orders in total. Hence the standard subset definition equals the expected marginal contribution of response i under a uniformly random permutation.

For each trace ν, define $Q_\nu = \{j \in \mathcal{J} : \nu \in R_j\}$. If $Q_\nu = \varnothing$, the trace never contributes to the game. Otherwise, it changes the coalition value exactly once in any permutation: when the first response from Q_ν is added. By symmetry, each member of Q_ν is first with probability $1/|Q_\nu| = 1/f(\nu)$. Thus trace ν contributes $1/(|\mathcal{N}|f(\nu))$ to $\mathrm{Sh}_i(u)$ when $\nu \in R_i$, and zero otherwise. Summing these contributions over traces gives Equation (28). Finally, every trace covered by the grand coalition allocates a total of $f(\nu) \cdot 1/(|\mathcal{N}|f(\nu)) = 1/|\mathcal{N}|$, proving efficiency. □

This equivalence is evaluated on the fixed response-level bundles $V_1, \ldots, V_g$ projected from the globally filtered survivor set. It therefore proves that the implemented allocation is the standard Shapley value of this response-level coverage game. It assigns credit to responses using fixed post-filter survivor bundles.

A.7 Group-Relative Policy Update

The rollout policy $\pi_{\theta_{\mathrm{old}}}$ samples a group, which is scored by Algorithm 4. Define

$$\widehat{A}_i = \frac{r_i - \mathrm{mean}_j\, r_j}{\mathrm{std}_j\, r_j + \delta_{\mathrm{adv}}}. \tag{30}$$

Let T_i be the number of generated tokens in response i. For token $a_{i,u}$ and its prefix $h_{i,u} = (\mathcal{L}, a_{i,<u})$, let

$$\rho_{i,u}(\theta) = \frac{\pi_\theta(a_{i,u} \mid h_{i,u})}{\pi_{\theta_{\mathrm{old}}}(a_{i,u} \mid h_{i,u})}, \tag{31}$$

$$q_{i,u}(\theta) = \frac{\pi_{\mathrm{ref}}(a_{i,u} \mid h_{i,u})}{\pi_\theta(a_{i,u} \mid h_{i,u})}, \qquad k_{i,u}(\theta) = q_{i,u} - \log q_{i,u} - 1. \tag{32}$$

The nonnegative quantity $k_{i,u}$ is the sampled conditional-token KL surrogate used by the implementation. Its expectation equals $D_{\mathrm{KL}}(\pi_\theta(\cdot \mid h_{i,u}) \| \pi_{\mathrm{ref}}(\cdot \mid h_{i,u}))$ when the token is sampled from $\pi_\theta(\cdot \mid h_{i,u})$. During an update, however, the saved token comes from $\pi_{\theta_{\mathrm{old}}}$, so $k_{i,u}$ is used as a sampled penalty under the saved old-policy tokens. With clipping half-width $\varepsilon > 0$, the group objective is

$$\mathcal{J}(\theta) = \mathbb{E}\left[\frac{1}{g_{\mathrm{rl}}} \sum_i \frac{1}{T_i} \sum_{u=1}^{T_i} \left(\min\left\{\rho_{i,u}\widehat{A}_i, \mathrm{clip}(\rho_{i,u}, 1-\varepsilon, 1+\varepsilon)\widehat{A}_i\right\} - \beta k_{i,u}\right)\right]. \tag{33}$$

After optimizing this objective, the updated policy samples the next group.

A.7.1 Group Reward Collapse and Shapley Tie-Breaking

Here *group reward collapse* means that all rewards in one sampled GRPO group are equal. Equation (30) then gives zero relative advantage to every response, so that group supplies no reward-driven policy-gradient signal; a separate KL term can still act. This condition concerns reward discrimination and does not require the responses themselves to be identical.

If $r_i^{\text{Clause-decomposed}} = c$ for every response in a group, then

$$r_i^{\mathrm{Full}} - \overline{r}^{\mathrm{Full}} = 0.3(\phi_i - \overline{\phi}), \qquad \mathrm{Var}_i(r_i^{\mathrm{Full}}) = 0.09\,\mathrm{Var}_i(\phi_i).$$

Thus Full restores a nonzero relative signal whenever equal-coverage responses have different Shapley credits. It can distinguish which responses contribute less redundant coverage after Clause-decomposed has lost that ranking signal.

For an illustrative calculation, let $\mathcal{N} = \{a, b, c\}$ and three responses reject $R_1 = \{a, b\}$, $R_2 = \{a, c\}$, and $R_3 = \{a, c\}$, respectively. All three Clause-decomposed rewards equal $2/3$, but the trace frequencies are $f(a) = 3$, $f(b) = 1$, and $f(c) = 2$. Their Shapley credits are $(4/9, 5/18, 5/18)$, giving Full rewards $(0.80, 0.75, 0.75)$. Full therefore supplies positive relative credit to the response that uniquely rejects b. This is a worked example of the reward mechanism. It is illustrative and does not report a training outcome.

Full does not guarantee a noncollapsed group. If every response rejects the same trace set, both Clause-decomposed coverage and Shapley credit remain equal; all-zero coverage also remains uninformative. The mechanism can resolve some coverage-score ties, but establishing a lower collapse frequency in training requires logged groups from the two policies.

A.8 Training–Inference Alignment

Inference, SFT label construction, and RL group scoring share the pool–filter–compose core in Equation (21). RL first screens its pooled candidates against observed reachable states before filtering. Inference checks the restored goals, SFT serializes an accepted fixed point, and RL projects the fixed point back to the responses that proposed its surviving clauses. Negative coverage supplies the target-independent training surrogate; restored-goal verification supplies the final verdict.

A.9 Extension to Verifiably Composable Generation

Definition 3 (Verifiably composable generation)**.** A task provides visible input x, hidden goals $\mathcal{G}$, an atomic decomposition parts_x returning an ordered sequence of admitted atoms, an order-preserving duplicate-removal operator Unique_x, a subsequence filter Filter_x, a composition operator $\mathrm{Compose}_x$, and a validity predicate Valid_x. For responses $y_{1:n}$, define

$$P_x(y_{1:n}) = \mathrm{Unique}_x(\mathrm{parts}_x(y_1) \| \cdots \| \mathrm{parts}_x(y_n)) \,, \tag{34}$$

$$O_x(y_{1:n}) = \mathrm{Compose}_x(\mathrm{Filter}_x(P_x(y_{1:n}))) \,. \tag{35}$$

The task is verifiably composable when the filter is conservative and sound: it returns atoms from its input and $\mathrm{Valid}_x(O_x(y_{1:n}))$ holds whenever the filter succeeds. It is target hidden when every operator above depends only on x; the goals $\mathcal{G}$ are used only by a final task-specific check. Complementary composition occurs when $O_x(y_{1:n})$ passes that final check although no individual $O_x(y_i)$ does.

Loop-invariant generation instantiates the atoms with invariant clauses, parts_x with Admit, composition with conjunction, validity with joint inductiveness, and the filter with bounded Houdini. The transfer property for any other instantiation satisfying Definition 3 is proved in Appendix A.10.

Definition 3 identifies the conditions needed to reuse CRAFT: responses must decompose into reusable atoms, atoms must admit pooling, a sound domain-specific filter must validate the pooled result, and provenance must map accepted atoms back to responses. Under these conditions, the same inference and training construction applies after replacing the invariant clauses, conjunction, and Houdini checks by the domain's atoms, composition operator, and verifier.

For *specification generation*, atoms can be precondition, postcondition, frame, or invariant clauses; composition is conjunction; and the filter checks syntax, consistency, and the corresponding proof obligations. Separate responses can contribute complementary clauses to one verified contract. For *testcase generation*, atoms are executable tests; composition is set union; the filter removes invalid or duplicate tests; and structural or behavioral coverage supplies a target-independent signal. Separate responses can cover disjoint behaviors even when no one response forms an adequate suite.

These domains share the formal interface and algorithms. Each domain supplies its verifier and measurable training signal.

A.10 Theoretical Properties and Proofs

This subsection states and proves the properties of the algorithms defined above. Write $\mathcal{L} = (\mathsf{Init}, B, S)$.

A.10.1 Conjunction of Valid Invariants

Proposition 4 (Closure under conjunction)**.** *If $I_1, \ldots, I_m$ are valid invariants of the same loop, then $\bigwedge_j I_j$ is valid and is at least as strong as every I_j.*

Proof of Proposition 4. For each j, validity gives $\mathsf{Init} \Rightarrow I_j$; hence $\mathsf{Init} \Rightarrow \bigwedge_j I_j = I$. Suppose $I \wedge B$ holds before one execution of S. Then $I_j \wedge B$ holds for every j, so preservation of I_j establishes every I_j afterward. Therefore I is preserved and is a valid invariant.

Because a conjunction implies each conjunct, I is at least as strong as every I_j. □

A.10.2 Mutual Support Among Arbitrarily Many Clauses

Proposition 5 (Mutual support)**.** *For a finite clause set C, $I(C)$ is valid if*

$$\mathsf{Init} \Rightarrow I(C), \qquad \forall c \in C : \ \mathrm{Pres}_{\mathcal{L},C}(c). \tag{36}$$

No clause must be preserved in isolation.

Proof of Proposition 5. The first premise is exactly establishment of $I(C)$. For preservation, assume $I(C) \wedge B$ before executing S. The second premise establishes each c_j afterward. Since every conjunct then holds in the same successor state, their conjunction $I(C)$ also holds. Thus $\{I(C) \wedge B\} S \{I(C)\}$, and $I(C)$ is valid.

No standalone preservation premise appears in this argument. A clause may use all other conjuncts in the pre-state, so clauses that fail preservation separately may still be preserved jointly. Establishment is different: $\mathsf{Init} \Rightarrow I(C)$ implies $\mathsf{Init} \Rightarrow c_j$ for every j. Hence a candidate false at some loop-entry state cannot be rescued by conjunction. □

A.10.3 Greatest Jointly Inductive Candidate Subset

Let $C_0 = C$, and let $C_{j+1} \subseteq C_j$ be the set remaining after one ideal Houdini deletion round in Equation (15). Because C is finite, this descending sequence reaches the fixed point $\mathsf{H}_{\mathcal{L}}(C)$.

Theorem 6 (Candidate-relative optimality)**.** *With exact logical decisions, the Houdini fixed point $\mathsf{H}_{\mathcal{L}}(C)$ is jointly inductive and contains every jointly inductive subset of C. Its conjunction is therefore the strongest invariant obtainable by selecting clauses from this pool.*

This candidate-relative property applies to ideal Houdini with the supplied candidate pool. Bounded filtering may discard a valid clause after timeout. Joint checking permits mutual support in preservation, but cannot rescue a clause false at loop entry.

Proof of Theorem 6. At the fixed point, every $c \in \mathsf{H}_{\mathcal{L}}(C)$ satisfies establishment and preservation under the full conjunction $I(\mathsf{H}_{\mathcal{L}}(C))$. Applying Proposition 5 shows that this conjunction is valid.

It remains to prove greatestness. Let $C' \subseteq C$ be any jointly inductive subset. We show by induction over deletion rounds that $C' \subseteq C_j$. The base case $C' \subseteq C_0 = C$ is immediate. Assume $C' \subseteq C_j$, and take any $c \in C'$. Joint establishment of C' gives $\mathsf{Init} \Rightarrow c$, so c cannot be deleted for establishment. Joint preservation gives

$$\{I(C') \wedge B\}\, S\, \{c\}.$$

Since $C' \subseteq C_j$, the antecedent $I(C_j) \wedge B$ implies $I(C') \wedge B$. Strengthening a valid Hoare precondition preserves validity, so

$$\{I(C_j) \wedge B\}\, S\, \{c\}$$

also holds. Therefore no member of C' is deleted in round j, proving $C' \subseteq C_{j+1}$. At termination, $C' \subseteq \mathsf{H}_{\mathcal{L}}(C)$. Because conjunction reverses set inclusion, $I(\mathsf{H}_{\mathcal{L}}(C))$ implies $I(C')$. □

Pooling before filtering. Each separately filtered set $\mathsf{H}_{\mathcal{L}}(A_i)$ is jointly inductive. By closure under conjunction, their union is a jointly inductive subset of $\bigcup_i A_i$. Theorem 6 therefore gives Equation (1). The inclusion can be strict when a clause requires a supporting hypothesis from another response. This ideal property does not ensure empirical monotonicity under finite budgets: pooling changes proof obligations and may trigger timeouts (Appendix K.6).

