When Do RTL Verifiers Lie? Measuring Proxy-Reward Overoptimization in LLM Hardware Design

Anonymous authors
Paper under double-blind review

Abstract

Reinforcement learning with verifiable rewards (RLVR) and inference-time search are increasingly used to improve generated register-transfer-level (RTL) designs. While these techniques require access to rigorous test scripts, high-quality RTL data, and verification infrastructure, these resources are scarce and often proprietary. Recent pipelines therefore rely on capable teacher models to generate synthetic RTL and automated testbenches. However, optimizing against these scalable verification proxies risks Goodhart effects: visible reward may improve without a corresponding gain in true functional correctness. This risk is especially acute because RTL correctness spans input sequences and clock cycles, whereas teacher-generated testbenches, randomized differential simulation, and bounded model checking (BMC) provide scalable but finite checks, with the fidelity of simulation and bounded checking depending on verification effort.

We compare four testbench conditions generated by Grok 4.6 and DeepSeek V4 Flash from either the specification or golden RTL, alongside differential simulation and BMC. A hidden formal-equivalence Oracle evaluates candidate RTL against its golden reference. The study covers 100 tasks, eight Qwen3 configurations, 50,948 candidates, Best-of-N search, multi-turn repair, and two-epoch RL. Qwen3-32B reasoning reaches 82% Oracle Pass@64, but teacher-generated testbenches select proved designs on only 61–66% of tasks despite visible success rates of up to 93%; tuned BMC and differential simulation reach 78% and 76%, respectively. Increasing verification effort narrows this gap, while weak direct-mode repair feedback can continue improving visible success after hidden progress stalls. In the Qwen3-4B RL run, the proxy–Oracle gap widens and rewarded formal counterexamples become more frequent without a clear improvement in Oracle proof rate; the 8B run shows a persistent proxy–Oracle discrepancy over the same training budget.

1 Introduction

Large language models (LLMs) support hardware design (Liu et al., 2023b), including register-transfer-level (RTL) generation (Liu et al., 2023d) and repair (Tsai et al., 2024). RTL describes a digital circuit's behavior in terms of its inputs and state.[1] Inference-time search selects among sampled designs using a verifier; reinforcement learning with verifiable rewards (RLVR) uses an executable check for training. Both can scale without manual labels.

Constructing a rigorous verifier for every RTL task is expensive, so pipelines use teacher-generated testbenches as scalable rewards for repair or RL, as in ChipSeek (Chen et al., 2025). Their finite tests must cover enough inputs and temporal behavior to reject incorrect RTL as optimization proceeds.

Testbenches use selected inputs, differential simulation samples traces, and bounded model checking (BMC) searches to a configured depth. Best-of-N, repair, and RL increase exposure to their blind spots: a higher proxy score need not mean higher functional correctness (Gao et al., 2023; Helff et al., 2026).

[1] Appendix A and C introduce the reader to RTL description and verification.

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.

We compare exposed rewards with a hidden formal-equivalence Oracle, except in an Oracle-feedback repair control. Four testbench conditions cross Grok 4.6 (SpaceXAI, 2026) and DeepSeek V4 Flash (DeepSeek-AI, 2026) with specification or golden RTL prompts; retained benches pass a golden self-check. We also vary GoldDiff simulation length and BMC depth. A 100-task, 50,948-candidate Qwen3 baseline supports Best-of-N search; repair and RL test adaptive optimization.

Our contributions are:

- We introduce a controlled framework for studying RTL rewards under Best-of-N search pressure, iterative repair, and policy optimization. The optimized reward is separated from a hidden formal-equivalence oracle.
- We show that scale and reasoning both enlarge the pool of formally correct RTL candidates across eight Qwen3 configurations and 50,948 designs, and we measure how much of that pool each proxy recovers.
- We show how BMC depth and randomized-simulation length change false positives, selection regret, hidden correctness, and verification cost, identifying when extra checker effort improves reward reliability.
- We show that reward-quality differences persist under multi-turn repair and extend the audit to RLVR training. At RL checkpoints, testbench reward increasingly disagrees with formal outcomes in the Qwen3-4B-tagged run; a Qwen3-8B-tagged run provides a second observed trajectory.