A.10.4 Soundness of Resource-Bounded Houdini

Proposition 7 (Bounded-filter soundness)**.** *Assume* $\mathrm{LoopVerify}_{\mathcal{L},\delta}$ *is sound. If* $\mathrm{Filter}_{\mathcal{L}}^{\delta,h_{\max}}(C)$ *returns a proved fixed point* D*, then* $D \subseteq C$ *and* $I(D)$ *is inductive. Timeout, unknown, and exhaustion of the round budget cannot cause an unproved clause set to be returned.*

Proof. Every deletion round in Equation (16) selects a subset of its input, so $D \subseteq C$. At a returned fixed point, every clause maps to valid in that same family. Verifier soundness in Equation (12) therefore gives $\mathrm{ValidInv}_{\mathcal{L}}(I(D))$. A timeout or unknown removes the affected clause, and reaching $h_{\max}$ returns the empty result rather than an intermediate set by Equation (18). □

A.10.5 Supporting Clauses and Credit

Proposition 8 (Coverage credit omits pure support)**.** *Suppose response* i *contributes a retained clause* c *that supports another retained clause according to Equation* (27)*, but* $\mathrm{rej}_{\mathcal{N}}(V_i) = \varnothing$*. Then the coverage term assigned to response* i *is zero, irrespective of whether removing* c *would prevent the other clause from surviving.*

Proof. The premise gives $R_i = \varnothing$. Therefore $|R_i| = 0$, so the coverage term in Equation (26) is zero. □

This formal edge case has a negligible aggregate effect in our evaluation. Post-filtering permits supporting clauses from different responses to survive jointly. Pre-filtering removes clauses that cannot survive within their own response. Under the longer verifier budget, their compose@k difference ranges from -0.2 to $+0.6$ percentage points across the reported budgets (Table 21). Thus, although the post-filter coverage score does not credit pure indirect support, such cross-response dependencies have negligible aggregate effect on verification in this benchmark.

A.10.6 Transfer to Verifiably Composable Problems

Proposition 9 (Composable transfer)**.** *For any instantiation satisfying Definition 3, the pooled output* $O_x(y_{1:n})$ *contains only atoms drawn from the responses and satisfies* Valid_x *whenever filtering succeeds. Moreover, its construction is independent of the hidden goals* $\mathcal{G}$*.*

Proof. Decomposition and union draw atoms only from $y_{1:n}$; Unique_x removes candidates and the conservative filter returns a subset, establishing provenance to the responses. Filter soundness gives $\mathrm{Valid}_x(O_x(y_{1:n}))$. Finally, every operator in Equation (35) is a function of x and the responses, not $\mathcal{G}$, so hidden goals enter only the final check. □

B CASE STUDY

Numerical loops, including both linear and nonlinear ones, are a primary subject of prior work on loop-invariant inference (Si et al., 2020; Yu et al., 2023; Kamath et al., 2024). We use this setting to examine target-hidden invariant generation, as well as the reuse and composition of generated invariants. In the target-hidden cases, the generator sees the loop and its entry context, while the target assertion is withheld until final verification. Case 1 shows recovery of a hidden algebraic relation. Case 2 shows an invariant stronger than its target and compares the actual invariant candidates of other tools. Case 3 shows how an invariant inferred for the first of two loops helps verify the final assertion. Case 4 shows how independently inductive clauses from different responses can jointly prove a target.

CASE 1: TARGET-HIDDEN INVARIANT DISCOVERY.

```
void foo(int k, int z) {
  int x = 1, y = 1, c = 1;
  while (c < k) { c++; x = x*z + 1; y = y*z; }
  /*@ assert 1+x*z-x-z*y == 0; */
}
```

If the assertion is visible, it can be copied as a loop-invariant candidate. It holds initially because $x = y = 1$. Writing $I(x, y) = 1 + xz - x - zy$, one loop iteration gives

$$I(xz + 1, yz) = 1 + (xz + 1)z - (xz + 1) - z(yz) = zI(x, y),$$

so $I = 0$ is inductive and discharges the exit assertion. Under target hiding, the model sees the initialization and recurrences while the assertion is withheld, and must recover the relation. Archived rollout pools recover three algebraically equivalent forms:

Model	First verified prefix	Rediscovered invariant clause
GPT-5	compose@1	$x - 1 = z(x - y)$
DeepSeek V4-Flash	compose@4	$x(z - 1) = zy - 1$
Claude Sonnet-4.6	compose@8	$x(z - 1) = yz - 1$

All three clauses rearrange to the hidden assertion. None of the ten raw responses in these pools passes as a whole on this program; pooled filtering removes incompatible siblings and retains the useful relation.

CASE 2: A STRONGER INVARIANT WHERE BASELINES FAIL.

The loop alternates which counter is incremented; its hidden target asks whether the counters agree after $2k$ iterations.

```
void foo(int b, int j, int n) {
  int k = 100, i = j;
  n = 0;
  while (n < 2*k) {
    if (b) i++; else j++;
    b = !b; n++;
  }
  /*@ assert i == j; */
}
```

Table 2 shows the saved candidate clauses; b_0, j_0, n_0 denote input values. CRAFT uses the first eight target-hidden GPT-5-nano responses, while baseline rows show their archived outputs.

Method	Saved invariant candidates	Full proof
CRAFT	$0 \le n \le 2k, \quad i + j = 2j_0 + n, \quad n \bmod 2 = 0 \Rightarrow i = j;$	4/4
	$n \bmod 2 = 1 \Rightarrow (b = 0 \wedge i - j = 1) \vee (b = 1 \wedge i - j = -1)$	
AutoSpec	$0 \le n, \quad n \le 2k$	0/4
Naive	$0 \le n \le 2k;$	0/4
	$i - j = (n \bmod 2 = 0)\ ?\ 0 : (b\ ?\ -1 : 1)$	
Daikon	$n \ge 0,\ k = 100,\ i \ge j_0,\ j \ge j_0;$	0/4
	$j_0 \ne 0,\ b \ne k,\ b \ne j_0,\ k \ne b_0,\ k \ne j_0;$	
	$k \ne n_0,\ b_0 \ne j_0,\ b_0 \ne n_0,\ j_0 \ne n_0$	
SESpec	No loop annotation	—
Clause2Inv	Unsupported on this instance	—

Table 2: Archived candidate clauses for Case 2. Full proof counts separately checked targets with invariant induction, not merely exit assertions.

At exit, CRAFT's parity and sum clauses prove the hidden $i = j$ and three additional goals: $i + j = 2j_0 + n$, $i = j_0 + n/2$, and $j = j_0 + n/2$. Naive's exit assertion is locally provable, but WP cannot establish its candidate's preservation; Daikon even proposes $j_0 \ne 0$, which the program does not guarantee. The extra goals were added after target-hidden generation and do not change benchmark scores.

CASE 3: A LOOP SUMMARY FOR SUBSEQUENT COMPUTATION.

An invariant can serve as an intermediate assertion even when the final target concerns later computation. Consider two sequential numerical loops:

```
/*@ requires 0 <= n <= 100; */
void foo(int n) {
  int i = 0, s = 0;
  while (i < n) {
    s += 2 * i + 1;
    i++;
  }
  int j = 0;
  while (j < n) {
    s -= n;
    j++;
  }
  /*@ assert s == 0; */
}
```

The final assertion describes the result of both loops, rather than the first loop alone. For generation, CRAFT receives only the first loop and its entry context, including the precondition; the second loop and final target are withheld. One response supplies the following clauses, all retained by joint inductive filtering:

$$0 \le i, \qquad i \le n, \qquad s = i^2.$$

At the first loop's exit, $i \ge n$ gives $i = n$, yielding the intermediate assertion $s = n^2$. This summarizes the first loop independently of the later computation. It initializes the second loop's invariant $0 \le j \le n \wedge s = n(n - j)$, which implies the final target $s = 0$ when $j = n$.

For this reuse check, we manually annotated the second loop, including its frame and termination conditions. Frama-C/WP discharged all 29 proof goals with runtime-error checks enabled, demonstrating summary reuse rather than end-to-end multi-loop synthesis or cross-response composition.

CASE 4: COMPOSITION WITHOUT CROSS-RESPONSE SUPPORT.

This case shows that clauses can each survive independent filtering yet require cross-response composition to prove the target. We compare independent and pooled filtering of the first four GPT-5-

nano responses for the same task, using $\mathrm{Filter}_{\mathcal{L}}$. The target assertion is hidden during generation and filtering, then restored for final verification.

```
/*@ requires n >= 0; */
void foo(int n) {
  int x = n, y = 0;
  while (x > 0) {
    y = y + 1;
    x = x - 1;
  }
  /*@ assert y == n; */
}
```

Table 3 reports both the relevant admitted proposal from each response and the result of filtering that response alone.

Response	Relevant proposal	Independent survivors	Target
1	$x \geq 0,\ y \geq 0$	$x \geq 0,\ y \geq 0$	fail
2	—	$\varnothing$	fail
3	$x = n - y,\ y = n - x$	$x = n - y,\ y = n - x$	fail
4	$x \geq 0,\ y \geq 0$	$x \geq 0,\ y \geq 0$	fail
Pooled 1–4	$x \geq 0,\ y \geq 0,\ x = n - y,\ y = n - x$		verified

Table 3: A cross-response composition case without mutual support: every pooled survivor already survives independent filtering, yet no response alone verifies the target.

Every clause retained by pooled filtering also survives independent filtering. The composed annotation is

```
/*@
  loop invariant x >= 0;
  loop invariant y >= 0;
  loop invariant x == n - y;
  loop invariant y == n - x;
*/
```

The proof-minimal cross-response core is $x \geq 0$ from Response 1 (also duplicated by Response 4) and $y = n - x$ from Response 3. Each clause alone fails the restored-target judge. Together, the exit condition gives $x \leq 0$, the bound gives $x = 0$, and the relation yields $y = n$. The gain comes from combining complementary clauses to prove the target.

C Implementation Details

Table 4 lists the implemented defaults. SFT synthesis, RL, and inference share Algorithm 1. RL filters the union once, then uses clause provenance to assign survivors back to their proposing rollouts before computing credit (Appendix K.6).

Symbol	Value	Role
r_{exec}	12	requested input executions; doubled for oracle-driven bodies
$\mathcal{X}_{\text{tier}}$	20; $[-64, 64]$	fixed near-input schedule and clamping interval
q_{far}	3, plus 2 ordered multi-parameter probes	deterministic far probes with magnitude at most 512
Δ_{off}	dense: $\pm\{1, 2\}$; terminal: $\pm\{1, 2, 3, 5, 8\}$	off-trajectory perturbations
h_{dense}	3	successor window used to establish an eligible perturbation base
b_{off}	96	cap on perturbation bases stratified across positives
h_{exit}	24	continuation horizon after a witnessed exit
$b^{-}_{\text{off}}, b^{-}_{\text{exit}}$	48, 12	off-trajectory and post-exit negative-trace budgets
Δ_{fill}	1–34; then 35–128	local-fill offset magnitudes, second stage if needed
$a_{\text{fill}}(q)$	$\max(1024, 100q)$	attempts per stage for remaining capacity q
$u_{\max}$	2×10^6	execution-iteration safety cap
w_c, w_g	1.0, 0.3	coverage and group-credit weights
$m_{\max}$	20 clauses	cap applied independently to each response
$\mathcal{K}$	$\{1, 4, 8, 16, 32\}$	reported local-model response budgets
n_{probe}	128; 100; 32	saved responses per task (8B; A3B/Llama; remaining base models)
ϑ	1.0	decoding temperature except for Loopy (Table 26)
p	1.0	nucleus-sampling probability
L_{api}	8,192 tokens	maximum API-model completion length
L_{local}	2,048 tokens	maximum local-model completion length
g_{sft}	16 responses	group sampled per SFT strengthening round
t_{sft}	5 rounds	maximum per loop; at most 80 responses
τ	0.5	SFT acceptance threshold on composed negative coverage
$h_{\max}$	500	maximum Houdini verification rounds
$\mathcal{P}_{\text{verify}}$	Alt-Ergo and Z3	prover portfolio; Typed memory model
δ	5 seconds per goal	per-prover goal budget used by LoopVerify
$\delta_{\text{short}}, \delta_{\text{long}}$	300, 900 seconds	verification caps in the pre-/post-filtering comparison

Table 4: Sampling, training, and verification parameters.

C.1 Loop-Verifier Realization

The implementation generates establishment and preservation goals for each clause in the ordered current family $D = \langle d_1, \ldots, d_m \rangle$. Establishment of d_k may use earlier annotations d_j, $j < k$, as hypotheses. Preservation uses the full current family as loop-head hypotheses and may use earlier annotations in the successor state. These hypotheses are available during the current round regardless of the status of their own goals. The verifier simplifies and splits the obligations by control-flow path. A clause receives valid only when every generated establishment and preservation subgoal is proved; timeout records an exhausted prover budget, and all other outcomes map to unknown. Houdini reruns the survivors after each deletion round.

C.2 Negative-Trace Sampling

Algorithm 5 constructs candidate negative traces using only visible program information. Budgets and perturbation parameters are listed in Table 4.

Algorithm 5 Target-hidden candidate-negative sampling

Input: masked loop $\mathcal{L}$, input schedule U, configuration Θ
Output: observed states $\mathcal{E}$, candidate traces $\mathcal{N}$

$\widetilde{\mathcal{L}} \leftarrow \mathrm{Determinize}(\mathcal{L})$ ▷ *execution only*
$T \leftarrow \mathrm{Execute}(\widetilde{\mathcal{L}}, U, \Theta)$; $\mathcal{E} \leftarrow \mathrm{UniqueHeads}(T)$
$C_{\mathrm{off}} \leftarrow \varnothing$; $C_{\mathrm{exit}} \leftarrow \langle\rangle$
for each eligible base s in T **do** ▷ *off-trajectory perturbations*
  for each allowed offset δ on one or two tracked variables **do**
    $s^{-} \leftarrow s + \delta$ ▷ *preserve entry snapshots*
    if sampled ranges and guard truth are preserved,
      and δ is not an integer multiple of the observed local step **then**
      add singleton trace $\langle s^{-} \rangle$ to C_{off}
for each eligible uncapped guard-false exit s_{exit} in T **do** ▷ *post-exit continuations*
  $s \leftarrow s_{\mathrm{exit}}$; $\nu \leftarrow \langle\rangle$
  repeat up to h_{exit} times
    $s \leftarrow \mathrm{ExecuteBody}(\widetilde{\mathcal{L}}, s)$ ▷ *ignore the original guard*
    append s to ν
  append ν to C_{exit} ▷ *one continuation per exit*
$\mathcal{N} \leftarrow \mathrm{CleanSelect}(C_{\mathrm{off}}, C_{\mathrm{exit}}, \mathcal{E}, \Theta)$
$\mathcal{N} \leftarrow \mathrm{LocalFill}(\mathcal{L}, T, \mathcal{E}, \mathcal{N}, \Theta)$ ▷ *unused budget*
return $(\mathcal{E}, \mathcal{N})$

Execution and state representation. We execute the target-hidden loop and collect loop-head states with their function-entry and loop-entry snapshots. Perturbations preserve these snapshots to distinguish execution contexts. Supported nondeterministic calls are replaced by sampled parameters fixed within each run for execution; generation and verification still use the original program.

Off-trajectory perturbations. We select eligible base states across observed executions and perturb one or two tracked variables by small integer offsets, preserving the entry snapshots. Single-variable changes use an offset d; paired changes use (d, d) or $(d, -d)$, with magnitudes specified in Table 4. Candidates must remain within their context's sampled coordinate ranges and preserve the guard's known truth value. We discard perturbation vectors that are integer multiples of the observed local execution step, avoiding obvious movements along the trajectory. Each state retained after cleaning forms a singleton candidate negative trace. These perturbations probe relations among variables that simple marginal bounds may fail to capture.

Post-exit continuations. Starting from a witnessed guard-false exit of an uncapped execution, we ignore the original guard and continue executing the same loop body for a bounded number of steps. Unlike off-trajectory perturbations, this construction uses body updates to generate a sequence of states beyond the observed exit, probing boundary constraints. Cleaning removes observed and duplicate states while preserving the order of the remainder; each nonempty continuation forms one candidate negative trace. The continuation horizon is specified in Table 4; capped runs and bodies unsupported for replay do not supply this family.

Cleaning and selection. Cleaning removes observed states, duplicates, and states passing a conservative entry-feasibility screen. Off-trajectory candidates are selected across perturbation strata, followed by nonempty cleaned post-exit groups; local random perturbations fill unused capacity under the same cleaning rules. For a deterministic execution from a fixed complete initial state, candidates outside a fully recorded loop-head trace through termination are unreachable in that entry context. Sparse recording and other limitations are discussed in Appendix L.

Coverage units. Each perturbed state forms a singleton trace, while each cleaned post-exit continuation remains one trace. A trace is rejected if any retained clause is definitely false at any of its states. Each trace contributes one coverage unit regardless of length.

C.3 Clause Normalization

Clause deduplication, which forms the clause set scored above, normalizes comparison direction, equality sides, parentheses, immediate commutative operands, harmless identities, and Boolean association, but avoids transformations that require additional arithmetic assumptions. Repeated clauses therefore collapse without affecting coverage. The reward and its empty-negative fallback are defined in Algorithm 4; the GRPO objective is given in Appendix A.7.

D Generation Prompt

The canonical evaluation template appears below; every target is stripped before insertion, and general annotation rules are supplied by the common system prompt. Both the SFT and RL data use this single prompt pair.

Generation prompt

```
You are given a C function. Produce the strongest inductive ACSL loop
invariants that capture the loop's behavior (conservation/relational laws
+ tight progress bounds). Output ONLY invariant lines, each formatted
exactly as:
loop invariant <expr>;
Output at most 20 invariant lines.

Program:
{program}
```

The accompanying system prompt appears verbatim below.

System prompt

```
You are a formal verification expert specializing in ACSL loop invariant
synthesis for Frama-C/WP.

## LOOP INVARIANT DEFINITION

A loop invariant is a property that:
1. Holds before the first loop iteration (initialization).
2. Is preserved by every loop iteration when the loop guard is true
   (inductiveness).
3. Combined with loop exit (`!guard`), helps establish required
   loop-exit facts.

## RULES

- Add one core conservation/relational equality that explains loop
  updates (state transition law).
- Add minimal progress bounds for loop counters and key arithmetic
  variables.
- Prefer strong invariants; avoid weak/tautological ones.
- Do not add unrelated invariants just to increase count.
- MUST NOT modify or rewrite any `unknown()`/`unknownN()` call.
- `\at(v, Pre)` can ONLY be used with function parameters, never with
  local variables initialized inside the function.
- `\at(v, LoopEntry)` denotes the value of v at the start of the loop
  and can ONLY be used with local variables initialized before the
  loop (using v = unknown();).
- Loop guard is NOT an invariant; weaken strict guards to non-strict
  bounds at exit.
- Parenthesize mixed equality and relational expressions so their
  grouping is explicit.
- Use `loop invariant` only as a line prefix, never inside expressions.
- Emit scalar invariant expressions only; do not declare predicates,
  logic functions, axioms, or lemmas.
- Do NOT use the ternary `? :` operator; split cases with `==>` instead.
- `^` is bitwise XOR in C/ACSL, not exponentiation --- write `x*x*x`
  for x cubed.
- Do NOT place a C boolean directly in arithmetic (`x + (a > b)` is
  invalid); use `==>` to split cases.
- Only use variables declared in the given function.
- Operators: comparison `==` `!=` `<` `<=` `>` `>=`; logical `&&` `||`
  `!` `==>` `<==>`; arithmetic `+` `-` `*` `/` `%`.

## ACSL INVARIANT SYNTAX

Allowed expression forms (all verified by Frama-C/WP):

1. **Bounds**: `0 <= i`, `i <= n`, `x >= 1`
2. **Linear equality**: `i + c == n`, `2 * s == i * (i - 1)`,
   `x == q * y + r`
3. **Polynomial equality**: `x == n * n * n`,
   `6 * s == i * (i + 1) * (2 * i + 1)`
4. **Relational identity** (multi-var algebraic law):
   `1 + x * z - x - z * y == 0`, `p * s - r * q == 1`
5. **Implication**: `i > 0 ==> s >= i`, `a != 0 ==> q >= 1`
6. **Modular**: `i % 5 == 0`, `r >= 0 && r < y`
7. **Bitshift** (power-of-2 only): `p == 1 << i`, `s == (1 << i) - 1`
8. **Conjunction**: `c >= 1 && c <= k`
9. **\at for params**: `n == \at(n, Pre)` (function parameters only)

Forbidden:
- `^` for exponentiation (it is XOR) --- use repeated `*` or `<<` for
  powers of 2.
- `\product`, `\sum`, `\factorial` --- not ACSL scalar operators.
- `? :` ternary --- rewrite with `==>`.
- `\old(x)` --- use `\at(x, Pre)`.
- Type casts `(int)`, `(long)` --- unnecessary in ACSL logic.
- Function calls, predicates, lemmas, axioms --- emit only scalar
  expressions.

Operators (complete list):
  comparison: `==` `!=` `<` `<=` `>` `>=`
  logical:    `&&` `||` `!` `==>` `<==>`
  arithmetic: `+` `-` `*` `/` `%` `<<` `>>`

## OUTPUT

Return ONLY invariant lines, one per line, formatted exactly as:
`loop invariant <property>;`
Do not return code fences, `loop assigns`, the surrounding C function,
or explanations.
```

E TRAINING INTERFACE

The training interface accepts a target-hidden program, sampled responses, and a reward configuration, and returns each response's reward and coverage.

F Program and Structural Overlap Between Training and Evaluation

We compare the RL (9,024 records and 9,024 distinct program sources) and SFT (33,346 records and 31,865 distinct program sources) training sets against all 832 evaluation programs after target removal. We canonicalize simple break-based loop guards and apply three increasingly aggressive normalizations: exact tokens, alpha-renamed identifiers, and alpha-renaming with nontrivial integer constants abstracted. No training program matches an evaluation program under these representations (Table 5).

We separately measure coarse structural overlap. A structural cell combines the control-flow profile, guard kind, nonlinearity, nondeterminism, and a variable-count band. The evaluation set occupies 122 such cells. RL and SFT share 38 and 41 of them, respectively; their union shares 41 cells. Equivalently, 524 (63.0%), 529 (63.6%), and 529 (63.6%) of the 832 evaluation programs fall in a cell represented by RL, SFT, and their union. The record-weighted total-variation distances over cells are 0.688 for RL, 0.651 for SFT, and 0.633 for their union, so the training and evaluation distributions remain different. Table 6 gives interpretable feature marginals behind this joint-cell comparison.

	RL	SFT	RL ∪ SFT
Records	9,024	33,346	42,370
Distinct program sources	9,024	31,865	33,479
Exact-token matches	0	0	0
Alpha-renamed matches	0	0	0
Alpha-renamed, constant-abstracted matches	0	0	0
Shared evaluation cells	38 / 122	41 / 122	41 / 122
Evaluation programs in shared cells	524 (63.0%)	529 (63.6%)	529 (63.6%)
Training records in evaluation-supported cells	5,056 (56.0%)	27,275 (81.8%)	32,331 (76.3%)
TV distance over structural cells	0.688	0.651	0.633

Table 5: Training–evaluation overlap and structural-distribution audit. Distribution statistics are weighted by training-record frequency.

Feature (% of records/programs)	Evaluation	RL	SFT
Constant guard	24.6	10.3	4.6
Single-variable guard	31.4	38.8	44.0
Variable-vs.-variable guard	37.3	49.4	50.6
Compound guard	6.7	1.5	0.8
Nonlinear	2.5	14.3	10.8
Nondeterministic	28.6	41.5	14.9
≤ 2 variables	35.1	14.6	46.2
3–4 variables	33.9	23.1	12.0
≥ 5 variables	31.0	62.3	41.8

Table 6: Marginal structural distributions of evaluation programs and training records.

G Training-Data Curation

Program sources. The raw training-program pool is drawn primarily from SV-COMP and SyGuS (Beyer, 2023; Alur et al., 2013). We retain scalar-integer single-loop programs and hide target assertions and postconditions in the model-facing input. RL uses 9,024 selected programs. SFT labels are synthesized and verified by the multi-round procedure in Section 2.3 with Qwen3-8B as the generator. The corpus is produced once and shared unchanged by every trained backbone; there are no backbone-specific synthesis runs. Table 7 summarizes the training sets. Loop shapes abstract identifiers and nontrivial constants in the guard and body; source overlap compares program text in training prompts.

Training set	Size and composition
RL set	9,024 records and distinct program sources; 1,794 loop shapes
SFT set	33,346 records over 31,865 distinct program sources
RL–SFT exact-source overlap	7,410 distinct program sources

Table 7: Training-set sizes, loop shapes, and source overlap. The SFT set is synthesized once with Qwen3-8B and shared by all trained backbones.

RL selection. The RL set contains unchanged rows selected from the raw program pool, with no repeated source text. Whole-source family caps admit at most three records per identifier-normalized family and eight per family after also abstracting nontrivial constants. The selected set spans 6,760 and 4,196 families under these two representations, respectively. These whole-source families differ from the loop-only shapes in Table 7. Static complexity criteria yield 529 easy, 7,745 medium, and 750 challenging programs; these labels summarize static program complexity. Selection rejects unsupported state, abnormal executions, insufficient negative traces, and sampler runs exceeding 15 seconds. All selected programs produced 60 negative-trace groups under the audited sampler with 12 runs and seed 0. The final records are shuffled into 141 batches of 64.

H Experimental Protocol Details

The evaluation workload composition is given in Section 3.1; the 466 Loopy programs are the integer subset of its original 469-program corpus (three floating-point programs excluded), and CRAFT retains all benchmark tasks in its verification denominator. External tools use their full supported sets, as specified in Appendix K.8.2.

The final judge uses Frama-C 27.1 (Cobalt), the WP plug-in, the Typed memory model, and the Alt-Ergo/Z3 prover portfolio.