2 Related Work

LLMs for code and hardware design. Execution-based evaluation gives code generators scalable feedback (Chen et al., 2021; Liu et al., 2023a). LLMs now generate, debug, and benchmark Verilog RTL (Thakur et al., 2023a; Liu et al., 2023c; Lu et al., 2024; Pinckney et al., 2025). AutoChip uses simulation feedback to revise generated RTL (Thakur et al., 2023b). Finite tests may miss state, reset, and long temporal behavior, so we study rewards for selection and improvement alongside one-shot accuracy.

Inference-time search and RL with verifiable rewards. Repeated sampling and generated tests guide code selection (Brown et al., 2024; Chen et al., 2023); execution feedback supports self-debugging and RL for code and reasoning (Chen et al., 2024; Le et al., 2022; Shao et al., 2024; DeepSeek-AI et al., 2025). RTL post-training pipelines use executable testbenches as rewards, but obtain them in different ways: teacher generation, rule-based generation from a golden reference, or curation from task data (Chen et al., 2025; Zhu et al., 2025; Teng et al., 2025; Wang et al., 2026b). AutoBench and CorrectBench generate and assess LLM-written hardware testbenches (Qiu et al., 2024; 2025). STG reports higher coverage and fewer false passes than an LLM-generated testbench (Huang et al., 2026). Here "verifiable" means executable, not a proof of complete functional correctness. We test reward alignment under search, repair, and RL checkpoints.

Reward overoptimization. Goodhart's law, reward hacking, and specification gaming describe failures in which optimization improves a measured objective without improving the underlying goal (Gao et al., 2023; Pan et al., 2022; Skalse et al., 2022). RLVR models can exploit checks that verifiers omit (Helff et al., 2026). Public and hidden tests expose a related gap in code generation and coding agents (Liu et al., 2023a; Jain et al., 2024; Zhao et al., 2026). RTL lets us vary verifier effort and check selected candidates with a formal oracle.

RTL verification. Simulation checks chosen traces; BMC searches for counterexamples within a bound. Formal equivalence gives stronger evidence subject to circuit and reset models, supported constructs, and solver limits. We study these methods as optimization rewards (McKeeman, 1998; Biere et al., 1999; Bradley, 2011). NotSoTiny and RTL-BenchLS benchmark with formal equivalence (Ghorab et al., 2025; Wang et al., 2026a); FormalRTL uses it during generation (Li et al., 2026).

3 PROBLEM FORMULATION

3.1 CANDIDATES AND VERIFIER OUTCOMES

We study M RTL tasks x_m, each with a specification and golden design. For each task, a generator attempts samples $y_{m,1}, \ldots, y_{m,N}$ in order; some attempts do not yield extractable RTL. A proxy verifier j scores each available candidate. The three main verifier families use binary scores on usable verdicts:

$$r_{\mathrm{TB},j}(x, y) = \mathbb{I}[\mathrm{TB}_{x,j}(y) = \texttt{PASS}], \tag{1}$$

$$r_{\mathrm{GoldDiff}}(x, y) = \mathbb{I}[\mathrm{GoldDiff}_x(y) = \texttt{PASS}], \tag{2}$$

$$r_{\mathrm{BMC}}(x, y) = \mathbb{I}[\mathrm{BMC}_x(y) = \texttt{NO_CEX_IN_BOUND}], \tag{3}$$

$$r_{\mathrm{Syntax}}(x, y) = \mathbb{I}[\mathrm{Syntax}(y) = \texttt{PASS}]. \tag{4}$$

Each of the four teacher and prompt-source combinations defines a separate frozen $\mathrm{TB}_{x,j}$. A testbench pass requires candidate compilation, simulation, and an accepted success marker; a GoldDiff pass means its finite simulated traces found no mismatch; a BMC pass means it found no counterexample through its configured bound. Syntax checks parsing and elaboration only. Each verifier effort setting defines its own j; a missing or unusable verdict is not a pass.