Before model invocation, we remove assert and ensures predicates, executable assertion calls, and conventional error-label names; preconditions and executable semantics remain visible. CRAFT evaluation runs use the canonical generation template, extractor, canonical-key deduplication, and Houdini implementation; external tools retain their own orchestration. Each model retains its serving interface and chat template. All API and locally served CRAFT models use reasoning-disabled decoding. Decoding uses temperature ϑ, nucleus parameter p, and completion caps L_{api} and L_{local} for API and local models, respectively, without an additional top-k, repetition penalty, re-roll, or target feedback. Each probe curve reuses the same saved response pool per task. The model-specific pool sizes and the common reported budget grid are listed in Table 4; every pool contains at least $k_{\max}$ responses. For task t, let $s_{t,i} \in \{0,1\}$ be the standalone verifier verdict for saved response i. Let $v_{t,k} \in \{0,1\}$ indicate whether filtering and composing the first k saved responses succeeds. Both metrics use this same prefix:

$$\text{pass}\,@k = \frac{100}{N}\sum_{t=1}^{N} \mathbf{1}[\exists i \le k :\ s_{t,i} = 1]\,,$$

$$\text{compose}\,@k = \frac{100}{N}\sum_{t=1}^{N} v_{t,k},$$

where N is the number of evaluated programs. Rates are reported as percentages of successful tasks.

Training configuration. Tables 8 and 9 record the SFT and RL run configurations. SFT uses LLaMA-Factory, while RL uses verl. Both stages run on eight NVIDIA A800 GPUs, with training seed 42 for the main experiments. SFT uses 3 epochs on 33,346 records; RL uses 2 epochs on 9,024 programs. For Qwen3-8B, SFT and RL take approximately 4 and 9 hours, respectively, on this eight-GPU setup. Reward and sampler constants are reported in Table 4.

Field	Value
Learning rate η_{sft}	1×10^{-5}
Epochs e_{sft}	3
Batch (per device $\times$ accumulation $\times$ GPUs)	global batch 32; $2 \times 2 \times 8$ or $1 \times 4 \times 8$
Sequence cutoff L_{sft}	2,048
DeepSpeed	ZeRO-3
Scheduler	cosine

Table 8: SFT training configurations.

Field	Value
Prompt batch size b_{rl}	64
Learning rate η_{rl}	1×10^{-6}
KL coefficient β	0.001
Advantage stabilizer δ_{adv}	10^{-6}
Rollouts per prompt g_{rl}	8
Tensor parallelism p_{tp}	1–2
GPU memory utilization μ_{gpu}	0.3–0.5
Epochs e_{rl}	2

Table 9: RL training configurations.

I Why Thinking Is Disabled for Local Models

The reasoning-disabled API experiments use compatible endpoint controls; Claude Sonnet disables adaptive thinking with a zero thinking budget. Local models use reasoning-disabled decoding during synthetic-data construction, training rollouts, and evaluation.

Explicit chain-of-thought adds reasoning tokens to every rollout, with total generation scaling with the sampling budget. The required output is a short, machine-parseable set of ACSL clauses. These clauses admit clause-level filtering and cross-rollout composition: Houdini can remove a failing clause, retain its siblings, and merge survivors from different rollouts. Reasoning traces are generated and consumed as whole sequences and are not included in this clause-level composition operator.

J Complete Results for Model Training Configurations

Model naming. Local models are named *Backbone* + *SFT* + *RL*, omitting unused stages. SFT uses composed labels; SFT (single) uses independently filtered responses. RL defaults to Full, while reward ablations are named explicitly (Appendix K.3).

Data and layout. The following tables report the primary compose metric and standalone pass baseline across training stages, rewards, program classes, and sampling budgets. All values are percentages; All denotes the complete benchmark. Bold marks each metric column's maximum across models and budgets, including ties; a dash denotes an unavailable result. Metric aggregation follows Appendix H. Qwen3-8B Full after SFT uses the same evaluation as the main results; the other rewards use the reward-ablation evaluations. Analyses appear in Appendix K.

J.1 BASE MODELS

Table 10: Base-model verification results.

		Linear / 316		NLA / 50		Loopy / 466		All / 832	
Model	k	compose	pass	compose	pass	compose	pass	compose	pass
Qwen3-1.7B	1	19.30	5.38	6.00	2.00	21.03	6.22	19.47	5.65
	4	29.75	9.81	8.00	4.00	30.69	12.23	28.97	10.82
	8	33.23	12.34	8.00	6.00	35.62	15.45	33.05	13.70
	16	35.76	15.19	10.00	**8.00**	37.77	18.88	35.34	16.83
	32	38.29	18.35	**14.00**	**8.00**	40.13	22.10	37.86	19.83
Qwen3-4B	1	33.54	5.06	8.00	0.00	35.19	6.01	32.93	5.29
	4	44.94	10.44	10.00	2.00	47.21	10.52	44.11	9.98
	8	50.00	13.29	10.00	2.00	50.00	13.09	47.60	12.50
	16	52.53	15.82	12.00	4.00	53.65	15.88	50.72	15.14
	32	54.11	18.35	12.00	6.00	56.22	18.88	52.76	17.91
Qwen3-8B	1	37.97	6.01	6.00	2.00	41.20	7.08	37.86	6.37
	4	50.95	12.97	6.00	4.00	55.15	14.16	50.60	13.10
	8	54.75	17.41	10.00	6.00	58.58	18.45	54.21	17.31
	16	56.65	21.84	10.00	6.00	61.16	23.18	56.37	21.63
	32	58.23	**25.63**	10.00	6.00	62.66	28.33	57.81	25.96
Qwen3-14B	1	48.10	6.01	4.00	2.00	44.64	9.44	43.51	7.69
	4	61.39	11.71	10.00	2.00	61.37	17.17	58.29	14.18
	8	64.56	14.87	12.00	4.00	64.81	21.46	61.54	17.91
	16	66.14	18.67	**14.00**	4.00	67.38	26.18	63.70	22.00
	32	**66.46**	22.78	**14.00**	6.00	**68.67**	**30.47**	**64.54**	**26.08**
Qwen3-30B-A3B	1	16.14	1.27	0.00	0.00	20.17	1.50	17.43	1.32
	4	29.43	4.75	4.00	0.00	34.33	4.94	30.65	4.57
	8	36.71	8.23	4.00	0.00	39.91	7.94	36.54	7.57
	16	43.35	13.29	8.00	0.00	45.28	12.66	42.31	12.14
	32	43.99	18.67	8.00	2.00	47.64	18.45	43.87	17.55
Llama 3.1-8B	1	28.16	0.63	6.00	0.00	30.04	0.64	27.88	0.60
	4	48.73	2.22	12.00	0.00	47.85	2.36	46.03	2.16
	8	54.43	4.11	12.00	0.00	53.22	4.08	51.20	3.85
	16	55.06	6.96	12.00	0.00	53.65	6.44	51.68	6.25
	32	55.06	10.76	12.00	0.00	54.51	10.09	52.16	9.74

J.2 SFT MODELS

Table 11: Verification after SFT on composed labels.

Model	k	Linear / 316 compose	Linear / 316 pass	NLA / 50 compose	NLA / 50 pass	Loopy / 466 compose	Loopy / 466 pass	All / 832 compose	All / 832 pass
Qwen3-4B + SFT	1	65.82	43.67	8.00	2.00	66.31	42.27	62.62	40.38
	4	78.48	58.54	10.00	4.00	76.82	58.37	73.44	55.17
	8	81.01	63.29	12.00	6.00	79.83	63.52	76.20	59.98
	16	82.28	67.41	12.00	8.00	80.90	67.81	77.28	64.06
	32	82.28	71.20	12.00	10.00	81.97	71.46	77.88	67.67
Qwen3-8B + SFT	1	68.35	45.57	8.00	4.00	66.95	43.56	63.94	41.95
	4	77.22	61.08	10.00	6.00	78.33	59.44	73.80	56.85
	8	80.06	65.51	10.00	8.00	80.69	63.95	76.20	61.18
	16	81.33	68.99	12.00	8.00	81.55	67.81	77.28	64.66
	32	81.65	72.47	12.00	10.00	82.40	71.24	77.88	68.03
Qwen3-14B + SFT	1	70.25	50.32	12.00	4.00	69.53	47.21	66.35	45.79
	4	81.33	63.92	**16.00**	8.00	80.04	61.37	76.68	59.13
	8	82.28	68.04	**16.00**	10.00	82.40	66.31	78.37	63.58
	16	**82.91**	71.52	**16.00**	12.00	83.48	70.60	79.21	67.43
	32	**82.91**	**74.05**	**16.00**	**14.00**	**83.69**	**74.25**	**79.33**	**70.55**
Llama 3.1-8B + SFT	1	67.09	43.67	8.00	2.00	63.09	40.34	61.30	39.30
	4	75.32	56.01	14.00	4.00	72.96	55.36	70.31	52.52
	8	77.53	60.13	14.00	6.00	76.39	60.09	73.08	56.85
	16	78.48	63.61	14.00	8.00	77.47	63.95	74.04	60.46
	32	78.80	66.77	14.00	10.00	77.90	67.38	74.40	63.70

J.3 SFT MODELS WITH SINGLE-RESPONSE LABELS

Table 12: Verification after SFT on single-response labels.

Model	k	Linear / 316 compose	Linear / 316 pass	NLA / 50 compose	NLA / 50 pass	Loopy / 466 compose	Loopy / 466 pass	All / 832 compose	All / 832 pass
Qwen3-4B + SFT (single)	1	28.48	20.89	0.00	0.00	22.32	17.81	23.32	17.91
	4	40.82	34.18	0.00	0.00	32.62	28.11	33.77	28.73
	8	45.57	40.51	2.00	2.00	39.70	33.91	39.66	34.50
	16	49.05	43.67	6.00	4.00	43.99	38.63	43.63	38.46
	32	**52.85**	**47.47**	**10.00**	4.00	**50.86**	44.21	**49.16**	43.03
Qwen3-8B + SFT (single)	1	27.22	21.84	0.00	0.00	22.96	20.82	23.20	19.95
	4	38.29	33.23	0.00	0.00	32.19	30.04	32.57	29.45
	8	43.04	38.92	2.00	2.00	38.41	35.41	37.98	34.74
	16	48.10	42.72	2.00	2.00	42.49	39.70	42.19	38.58
	32	51.90	46.20	8.00	**8.00**	49.14	45.06	47.72	43.27
Qwen3-14B + SFT (single)	1	27.53	21.20	0.00	0.00	20.60	18.24	22.00	18.27
	4	36.08	31.01	0.00	0.00	32.40	29.83	31.85	28.49
	8	39.87	36.39	2.00	0.00	38.20	34.76	36.66	33.29
	16	46.20	42.41	2.00	0.00	44.85	41.20	42.79	39.18
	32	50.00	45.89	6.00	2.00	50.21	**46.57**	47.48	**43.63**
Llama 3.1-8B + SFT (single)	1	25.95	19.94	0.00	0.00	21.24	18.67	21.75	18.03
	4	36.39	30.38	0.00	0.00	30.04	27.47	30.65	26.92
	8	38.61	35.13	2.00	0.00	34.55	31.76	34.13	31.13
	16	42.09	38.92	2.00	0.00	40.13	35.62	38.58	34.74
	32	46.52	40.82	4.00	0.00	44.21	39.27	42.67	37.50

J.4 RL DIRECTLY FROM BASE

Table 13: Verification after RL from Base with the Full reward.

Model	k	Linear / 316 compose	Linear / 316 pass	NLA / 50 compose	NLA / 50 pass	Loopy / 466 compose	Loopy / 466 pass	All / 832 compose	All / 832 pass
Qwen3-4B + RL	1	52.22	2.22	10.00	0.00	50.64	1.50	48.80	1.68
	4	63.61	4.11	10.00	0.00	61.59	3.22	59.25	3.37
	8	66.46	5.38	12.00	0.00	65.24	4.29	62.50	4.45
	16	69.30	6.65	12.00	2.00	67.17	5.58	64.66	5.77
	32	70.25	8.23	12.00	2.00	68.67	7.30	65.87	7.33
Qwen3-8B + RL	1	54.43	2.53	6.00	0.00	54.94	3.43	51.80	2.88
	4	61.71	4.75	10.00	0.00	65.02	6.44	60.46	5.41
	8	66.46	7.59	12.00	0.00	68.24	8.80	64.18	7.81
	16	71.52	9.18	12.00	0.00	73.18	10.94	68.87	9.62
	32	73.10	10.44	12.00	0.00	75.32	12.66	70.67	11.06
Qwen3-14B + RL	1	66.77	9.49	**14.00**	0.00	69.96	12.02	65.38	10.34
	4	72.78	14.87	**14.00**	2.00	74.68	18.03	70.31	15.87
	8	74.37	18.04	**14.00**	2.00	77.04	20.82	72.24	18.63
	16	75.63	21.20	**14.00**	**4.00**	78.54	23.61	73.56	21.51
	32	**76.58**	**24.68**	**14.00**	**4.00**	**79.40**	**25.97**	**74.40**	**24.16**
Llama 3.1-8B + RL	1	51.90	6.01	6.00	0.00	46.35	3.86	46.03	4.45
	4	58.86	10.76	8.00	0.00	55.15	7.94	53.73	8.53
	8	60.44	12.66	8.00	0.00	58.58	10.09	56.25	10.46
	16	62.34	14.87	8.00	0.00	59.66	11.80	57.57	12.26
	32	63.29	16.77	10.00	0.00	61.16	13.52	58.89	13.94

J.5 RL AFTER SFT

Table 14: Verification after SFT and RL with the Full reward.