The Oracle checks the candidate against the golden design. It is hidden from proxy selection and testbench-reward RL; one repair control explicitly uses Oracle feedback. Define $h(x, y) = 1$ only when it returns `PROVED_EQUIVALENT`; otherwise set $h(x, y) = 0$ for fixed-denominator rates. A counterexample establishes a functional mismatch under the Oracle model. A timeout, unsupported case, or tool failure remains unresolved, even though it contributes zero to these conservative rates. Candidate-level agreement statistics use only proofs and counterexamples.

3.2 SEARCH METRICS

Best-of-N chooses the highest-scoring candidate among the first N sample slots with a usable proxy verdict:

$$\hat{y}^{(N)}_{m,j} = \arg \max_{i \le N:\, r_j(x_m, y_{m,i}) \text{ valid}} r_j(x_m, y_{m,i}). \tag{5}$$

Ties go to the lowest sample index. If there is no usable verdict, there is no winner, and the task contributes zero to the fixed-denominator rates.

Oracle Pass@N is the fraction of tasks with at least one *proved* candidate in the first N slots. Hidden BoN@N is the fraction for which the proxy selects a proved candidate:

$$\mathrm{Pass@}N = \frac{1}{M} \sum_m \max_{i \le N} h(x_m, y_{m,i}), \tag{6}$$

$$\mathrm{BoN}_j@N = \frac{1}{M} \sum_m h(x_m, \hat{y}^{(N)}_{m,j}). \tag{7}$$

Missing samples and absent winners contribute zero. Their difference, $R_j@N = \mathrm{Pass@}N - \mathrm{BoN}_j@N$, is *selection regret*: a proved candidate was available but the selected one was not proved. Pass@N is a ceiling for *proved* selection on this pool, not necessarily for actual correctness when Oracle outcomes are unresolved.

For binary proxies, let $V_j@N$ be the fraction of all M tasks whose selected candidate passes the proxy; tasks without a winner contribute zero. We report the matched-denominator proxy–proof gap $G_j@N = V_j@N - \mathrm{BoN}_j@N$. This gap compares aggregate rates. It does not count individual false positives: false negatives can offset them, and an unresolved Oracle result does not prove a bug.

3.3 ADAPTIVE OPTIMIZATION

In iterative repair, $y^{(t)}_{m,j}$ is the design after feedback turn t. Alongside correctness at each turn, we report the fraction of tasks with a proof at any turn through t:

$$C^{(t)}_j = \frac{1}{M} \sum_m \max_{0 \le s \le t} h(x_m, y^{(s)}_{m,j}). \tag{8}$$

In RL, GRPO updates the policy using only the binary DeepSeek testbench reward on training tasks. At each checkpoint we score the same held-out tasks with the frozen testbench and Oracle. Unlike the task-level search metrics, checkpoint rates average over generated outputs. We report testbench pass, Oracle proof, their difference, and the share of outputs that pass the testbench but receive a formal counterexample. An increasing proxy–proof gap shows growing reward disagreement; the counterexample share directly measures rewarded incorrect outputs. Neither measure alone implies that the overall Oracle proof rate fell.

4 Experimental Setup

4.1 Tasks and generators

We use the held-out `rl_val` split of the CodeV-R1 dataset (Zhu et al., 2025). Our baseline covers 100 tasks, eight Qwen3 configurations (Yang et al., 2025), and 50,948 extracted candidates; preparation and sampling settings are in Appendix B.

4.2 LLM-generated testbench conditions

Teacher-generated testbenches model a common RTL reward pipeline: a strong model writes a checker that a generator or policy tries to pass. We use Cursor Grok 4.6 at extra-high reasoning effort and open-weight DeepSeek V4 Flash at high reasoning effort (DeepSeek-AI, 2026).

For each teacher, we construct two prompt-source regimes. The *specification* condition receives the natural-language specification and module ports but not the golden RTL. The *golden* condition receives the golden RTL and ports but not the natural-language specification. This gives the four conditions in Appendix Table 3.

Retained benches pass a golden-RTL self-check; see Appendix B.