Model	k	Linear / 316 compose	Linear / 316 pass	NLA / 50 compose	NLA / 50 pass	Loopy / 466 compose	Loopy / 466 pass	All / 832 compose	All / 832 pass
Qwen3-4B + SFT + RL	1	72.78	36.39	12.00	2.00	69.96	37.77	67.55	35.10
	4	79.11	51.27	14.00	4.00	79.83	48.93	75.60	47.12
	8	81.33	57.91	**16.00**	6.00	82.40	55.15	78.00	53.25
	16	81.96	61.71	**16.00**	6.00	83.48	61.37	78.85	58.17
	32	82.28	**66.77**	**16.00**	**8.00**	83.91	**65.88**	79.21	**62.74**
Qwen3-8B + SFT + RL	1	73.73	40.82	12.00	4.00	72.32	37.55	69.23	36.78
	4	80.06	56.33	12.00	6.00	80.26	53.86	76.08	51.92
	8	81.65	59.49	12.00	**8.00**	83.91	59.66	78.73	56.49
	16	**82.59**	62.66	12.00	**8.00**	84.98	62.88	79.69	59.50
	32	**82.59**	65.19	12.00	**8.00**	**85.41**	64.81	**79.93**	61.54
Qwen3-14B + SFT + RL	1	72.78	40.51	12.00	2.00	73.18	36.27	69.35	35.82
	4	77.85	53.16	14.00	**8.00**	80.26	50.21	75.36	48.80
	8	80.06	57.91	14.00	**8.00**	81.76	54.51	77.04	53.00
	16	81.65	60.44	14.00	**8.00**	83.48	57.30	78.61	55.53
	32	81.65	62.34	**16.00**	**8.00**	84.98	60.52	79.57	58.05
Llama 3.1-8B + SFT + RL	1	66.77	39.87	8.00	4.00	64.38	36.27	61.90	35.70
	4	75.63	52.53	12.00	4.00	75.97	50.21	72.00	48.32
	8	78.16	57.59	12.00	6.00	78.54	54.94	74.40	53.00
	16	79.75	60.76	12.00	6.00	78.97	59.87	75.24	56.97
	32	79.75	63.92	12.00	6.00	79.83	62.88	75.72	59.86

J.6 Reward Variants from Base

Table 15: Qwen3-8B reward ablation from Base.

Model	k	Linear / 316 compose	Linear / 316 pass	NLA / 50 compose	NLA / 50 pass	Loopy / 466 compose	Loopy / 466 pass	All / 832 compose	All / 832 pass
Qwen3-8B + RL (Binary)	1	4.75	4.75	0.00	0.00	4.72	4.72	4.45	4.45
	4	4.75	4.75	0.00	0.00	4.72	4.72	4.45	4.45
	8	4.75	4.75	0.00	0.00	4.72	4.72	4.45	4.45
	16	4.75	4.75	0.00	0.00	4.72	4.72	4.45	4.45
	32	4.75	4.75	0.00	0.00	4.72	4.72	4.45	4.45
Qwen3-8B + RL (Whole-rollout)	1	24.68	17.41	4.00	**2.00**	19.96	15.67	20.79	15.50
	4	26.27	18.67	4.00	**2.00**	20.82	16.74	21.88	16.59
	8	26.58	18.67	4.00	**2.00**	21.24	17.17	22.24	16.83
	16	26.58	19.30	4.00	**2.00**	22.32	18.24	22.84	17.67
	32	26.90	19.62	4.00	**2.00**	22.96	19.31	23.32	18.39
Qwen3-8B + RL (Clause-decomposed)	1	52.53	9.18	6.00	**2.00**	50.86	9.66	48.80	9.01
	4	61.39	17.09	8.00	**2.00**	60.52	16.52	57.69	15.87
	8	63.92	20.89	12.00	**2.00**	63.73	19.31	60.70	18.87
	16	66.77	22.78	12.00	**2.00**	67.17	20.60	63.70	20.31
	32	69.30	**24.37**	**14.00**	**2.00**	69.96	**21.89**	66.35	**21.63**
Qwen3-8B + RL	1	54.43	2.53	6.00	0.00	54.94	3.43	51.80	2.88
	4	61.71	4.75	10.00	0.00	65.02	6.44	60.46	5.41
	8	66.46	7.59	12.00	0.00	68.24	8.80	64.18	7.81
	16	71.52	9.18	12.00	0.00	73.18	10.94	68.87	9.62
	32	**73.10**	10.44	12.00	0.00	**75.32**	12.66	**70.67**	11.06

J.7 Reward Variants after SFT

Table 16: Qwen3-8B reward ablation after SFT.

Model	k	Linear / 316 compose	Linear / 316 pass	NLA / 50 compose	NLA / 50 pass	Loopy / 466 compose	Loopy / 466 pass	All / 832 compose	All / 832 pass
Qwen3-8B + SFT + RL (Binary)	1	7.59	8.54	0.00	0.00	7.51	9.87	7.09	8.77
	4	10.76	11.08	0.00	0.00	12.02	12.66	10.82	11.30
	8	13.61	13.29	0.00	0.00	14.59	14.59	13.34	13.22
	16	15.82	15.51	0.00	0.00	15.24	15.02	14.54	14.30
	32	18.04	17.72	0.00	0.00	16.31	15.67	15.99	15.50
Qwen3-8B + SFT + RL (Whole-rollout)	1	46.20	41.77	6.00	4.00	45.92	40.56	43.63	38.82
	4	53.16	48.73	8.00	6.00	52.58	47.00	50.12	45.19
	8	56.01	51.58	8.00	**8.00**	56.44	51.07	53.37	48.68
	16	57.28	53.80	8.00	**8.00**	60.30	53.43	56.01	50.84
	32	59.81	57.28	8.00	**8.00**	63.09	56.22	58.53	53.73
Qwen3-8B + SFT + RL (Clause-decomposed)	1	71.84	36.39	**12.00**	2.00	71.24	39.27	67.91	35.94
	4	80.06	51.90	**12.00**	**8.00**	79.83	55.79	75.84	51.44
	8	80.70	56.01	**12.00**	**8.00**	81.12	58.15	76.80	54.33
	16	81.01	60.13	**12.00**	**8.00**	82.40	61.37	77.64	57.69
	32	81.01	63.92	**12.00**	**8.00**	83.48	**65.24**	78.25	61.30
Qwen3-8B + SFT + RL	1	73.73	40.82	**12.00**	4.00	72.32	37.55	69.23	36.78
	4	80.06	56.33	**12.00**	6.00	80.26	53.86	76.08	51.92
	8	81.65	59.49	**12.00**	**8.00**	83.91	59.66	78.73	56.49
	16	**82.59**	62.66	**12.00**	**8.00**	84.98	62.88	79.69	59.50
	32	**82.59**	**65.19**	**12.00**	**8.00**	**85.41**	64.81	**79.93**	**61.54**

Table 17: Whole-rollout reward controls across backbones.

Model	k	Linear / 316 compose	Linear / 316 pass	NLA / 50 compose	NLA / 50 pass	Loopy / 466 compose	Loopy / 466 pass	All / 832 compose	All / 832 pass
Qwen3-4B + SFT + RL (Whole-rollout)	1	58.23	45.25	2.00	2.00	55.36	44.21	53.25	42.07
	4	65.19	53.48	4.00	2.00	64.59	54.08	61.18	50.72
	8	68.04	57.28	6.00	2.00	67.60	57.30	64.06	53.97
	16	70.57	60.44	6.00	2.00	69.74	59.87	66.23	56.61
	32	71.84	61.71	6.00	2.00	71.03	61.80	67.43	58.17
Qwen3-14B + SFT + RL (Whole-rollout)	1	67.09	50.32	2.00	0.00	55.15	40.77	56.49	41.95
	4	70.89	56.65	4.00	2.00	64.38	51.72	63.22	50.60
	8	75.95	62.03	6.00	2.00	69.10	55.36	67.91	54.69
	16	77.53	64.56	6.00	2.00	72.10	58.58	70.19	57.45
	32	**79.43**	65.51	8.00	2.00	74.03	61.80	72.12	59.62
Llama 3.1-8B + SFT + RL (Whole-rollout)	1	57.59	46.20	6.00	2.00	54.72	43.13	52.88	41.83
	4	71.20	58.23	10.00	4.00	66.95	55.58	65.14	53.49
	8	74.37	62.34	**12.00**	**8.00**	71.67	60.09	69.11	57.81
	16	76.27	66.46	**12.00**	**8.00**	74.46	65.24	71.39	62.26
	32	77.53	**70.57**	**12.00**	**8.00**	**76.82**	**68.67**	**73.20**	**65.75**

K Supplementary Experiments and Analyses

Each experiment below asks a concrete question and analyzes the corresponding results. Appendix J retains the uniform raw-data tables; the analyses here explain what the comparisons reveal.

K.1 Can Failed Responses Still Supply a Successful Proof?

Figure 6 reports the primary compose metric and standalone pass baseline for six Base models; Table 10 gives the numerical results. At $k = 1$, processing acts within one response; larger budgets also allow cross-response composition.

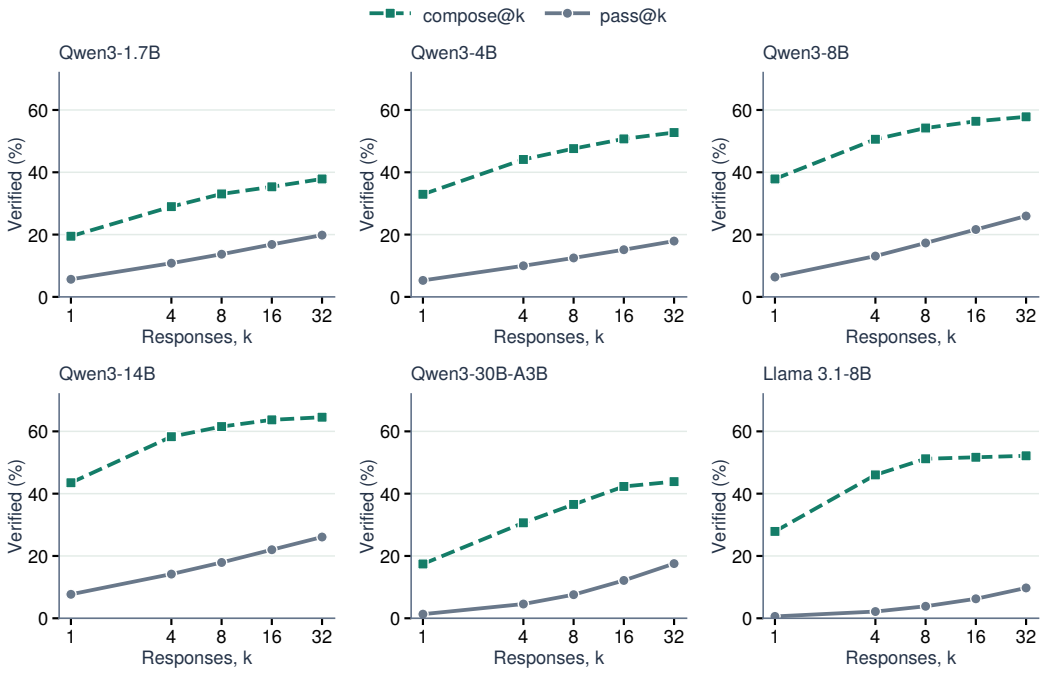

Figure 6: Compose@k and pass@k for six Base models.

Useful clauses are already recoverable at a one-response budget: Qwen3-8B reaches 37.86% compose@1 versus 6.37% pass@1, and Llama 3.1-8B reaches 27.88% versus 0.60%. Additional sampling helps, but the final budget doubling yields smaller compositional gains. From 16 to 32 responses, compose rises from 56.37% to 57.81% for Qwen3-8B and from 51.68% to 52.16% for Llama, while pass rises from 21.63% to 25.96% and from 6.25% to 9.74%, respectively. These curves show diminishing returns within the tested range.

The Qwen3-30B-A3B MoE model underperforms the smaller dense Qwen3-4B, Qwen3-8B, and Qwen3-14B models in compose@k at every tested budget. Thus its larger total parameter count does not translate into better compositional verification under this evaluation protocol.

K.2 Can Better Standalone Answers Make Composition Worse?

We compare Base, SFT (single), and composed-label SFT on the same 832 programs. The two SFT datasets contain approximately equal numbers of training examples. Single-response labels are independently filtered responses; composed labels pool discoveries across responses.

SFT (single) lowers compose over Base at every reported budget, although it improves pass. Composed-label SFT exceeds both models on both metrics: relative to SFT (single), its compose gains are 40.74 points at $k = 1$ and 30.16 at $k = 32$, with corresponding pass gains of 22.00 and 24.76 points. Thus better standalone answers need not supply better compositions; supervision from broader candidate pools improves both in this comparison. Main-text Figure 4 shows compose, and Figure 7 shows pass; per-stratum data are in Tables 11 and 12.

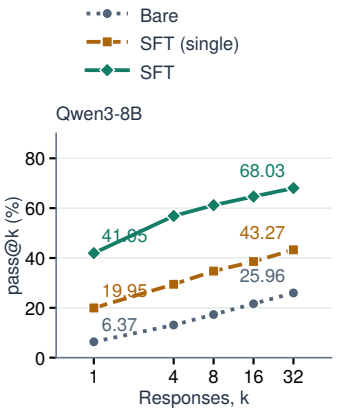

Figure 7: Effect of SFT label construction on Qwen3-8B pass@k.

K.3 Reward Ablation: Should One Bad Clause Cancel All Credit?

Reward configuration. The reported RL runs apply the per-response admission cap without penalizing clauses beyond that cap, consistently with the reward definition above.

The reward variants use different correctness gates for Equation 6. *Binary* assigns one iff the processed response is nonempty and every clause survives Houdini. *Whole-rollout* pays the strength score $\mathrm{cov}(V_i)$ only when every processed clause survives,

$$r_i^{\text{whole}} = \mathbf{1}[V_i = \mathrm{Unique}(\mathrm{First}_{m_{\max}}(\mathrm{parts}(y_i)))]\frac{|R_i|}{|\mathcal{N}|}.$$

Clause-decomposed removes this all-or-nothing indicator and scores each response by the coverage of its surviving V_i. Full (default) adds response-level Shapley credit: $r_i^{\text{Full}} = r_i^{\text{Clause-decomposed}} + 0.3\phi_i$, with the fixed survivor coverage game defined in Appendix A.6.1. All four variants admit at most $m_{\max}$ clauses per response and include no separate redundancy or overflow penalty.

Table 18: Reward ablations across backbones and initializations (%).

	compose@k					pass@k				
Reward	1	4	8	16	32	1	4	8	16	32
Qwen3-8B, RL directly from Base										
Binary	4.45	4.45	4.45	4.45	4.45	4.45	4.45	4.45	4.45	4.45
Whole-rollout	20.79	21.88	22.24	22.84	23.32	15.50	16.59	16.83	17.67	18.39
Clause-decomposed	48.80	57.69	60.70	63.70	66.35	9.01	15.87	18.87	20.31	21.63
Full (default)	51.80	60.46	64.18	68.87	70.67	2.88	5.41	7.81	9.62	11.06
Qwen3-8B, RL after SFT										
Binary	7.09	10.82	13.34	14.54	15.99	8.77	11.30	13.22	14.30	15.50
Whole-rollout	43.63	50.12	53.37	56.01	58.53	38.82	45.19	48.68	50.84	53.73
Clause-decomposed	67.91	75.84	76.80	77.64	78.25	35.94	51.44	54.33	57.69	61.30
Full (default)	69.23	76.08	78.73	79.69	79.93	36.78	51.92	56.49	59.50	61.54
Qwen3-4B, RL after SFT										
Whole-rollout	53.25	61.18	64.06	66.23	67.43	42.07	50.72	53.97	56.61	58.17
Full (default)	67.55	75.60	78.00	78.85	79.21	35.10	47.12	53.25	58.17	62.74
Qwen3-14B, RL after SFT										
Whole-rollout	56.49	63.22	67.91	70.19	72.12	41.95	50.60	54.69	57.45	59.62
Full (default)	69.35	75.36	77.04	78.61	79.57	35.82	48.80	53.00	55.53	58.05
Llama 3.1-8B, RL after SFT										
Whole-rollout	52.88	65.14	69.11	71.39	73.20	41.83	53.49	57.81	62.26	65.75
Full (default)	61.90	72.00	74.40	75.24	75.72	35.70	48.32	53.00	56.97	59.86

Preserving partial contributions improves composition. From Base Qwen3-8B, Whole-rollout lowers compose@32 from 57.81% to 23.32%, despite raising pass@1 from 6.37% to 15.50%. Clause-decomposed restores 66.35% compose@32 by crediting surviving clauses even when their response fails. Its advantage over Whole-rollout is 43.03 points from Base and 19.72 points after SFT at $k = 32$, supporting clause-level credit under both initializations. Binary reaches only 4.45% and 15.99%, respectively.

Full also exceeds Whole-rollout after SFT on Qwen3-4B and Llama 3.1-8B, by 14.30 and 9.02 points at compose@1. The separate Full versus Clause-decomposed comparison examines Shapley credit (Appendix K.9).

K.3.1 Does Training Reward Track Verification Quality?

Figure 8 reports per-step mean rewards over two RL epochs for Qwen3-8B, comparing four rewards from Base and SFT.

Binary reward saturates without strong verification. In both initializations, every recorded Binary reward in the final 20 updates is 1.000. Yet the corresponding two-epoch evaluations reach only 4.45% compose@32 from Base and 15.99% after SFT (Table 18). Saturating the correctness reward therefore does not ensure invariants strong enough to prove hidden targets.

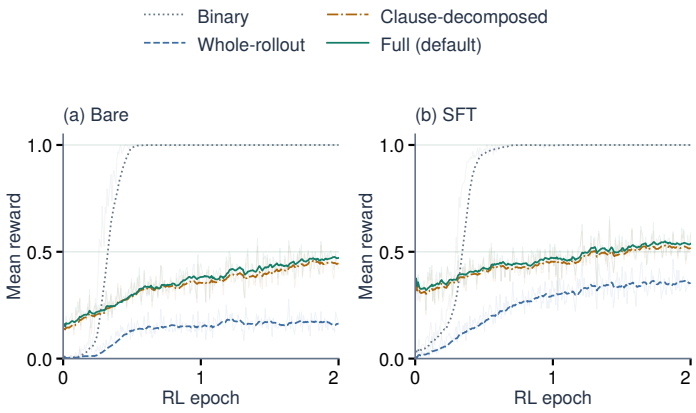

Figure 8: Training rewards from (a) Base and (b) SFT. Pale lines: per-step means; dark lines: trailing 15-step means. Reward scales differ across objectives.

Reward growth must be judged against verification. Full, Clause-decomposed, and Whole-rollout increase their training rewards under both initializations. Their different correctness gates and Shapley weights produce different reward scales; final compose results measure the resulting verification quality.

Per-group reward variance, not logged step means, is required to measure how often Full prevents group reward collapse. The mechanism is formalized in Appendix A.7.1, and additional exploration diagnostics are analyzed in Appendix K.9.

K.4 Can More Clauses Hurt Answers Yet Help Proofs?

We vary the per-response clause cap for Base, SFT, and SFT+RL Qwen3-8B. Within each model, all caps reuse the same saved response pool and evaluation settings: only the first c parsed clauses of each response are admitted to standalone verification or pooled filtering. This sweep uses a separate evaluation pool from the main results.

Table 19: Qwen3-8B clause-cap ablation on fixed response pools (%). Bold marks the best cap in each column.

	compose@1			compose@8			pass@1		
Cap	Base	SFT	SFT+RL	Base	SFT	SFT+RL	Base	SFT	SFT+RL
1	11.06	19.23	22.72	17.67	35.10	25.48	**12.14**	19.59	22.84
2	17.31	30.89	34.01	25.24	49.40	39.18	9.50	27.28	30.29
4	27.76	45.79	50.48	40.75	67.07	57.45	8.17	41.47	**45.31**
8	35.46	57.09	63.34	52.40	73.56	71.39	7.09	**44.47**	41.95
16	**39.42**	**64.90**	**69.11**	**54.21**	**76.56**	**76.32**	6.61	42.79	30.89

The best clause budget depends on the success criterion. Compose@1 and compose@8 increase monotonically with the cap for all three models. For SFT+RL, increasing the cap from 4 to 16 raises compose@1 from 50.48% to 69.11% and compose@8 from 57.45% to 76.32%, while pass@1 falls from 45.31% to 30.89%. Base pass@1 is highest at cap 1 and falls as the cap grows; SFT and SFT+RL instead peak at caps 8 and 4, respectively. Additional clauses can therefore hurt complete-response verification while expanding the candidates available for filtering.

Does Better Composition Require More Distinct Clauses? Table 20 reports mean admitted clauses per response (Clauses), distinct canonical keys across eight responses (Unique@8), and clauses retained by pooled filtering (Survivors@8).

Table 20: Mean admitted, distinct, and surviving clause counts in the clause-cap sweep.

Model	Statistic	Cap 1	Cap 2	Cap 4	Cap 8	Cap 16
Base	Clauses	1.00	1.98	3.88	7.24	12.37
	Unique@8	2.72	5.50	11.67	24.30	41.69
	Survivors@8	1.42	2.49	5.37	11.33	19.91
SFT	Clauses	1.00	2.00	3.96	7.52	12.73
	Unique@8	3.37	6.69	14.26	30.14	56.07
	Survivors@8	2.92	5.71	12.10	25.27	46.35
SFT+RL	Clauses	1.00	2.00	4.00	7.98	14.03
	Unique@8	1.41	2.90	6.41	15.90	35.80
	Survivors@8	1.18	2.33	5.06	12.06	26.07

At cap 16, Base and SFT admit similar numbers of clauses per response (12.37 versus 12.73), but SFT supplies more unique clauses (56.07 versus 41.69) and more than twice as many survivors (46.35 versus 19.91). Output length alone therefore does not explain the SFT gain.

Post-SFT RL shows a different pattern. It admits more clauses per response than SFT (14.03 versus 12.73), yet produces fewer unique clauses across eight responses (35.80 versus 56.07) and fewer survivors (26.07 versus 46.35). Nevertheless, compose@1 is 4.21 points higher, while compose@8 is nearly unchanged (76.32% versus 76.56%). Thus its one-response gain coexists with a smaller distinct candidate pool, consistent with useful candidates becoming easier to obtain at a small sampling budget, rather than increased cross-response diversity.

K.5 Does Negative Coverage Align with Final Verification?

This diagnostic evaluates negative coverage on fixed candidates. The population contains 5,711 candidate sets from 691 benchmark programs with both accepted and rejected sets, including archived target-hidden GPT-5-nano tool outputs and one verified reference set per program. These reference positives are also generated by CRAFT using GPT-5-nano and accepted by final Frama-C/WP verification. The population additionally includes 45 empty-set negatives for programs whose original candidates contain no failure. The remaining 141 programs are excluded because they lack a verified reference set or have trivial verification targets.

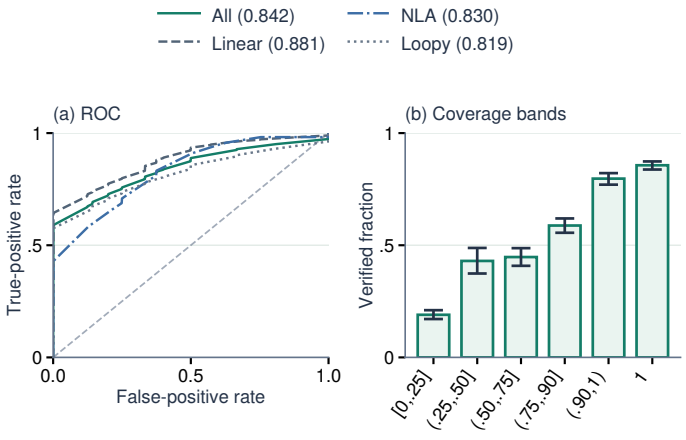

Figure 9: Negative coverage on 5,711 candidate sets. (a) Macro ROC and AUROC. (b) Verification by coverage band, with Wilson 95% intervals.

Coverage is computed from frozen Houdini survivors and positive states using the target-hidden program, seed 0, and a budget of 60 negative traces. Hidden targets supply only final Frama-C/WP labels, never scores. AUROC is computed within each program and macro-averaged. Confidence intervals use 5,000 bootstrap resamples over programs.

Table 21: Qwen3-8B filtering-order comparison: compose@k (%) and total CPU-hours.

Method	Subset	@1	@4	@8	@16	@32	CPU-h
Post-filter (300 s)	All	40.4	52.8	56.0	58.1	58.7	113
	Loopy	43.3	56.9	60.9	63.1	63.7	
	Linear	41.1	53.8	56.3	57.9	58.5	
	NLA	8.0	8.0	8.0	12.0	12.0	
Post-filter (900 s)	All	40.4	53.4	58.4	62.5	64.2	127
	Loopy	43.3	57.7	63.3	68.7	70.2	
	Linear	41.1	54.1	59.2	61.4	63.3	
	NLA	8.0	8.0	8.0	12.0	14.0	
Pre-filter (900 s)	All	40.4	52.8	57.9	62.1	64.4	1472
	Loopy	43.3	57.3	62.9	68.0	71.0	
	Linear	41.1	53.2	58.5	61.4	63.0	
	NLA	8.0	8.0	8.0	12.0	12.0	

Population	Programs	Candidates	Coverage AUROC
Linear	248	2,173	0.881
NLA	49	439	0.830
Loopy	394	3,099	0.819
All	691	5,711	0.842

Negative coverage achieves a macro AUROC of 0.842 (program-level bootstrap 95% CI: $[0.826, 0.858]$), with consistent alignment across Linear, NLA, and Loopy. Coverage thus ranks verification outcomes well on this fixed candidate population. Training-time reward diagnostics are reported in Appendix K.3.1.