4.3 Verification ladder

We compare teacher-generated testbenches, GoldDiff, and BMC against the hidden formal Oracle. Tuned comparisons use GoldDiff at 10^5 nominal steps and BMC at sequential bound $k = 8$. Exact pass meanings, effort tiers, and Oracle configuration appear in Appendix L and Appendix B.

4.4 Optimization regimes

Best-of-N. We evaluate fixed candidate prefixes at $N \in \{1, 2, 4, 8, 16, 32, 64\}$ on the same candidate pool. Score ties select the earliest sample; unusable verdicts provide no selection score. Task-level Pass@N and BoN@N use the fixed denominator of 100 tasks.

Iterative repair. Qwen3-32B direct-mode repair starts from turn 0 and runs ten feedback turns on 100 tasks. Six proxy conditions complete all trajectories; the Oracle feedback control completes 99. Every available turn is scored by Oracle. The control exposes Oracle itself, so public and hidden success coincide on labeled turns. The `sliding-window-v1` protocol is detailed in Appendix B. We also evaluate reasoning-mode repair under the same seven conditions, but only 82–90 trajectories complete per condition; its outcomes and coverage are in Appendix J.1.

Policy optimization. We compare GRPO-based policies (Shao et al., 2024) at a fixed budget of two epochs over 815 training tasks, using a DeepSeek-generated testbench reward. With batch size 16, each epoch corresponds to 50 optimizer steps; both reported trajectories therefore end at step 100. The main run is tagged Qwen3-4B and uses 16 rollouts per training prompt; the supporting Qwen3-8B-tagged run uses eight. Both have joint proxy and Oracle validation every five steps on the same 100-task `rl_val` split, with five sampled outputs per task plus four additional logged outputs per checkpoint.

We generated the task-specific training testbenches ourselves by prompting DeepSeek V4 Flash with each task specification and interface, retaining a bench only after it compiled and passed on the golden RTL. We froze these benches before RL. For generated RTL y on task x, the scalar training

reward is $r_{\mathrm{TB}}(x, y) = \mathbb{I}[\mathrm{TB}_x(y) = \mathrm{PASS}]$. A pass requires RTL extraction, compilation with the frozen bench, execution, and an accepted success marker; every other outcome receives zero. The reward has no partial-credit terms. Oracle proof and counterexamples are logged for evaluation, never returned as the GRPO reward. Appendix B.3 gives the available training settings.

4.5 Oracle outcomes and statistics

Only an Oracle proof counts as success in fixed-denominator rates; unresolved outcomes remain separate from counterexamples. Coverage counts and resampling methods appear in Appendices B, H, and Q; Appendix Figure 4 shows the full study workflow.

5 Results

5.1 Scale and reasoning improve the candidate pool

Figure 1 summarizes how scale, reasoning, and search change Oracle Pass@N (tasks with at least one proof among the first N samples). Appendix Table 5 reports exact rates and proved-candidate density (proved samples divided by all samples). Full Pass@N curves are in Appendix Table 10.

Reasoning can substitute for a large increase in size. A sharper comparison is across modes. Qwen3-4B reasoning matches Qwen3-32B direct at both Pass@1 (14%) and Pass@64 (45%), and the two curves remain close at the intermediate budgets. Qwen3-8B reasoning exceeds 32B direct at every reported N, and 14B reasoning exceeds it by a wide margin (20% versus 14% at Pass@1; 61% versus 45% at Pass@64). In this family, enabling reasoning is a larger capability intervention than moving from 4B to 32B in direct mode.

Why this pool is the right test of proxy rewards. On a weak candidate pool, a selector cannot recover a solution that was never generated. The 4B and 8B direct pools have only a 20% Pass@64 ceiling; even a perfect proxy would look weak there. The 32B reasoning pool creates a 49-point gap between one-shot performance and the oracle ceiling. That gap is the clearest setting for measuring whether a proxy identifies the available correct design, which is the question of the next subsections.

Additional one-shot, sampling, proof-density, coverage, and cross-model selection results appear in Appendix F.