K.6 When Does Filtering Together Beat Filtering Separately?

We compare filtering each response before pooling (pre-filtering) with one joint filter over the pooled clauses (post-filtering), using the same saved Qwen3-8B response prefixes on 832 programs. CRAFT uses the latter order in inference and training; the operator and its theoretical properties are defined in Appendix A.

Joint filtering approaches the same verification rate at much lower cost. With the 900-second cap, post-filtering reaches 64.2% compose@32 versus 64.4% for pre-filtering (Table 21). The recorded whole-workload cost is 127 versus 1,472 CPU-hours, or about 8.6% as much. The main observed advantage is avoiding repeated filtering calls while retaining nearly the same verification rate.

A tight timeout can erase the benefit of a larger candidate pool. At a 300-second cap, post-filtering costs 113 CPU-hours but reaches only 58.7% compose@32. Increasing the cap to 900 seconds recovers 5.5 points with 14 additional CPU-hours. Thus the cost advantage depends on allowing enough time for the larger pooled obligations; the ideal inclusion property does not guarantee higher verification under finite prover budgets.

K.7 How Does Target Visibility Change Verification and Sampling Needs?

We expose the target assertions and postconditions during generation for the same Qwen3-8B SFT+RL model, holding other settings and response budgets fixed. The model was trained with targets hidden; neither condition adds training or iterative verifier feedback. The hidden reference matches Table 14.

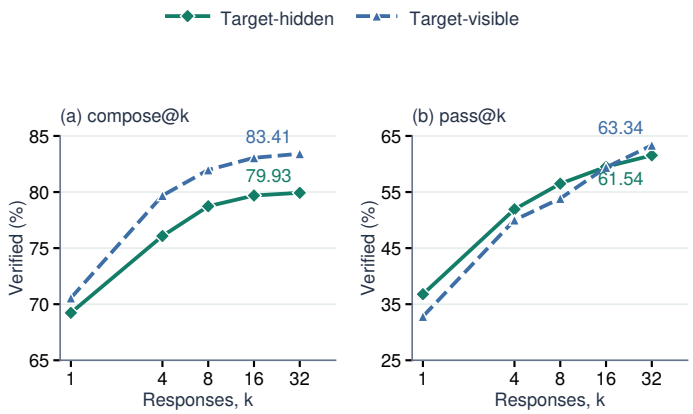

Figure 10: Target visibility for Qwen3-8B SFT+RL: (a) compose@k; (b) pass@k.

Target visibility does not remove the benefit of CRAFT's clause processing. With one independently generated response and no iterative generation refinement, filtering and composing one visible-target response reaches 70.55% compose@1, compared with pass@1 at 32.81%. At larger budgets, responses are still generated independently without verifier feedback to the model and are pooled only after generation.

Target visibility further improves whole-workload compose at every tested budget (Figure 10). Relative to target hiding, compose gains 1.32 points at $k = 1$ and 3.24–3.61 points at $k \geq 4$, corresponding to 11 and 27–30 additional verified programs. In contrast, visible-target pass is lower by 3.97 points at $k = 1$, remains lower through $k = 16$, and exceeds hidden-target pass by 1.80 points at $k = 32$. Thus visible targets may elicit stronger but less inductive clauses; because pass rejects an entire response when any clause fails, this can lower pass at small sampling budgets.

Visible compose@4 reaches 79.69%, exceeding hidden compose@8 at 78.73%. Target visibility therefore reduces the sampling budget in this evaluation.

Table 22: Target-visible and target-hidden verification with the same Qwen3-8B SFT+RL model (%).

Configuration	k	Linear / 316 compose	Linear / 316 pass	NLA / 50 compose	NLA / 50 pass	Loopy / 466 compose	Loopy / 466 pass	All / 832 compose	All / 832 pass
CRAFT (target-hidden)	1	73.73	40.82	12.00	4.00	72.32	37.55	69.23	36.78
	4	80.06	56.33	12.00	6.00	80.26	53.86	76.08	51.92
	8	81.65	59.49	12.00	8.00	83.91	59.66	78.73	56.49
	16	82.59	62.66	12.00	8.00	84.98	62.88	79.69	59.50
	32	82.59	65.19	12.00	8.00	85.41	64.81	79.93	61.54
CRAFT (target-visible)	1	74.37	35.76	14.00	4.00	74.03	33.91	70.55	32.81
	4	83.54	52.22	16.00	6.00	83.91	53.22	79.69	50.00
	8	85.44	55.38	16.00	6.00	86.70	57.94	81.97	53.85
	16	86.39	62.34	16.00	6.00	87.98	63.09	83.05	59.38
	32	86.71	65.82	16.00	6.00	88.41	67.81	83.41	63.34

The results isolate the effect of target visibility without iterative refinement.

K.8 Does the Advantage Survive Changes in Models and Tools?

We examine generator choice, verification cost with a fixed model, and how tool interfaces and feedback conditions affect the comparison.

K.8.1 Does the Best Standalone Model Also Compose Best?

Table 23 holds target hiding and reasoning-disabled generation fixed across four API models. Claude Sonnet-4.6 leads at compose@1 (56.97%), while DeepSeek V4-Flash leads at compose@8 (74.40%). At eight responses, GPT-5 has lower compose than DeepSeek (72.00% versus 74.40%) but higher pass (51.68% versus 48.20%). Choosing a generator by standalone success can therefore miss the best generator for composition.

Each model uses ten saved responses per task, with aggregation as in Appendix H. GPT-5-mini is excluded because it has no reasoning-disabled setting. Bold marks the first maximum in each metric column.

Table 23: API-model verification by program class and sampling budget, with reasoning disabled (%).

		Linear / 316		NLA / 50		Loopy / 466		All / 832	
Model	k	compose	pass	compose	pass	compose	pass	compose	pass
GPT-5-nano	1	34.81	3.80	22.00	0.00	34.55	3.86	33.89	3.61
	4	48.10	10.76	30.00	2.00	51.72	11.37	49.04	10.58
	8	55.38	15.82	44.00	4.00	56.87	17.60	55.53	16.11
GPT-5	1	52.22	35.13	36.00	6.00	52.36	33.69	51.32	32.57
	4	64.87	47.78	52.00	8.00	67.38	50.64	65.50	47.00
	8	70.25	**51.90**	66.00	**10.00**	73.82	**56.22**	72.00	**51.68**
Claude Sonnet-4.6	1	55.38	34.49	44.00	8.00	59.44	40.77	56.97	36.30
	4	62.97	40.19	52.00	8.00	66.31	47.21	64.18	42.31
	8	66.46	42.72	64.00	10.00	67.60	50.21	66.95	44.95
DeepSeek V4-Flash	1	52.22	28.80	44.00	6.00	51.72	30.04	51.44	28.13
	4	66.46	42.41	60.00	8.00	69.96	47.64	68.03	43.27
	8	**71.52**	46.52	**72.00**	8.00	**76.61**	53.65	**74.40**	48.20

K.8.2 What Verification–Cost Tradeoff Does a Fixed Model Achieve?

With GPT-5-nano and target hiding fixed, CRAFT compose@4 verifies 49.04% of programs using 1,905.10 tokens per task. Table 25 reports the tool comparison, and Figure 11 shows the end-to-end time results.

The trained Qwen3-8B model reaches 69.23%, 76.08%, and 78.73% at compose@1, @4, and @8, taking 28.22, 33.53, and 40.90 seconds per task (Table 24). At @4 and @8, it verifies more programs in less time than GPT-5-nano CRAFT (46.40 and 69.99 seconds).

Sampling parameters are listed in Table 26. Times reflect each method's native serving setup; GPT-5-nano CRAFT's time combines recorded request latency with measured filtering and final verification.

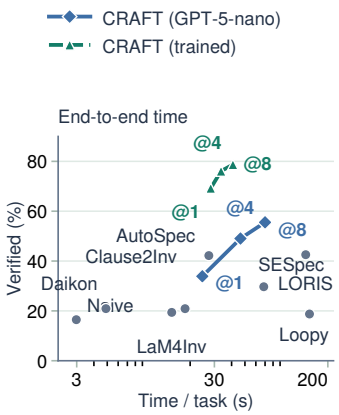

Figure 11: Verification versus end-to-end time under native serving setups.

Table 24: Verification and inference cost of trained Qwen3-8B CRAFT.

Method	Linear	NLA	Loopy	All	Tokens/task	Time/task
CRAFT (compose@1)	73.73%	12.00%	72.32%	69.23%	1,348.23	28.22 s
CRAFT (compose@4)	80.06%	12.00%	80.26%	76.08%	2,053.71	33.53 s
CRAFT (compose@8)	81.65%	12.00%	83.91%	78.73%	2,993.44	40.90 s

Table 25: Target-hidden tool comparison with GPT-5-nano and reasoning disabled. Dashes denote unsupported categories.

Method	Linear	NLA	Loopy	Supported Total	Tokens/task	Time/task
AutoSpec	47.78%	6.00%	42.27%	42.19%	2,216.11	27.34 s
SESpec	28.80%	4.00%	33.05%	29.69%	22,081.61	68.50 s
Clause2Inv	20.89%	—	—	20.89%	1,461.80	18.37 s
LORIS	42.09%	22.00%	45.06%	42.55%	47,816.27	138.17 s
LaM4Inv	22.15%	2.00%	—	19.40%	1,470.80	14.70 s
Daikon	23.10%	0.00%	21.67%	20.91%	0	4.89 s
Loopy	20.25%	0.00%	19.74%	18.75%	14,513.37	147.16 s
Naive	15.19%	2.00%	18.88%	16.47%	225.95	2.98 s
CRAFT (compose@1)	34.81%	22.00%	34.55%	33.89%	1,136.82	24.52 s
CRAFT (compose@4)	48.10%	30.00%	51.72%	49.04%	1,905.10	46.40 s
CRAFT (compose@8)	55.38%	44.00%	56.87%	55.53%	2,929.47	69.99 s
CRAFT (pass@1)	2.53%	0.00%	4.72%	3.61%	1,136.82	7.86 s

Table 26: Sampling parameters for the tool comparison.

Method	Sampling budget / task	Temperature / top-p	Completion cap	Reasoning
AutoSpec	8	1.0 / 1.0	8,192	none
SESpec	1 initial + up to 3 refinements per phase	1.0 / 1.0	8,192	none
Clause2Inv	adaptive; 1 per call	1.0 / 1.0	8,192	none
LORIS	1 initial + adaptive local-proof repairs	1.0 / 1.0	8,192	none
LaM4Inv	5 initial + adaptive VC repairs	1.0 / 1.0	8,192	none
Daikon	no model call	–	–	–
Naive	1	1.0 / 1.0	8,192	none
CRAFT	1, 4, or 8	1.0 / 1.0	8,192	none
Loopy	15 initial + up to 7 repairs	0.7 / 1.0	8,192	none

Cost means cover each method's supported set where measurements are available; verification retains every supported task. AutoSpec's token mean uses 674 exact-usage tasks, excluding 154 tokenizer-estimated and four missing-usage records. Its eight-choice request counts the prompt once and all completions. Four timed-out LORIS runs lack final usage records but remain verification failures. CRAFT uses the same ten-response pools as Table 23.

K.8.3 Tool-by-Tool Analysis

The direct comparisons use GPT-5-nano with reasoning disabled and a common restored-target final judge. Target/postcondition feedback is removed from Clause2Inv, LaM4Inv, and LORIS; only target-independent initiation and preservation feedback is retained. Each method uses its full supported set; unsupported categories are excluded rather than counted as failures.

Direct invariant-strength comparison. We also compare target-free, inductively filtered invariants by checking implication in both directions under C integer bounds. On the 316 Linear programs supported by all eight baselines (the seven tools above plus Naive), all pairwise comparisons are resolved for 279 programs. With GPT-5-nano and reasoning disabled, CRAFT compose@8 is strictly stronger than at least six baselines on 186 of these programs and than all eight on 58. A baseline with no surviving invariant counts as beaten only when CRAFT has a nontrivial one.

AutoSpec. AutoSpec pools eight parallel proposals in its native workflow, reaching 42.19% verification with 2,216.11 tokens per task. CRAFT compose@4 reaches 49.04% with 1,905.10 tokens. Both methods exploit multiple proposals; CRAFT's clause-processing pipeline achieves higher verification with fewer tokens.

SESpec. SESpec uses verifier feedback to iteratively repair a running candidate. Under target hiding, it verifies 29.69% with 22,081.61 tokens per task, versus 49.04% with 1,905.10 tokens for CRAFT compose@4. With limited informative feedback, parallel sampling and clause composition are more effective than LLM repair, verifying more programs with fewer tokens.

Clause2Inv. Clause2Inv normally derives false-state counterexamples from the postcondition (Cao et al., 2025b), a feedback source unavailable under target hiding. Its released VC interface supports the 316 Linear programs, on which it verifies 20.89%, compared with 48.10% for CRAFT compose@4.