5.2 Weak proxies leave correct designs unselected

Table 1 separates visible success from hidden selection quality on the Qwen3-32B reasoning pool. Tuned BMC selects an oracle-correct candidate for 78% of tasks, leaving 4 points of regret relative to the 82% ceiling. GoldDiff reaches 76%, leaving 6 points. In contrast, LLM-generated testbench variants recover 61–66% and leave 16–21 points of regret. Syntax recovers only 34%. Against the specification-prompted DeepSeek testbench, BMC selects 14 more formally proved designs on the same 100 tasks (95% paired task-bootstrap interval: 7–22); GoldDiff selects 12 more (interval: 4–20). These intervals capture which tasks are sampled, conditional on the frozen candidate pool and its fixed selection order.

Visible proxy pass can give a different impression. On the same fixed 100-task denominator, Grok-TB Spec reports 93% visible success but 61% hidden correctness. DeepSeek-TB Spec reports 86% visible success but 64% hidden correctness. Their proxy–proof gaps are 32 and 22 points, respectively. Tuned GoldDiff and BMC each have an 8-point fixed-denominator gap.

The teacher-family and prompt-source contrasts and candidate-level verifier diagnostics are in Appendix F.

Tuned BMC and GoldDiff recover most proved designs available in this pool; testbench rewards leave substantially more capability unselected.

Full effort tables, candidate-level inclusion rules, and verifier diagnostics appear in Appendix F.

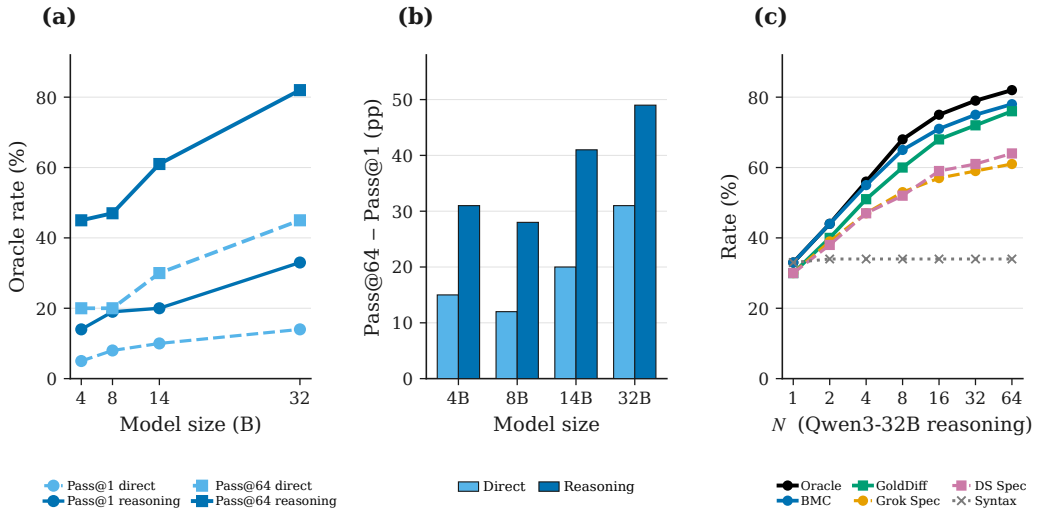


Figure 1: **Scale, reasoning, and search pressure.** Larger models and reasoning mode raise both one-shot correctness and the density of proved candidates. Reasoning at 4B matches 32B direct, while 32B reasoning reaches an 82% Pass@64 ceiling. The search gain is larger for every reasoning configuration, but the amount recovered from that stronger pool still depends on the selection proxy.

Table 1: **Best-of-64 selection on Qwen3-32B reasoning.** Visible, hidden, gap, and regret use the same fixed 100-task denominator. Precision and FPR are candidate-level metrics on binary Oracle labels.

Selector	Visible	Hidden	Gap	Regret	Precision	FPR
Syntax	100%	34%	66 pp	48 pp	41.4%	96.2%
Grok-TB Spec	93%	61%	32 pp	21 pp	66.1%	32.6%
Grok-TB Golden	84%	66%	18 pp	16 pp	80.5%	15.2%
DeepSeek-TB Spec	86%	64%	22 pp	18 pp	73.5%	22.9%
DeepSeek-TB Golden	84%	64%	20 pp	18 pp	74.6%	21.1%
BMC ($k = 8$)	86%	78%	8 pp	4 pp	91.4%	6.65%
GoldDiff (10^5 nominal)	84%	76%	8 pp	6 pp	97.0%	1.77%

5.3 Verification effort controls reward reliability

Figure 2 shows how BMC improves as the bound increases. Hidden BoN@64 rises from 61% at $k = 1$ and 66% at $k = 2$ to 77% at $k = 4$ and 78% at $k = 8$. Increasing the bound to 16 or 32 produces 77% and 76%, respectively, showing no additional gain in this fixed pool. The candidate-level FPR also changes substantially: reducing the bound from $k = 8$ to $k = 2$ increases FPR from 6.7% to 37.0%.

GoldDiff shows a similar curve. Hidden BoN@64 rises from 58% at 10 cycles to 72% at 100 cycles, 74% at 10^3, and 76% at 10^4 and 10^5 cycles. The value is 75% at $5{\times}10^5$ cycles. Small non-monotonic changes are expected because each setting selects one candidate per task from a fixed pool. The broad result is stable: very short traces leave large regret, moderate effort captures most of the available gain, and the highest efforts show diminishing returns.

Appendix Figure 8 shows how every effort tier behaves across the full Best-of-N range.

The effort study gives a practical conclusion. Choosing the lowest-cost check can waste generation compute because it selects the wrong candidate. At the same time, running the strongest setting on every candidate can add cost with little gain after the verifier reaches its saturation region. The useful operating point depends on both verifier cost and selection regret. On the 6,397-candidate Qwen3-32B reasoning pool, tuned BMC uses 2.75 total CPU hours and tuned GoldDiff uses 2.53 hours.

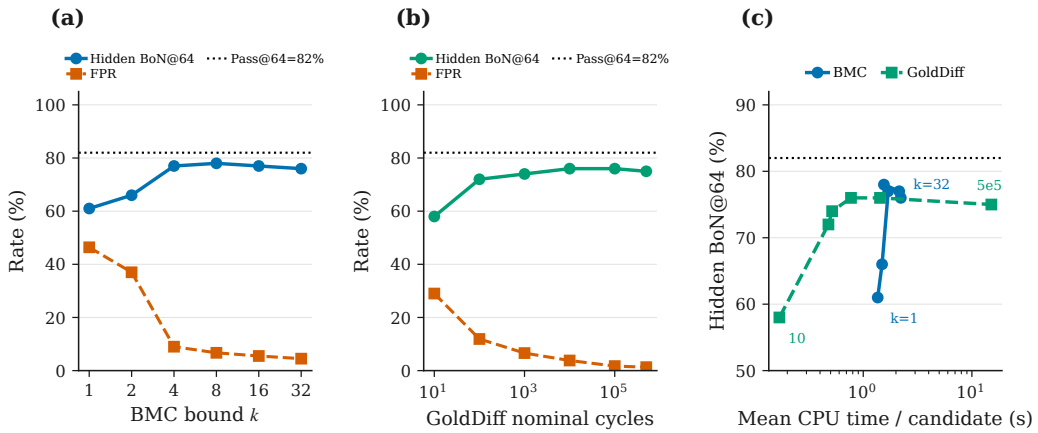


Figure 2: **Verification effort–fidelity frontier.** Stronger BMC bounds and longer simulation traces reduce selection error. Most gains occur before $k = 8$ for BMC and around 10^4 cycles for GoldDiff on this benchmark. The cost panel reports process-tree CPU time rather than cross-host wall time.

Table 2: **Qwen3-32B direct-mode cumulative repair outcomes at turn 10.** Counts use the fixed 100-task denominator. Oracle feedback uses the formal oracle as the exposed reward.

Feedback	Proxy success	Oracle success	Gap
GoldDiff	38	37	1
BMC	37	30	7
Oracle	30	30	0
DeepSeek-TB Golden	38	28	10
DeepSeek-TB Spec	41	28	13
Grok-TB Golden	33	26	7
Grok-TB Spec	48	26	22

Increasing GoldDiff from 10^5 to 5×10^5 cycles raises total CPU use to 26.84 hours while hidden BoN@64 changes from 76% to 75%. We avoid using raw cross-proxy wall time as an intrinsic cost measure because several jobs experienced scheduling and host contention.