LORIS. LORIS repairs model reasoning for failed target-facing verification conditions in its native workflow (Li et al., 2026a). Under target hiding, it verifies 42.55% on the full benchmark with 47,816.27 tokens per task, compared with 49.04% and 1,905.10 tokens for CRAFT compose@4. Local-proof repair thus incurs substantially more token use in this feedback regime.

LaM4Inv. LaM4Inv uses postcondition verification conditions to guide native refinement (Wu et al., 2024a). Its required CFG/SMT artifacts are unavailable for Loopy, so evaluation covers 316 Linear and 50 NLA programs. Without target feedback, it verifies 19.40% on this supported set using 1,470.80 tokens per task; CRAFT compose@4 verifies 45.63% on the same set.

Loopy. Loopy follows a similar high-level strategy to AutoSpec: generating multiple candidate invariants and using Houdini to filter them. Loopy's initial responses contain invariant annotations in 98.8% of cases, but its native extractor passes only 1.75% of initial responses to Houdini. Generating annotations therefore does not ensure that they reach the verifier.

Daikon. Daikon provides a non-LLM reference, verifying 20.91% without model calls and averaging 4.89 seconds per task. It offers low-cost verification, while CRAFT's sampled candidates extend coverage beyond this baseline.

Naive. The single-response Naive baseline verifies 16.47% with 225.95 tokens per task. It supplies a low-cost reference without CRAFT's clause-processing pipeline; CRAFT compose@1 reaches 33.89% at higher token cost.

Quokka. Quokka evaluates 866 SV-COMP instances, fine-tunes on 3,589 separate training instances, and directly tests whether a generated invariant helps prove the visible target assertion (Wei et al., 2026). It studies verification acceleration with visible targets, whereas CRAFT evaluates target-hidden invariant generation with a common Frama-C backend. These differences in task and benchmark preclude a direct numerical comparison.

K.9 Shapley Analysis: Does Group Credit Help Exploration?

We compare Full with Clause-decomposed on Qwen3-8B to examine when added group credit helps from Base versus SFT after 2 RL epochs.

At equal coverage, Full gives more credit to traces rejected by fewer responses, encouraging complementary effective coverage. The allocation and its Shapley equivalence are defined in Appendix A.6.1.

Table 27 compares Clause-decomposed and Full on Qwen3-8B after 2 RL epochs, with target-hidden negative-trace construction and 9,024 training programs. Full after SFT uses the main evaluation at 69.23% compose@1, consistently with Tables 14 and 18.

Table 27: Clause-decomposed versus Full after two RL epochs on Qwen3-8B (%). Bold marks the better reward within each initialization.

	compose@k					pass@k				
Reward	1	4	8	16	32	1	4	8	16	32
RL from Base										
Clause-decomposed	48.80	57.69	60.70	63.70	66.35	**9.01**	**15.87**	**18.87**	**20.31**	**21.63**
Full (default)	**51.80**	**60.46**	**64.18**	**68.87**	**70.67**	2.88	5.41	7.81	9.62	11.06
RL after SFT										
Clause-decomposed	67.91	75.84	76.80	77.64	78.25	35.94	51.44	54.33	57.69	61.30
Full (default)	**69.23**	**76.08**	**78.73**	**79.69**	**79.93**	**36.78**	**51.92**	**56.49**	**59.50**	**61.54**

From Base, Full improves composition throughout the tested budgets. Full improves compose by 2.77–5.17 points across the tested budgets, including 48.80% to 51.80% at $k = 1$ and 66.35% to 70.67% at $k = 32$. Its advantage remains substantial at the largest measured budget, while pass decreases by 6.13–11.06 points: better composition can accompany less successful complete responses.

After SFT, the added benefit is smaller. Full improves compose by 0.24–2.05 points across the tested budgets: 67.91% to 69.23% at $k = 1$, and 78.25% to 79.93% at $k = 32$. Pass also improves by 0.24–2.16 points. The added composition benefit after SFT is smaller than from Base, but remains positive at every tested budget.

Does Full retain more policy entropy? The training diagnostics compare Full and Clause-decomposed over 2 RL epochs. Table 28 reports an early reference measurement and the end-of-training value. Retention is final entropy divided by the early reference value.

Table 28: Policy entropy over two RL epochs. Bold marks the higher final entropy and retention within each initialization.

Initialization	Reward	Early entropy	Final entropy	Retention (%)
SFT	Full (default)	0.3195	**0.1842**	**57.7**
	Clause-decomposed	0.3233	0.1323	40.9
Base	Full (default)	0.0588	**0.0130**	**22.1**
	Clause-decomposed	0.0773	0.0120	15.5

Full retains more policy entropy under both initializations. The absolute final-entropy difference is larger after SFT (0.1842 versus 0.1323) than from Base (0.0130 versus 0.0120); Base also starts from unequal reference entropies. These observations suggest that Full slows entropy loss during training.

Reward dispersion and policy entropy. The training summary reports a batch mean-to-maximum reward ratio 0.01–0.06 lower for Full than for Clause-decomposed.

Full improves compose@k from Base and retains higher policy entropy than Clause-decomposed. The Shapley mechanism and its limits are formalized in Appendix A.7.1.

K.10 How Does Reward Design Affect Sharpening Across Sampling Budgets?

We compare sharpening under two reward designs: Whole-rollout with pass@k, and Full with compose@k. Figure 12 shows Qwen3-8B before and after two RL epochs on 832 programs for Whole-rollout from Base and SFT, and Full after SFT, together with gains over each initialization at the same sampling budget. Full from Base is shown in Figure 3.

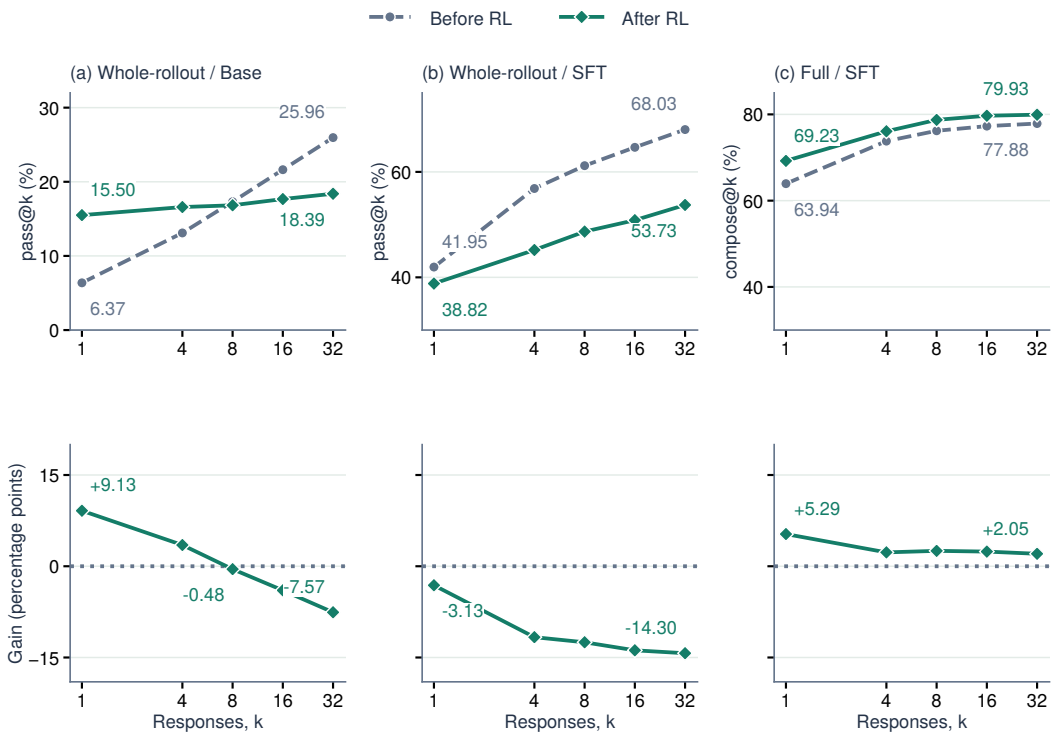

Figure 12: Qwen3-8B verification before and after RL (top) and gains over the corresponding initialization (bottom): Whole-rollout from (a) Base and (b) SFT, measured by pass@k; (c) Full after SFT, measured by compose@k. Top panels use different scales; bottom panels share a scale, with dotted lines marking zero gain. Whole-rollout falls below Base at $k = 8$ and below SFT at every tested budget, while Full retains positive compose gains through $k = 32$ after SFT.

Whole-rollout exhibits an early crossing in standalone success. From Base, Whole-rollout raises pass@1 from 6.37% to 15.50% but lowers pass@32 from 25.96% to 18.39%. Its gain becomes negative at $k = 8$, the first tested budget where the trained curve falls below Base. This crossing is consistent with sharpening: complete successful responses become more frequent at one sample, while fewer tasks are solved within 32 samples. After SFT, Whole-rollout lowers pass@1 from 41.95% to 38.82% and pass@32 from 68.03% to 53.73%. Its gain is negative at every tested budget, worsening from -3.13 to -14.30 points. Thus, the characteristic increase in pass@1 alongside a decrease in pass@32 occurs from Base; after SFT, Whole-rollout degrades standalone verification throughout the tested range, with a larger loss at higher budgets.

Full preserves gains at larger composition budgets. After SFT, Full improves compose@1 by 5.29 points and compose@32 by 2.05 points. The larger gain at one sample is consistent with useful clauses becoming available earlier, although the gain does not decrease strictly monotonically with k. From Base, the respective gains are 13.94 and 12.86 points, so the largest measured budget retains most of the improvement. Thus, a sharpening pattern is weaker for composition and is clearest after SFT. Full's compose gain remains positive at every tested budget. SFT already reaches 77.88% compose@32, which also leaves less room for improvement at that budget.

Partial credit can preserve useful variation in GRPO rewards. A plausible explanation is that Full rewards surviving clauses from otherwise unsuccessful responses. Whole-rollout's gate instead removes the coverage reward whenever any processed clause fails, potentially assigning zero to every response in a GRPO group. Full removes this source of reward sparsity and can distinguish responses through the coverage of their surviving clauses. Its Shapley term can also distinguish responses

with equal coverage but different complementary contributions. By reducing reward sparsity and providing finer reward distinctions within each GRPO group, Full can help mitigate sharpening.

L Detailed Limitations

Conjunctive composition. CRAFT combines retained clauses by conjunction and does not search for disjunctions among candidates. A disjunction can be inductive even when its individual disjuncts are not, so this restriction can miss useful invariants that describe alternative execution modes. However, $c_1 \vee c_2$ is logically weaker than either c_1 or c_2: it admits more states and can lose constraints needed to prove a target. Disjunctive combinations also introduce Boolean alternatives that can increase case splitting and SMT solving cost, especially alongside arithmetic constraints. Our composition therefore focuses on strengthening summaries through conjunction; exploring disjunctions while controlling precision and verification cost remains future work.

Program scope. Our implementation targets numerical programs with one scalar-integer loop. The evaluation excludes nested loops and invariants over arrays, heaps, or recursively defined predicates. The negative-trace construction probes scalar arithmetic relations and exit boundaries (Liu et al., 2024).

Nonlinear arithmetic. Most NLA programs remain unsolved by the evaluated local Qwen3 models, even after SFT and RL. This suggests that invariants sufficient for these programs largely lie beyond the candidates these models can access within the tested sampling budgets. Composition can recover useful generated clauses, but cannot supply nonlinear relations absent from the candidate pool.

Conditional reachability of negative traces. Nondeterminism, truncated or sparsely recorded traces, and unrecorded state affecting execution can prevent cleaning from establishing unreachability. The sampler uses sparse recording for long runs and calls no general unreachability solver. Coverage is therefore an empirical strength signal; unknown evaluations count as non-rejection and may underestimate it. Final soundness is established independently by Frama-C/WP.

Inference engine. Local base and trained models are served with vLLM. In preliminary comparisons under nominally matched decoding settings, these models had lower verification rates with vLLM than with the other evaluated backends. API models retain their provider serving interfaces and chat templates. The reported decoding and reasoning controls are applied to each backend.

Verifier and specification language. Our filtering and all final judgments use Frama-C/WP, and candidates use ACSL. Parser behavior, generated obligations, prover integration, and syntax affect which clauses survive. The reported results characterize this verifier, specification language, and source language.

Finite candidate-pool approximation. Before Houdini, the pipeline implementation applies a lightweight AST-based filter to parse, normalize, and deduplicate candidate clauses. Clauses outside its supported ACSL subset are omitted. Each response contributes at most $m_{\max}$ clauses before global pooling; later clauses in that response are truncated, while no separate cap is applied to the pooled union.

Feedback regime. The evaluation covers target-independent training and finite-budget composition without iterative verifier-feedback optimization. In pilot runs, models reached inductive but weak invariants: the invariants passed induction, while the hidden goals left the verifier without a target-directed signal for the missing behavioral strength. The marginal gain from refinement decreased as the initial rollout pool grew.

Training variance. We did not repeat training and inference multiple times for every experimental configuration.