5.4 Repair amplifies differences between feedback signals

Qwen3-32B direct-mode repair provides a second type of optimization pressure. Under GoldDiff feedback, cumulative oracle success increases from 15 tasks at turn 0 to 37 at turn 10; visible proxy success reaches 38. Under BMC feedback, oracle success reaches 30 tasks while proxy success reaches 37. Grok-TB Spec shows a larger divergence: visible cumulative success reaches 48 tasks, while oracle success reaches only 26. Hidden progress stops after turn 5 even though the exposed reward continues to improve. Table 2 summarizes the turn-10 outcomes.

Repair is therefore not uniformly ineffective. Strong feedback helps the model accumulate correct solutions and keeps visible success close to hidden success. Weak feedback can keep rewarding revisions after functional progress has stalled. The Oracle feedback control removes the public–hidden gap by construction (30% / 30% at turn 10), but it does not exceed GoldDiff's hidden cumulative success (37%). On this direct-mode repair setup, the bottleneck is therefore not only weak feedback: even oracle feedback leaves most tasks unsolved.

Reasoning-mode repair also exposes the reward-reliability problem. GoldDiff reaches 69 proxy successes and 66 Oracle proofs, whereas Grok-TB Spec reaches 83 and 63, respectively. On 38 tasks, a Grok-TB Spec-passing RTL receives a formal counterexample at the same turn; GoldDiff has 7 such tasks. Thus weak feedback can accept functional errors even when the generator uses reasoning; the aggregate gap is not explained only by unresolved Oracle outcomes. These are fixed-denominator observations with 82–90 completed trajectories per condition. Different decoding settings and in-

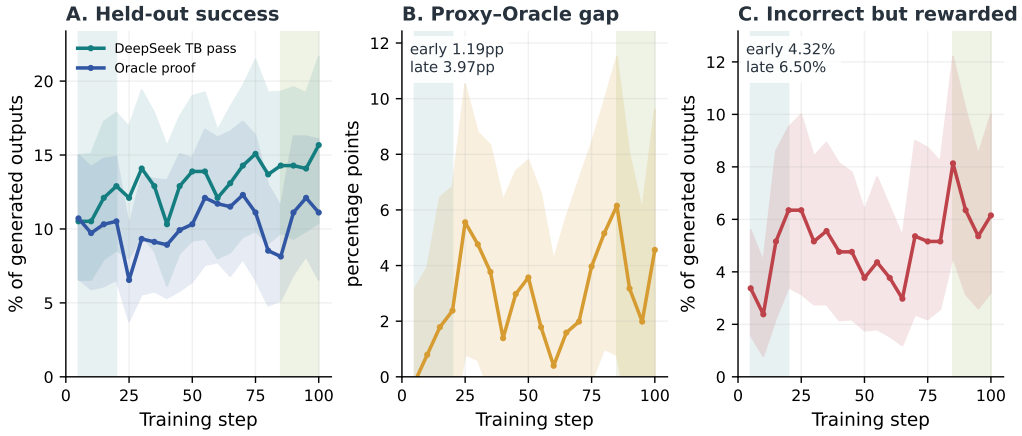


Figure 3: **Observed reward disagreement in the Qwen3-4B RL trajectory.** In the Qwen3-4B-tagged DeepSeek-testbench run, (A) held-out testbench pass rises while Oracle proof changes little; (B) the proxy-minus-proof gap grows by 2.78 percentage points from early (steps 5–20) to late (85–100) checkpoints; (C) the share of all outputs that pass the testbench and receive a formal counterexample grows by 2.18 points. Curves plot individual checkpoints; the reported early and late values average the four checkpoints in each shaded window. Each checkpoint contains 504 logged outputs over 100 tasks. Translucent bands show pointwise 95% task-bootstrap intervals, excluding training-seed uncertainty. The run tag does not independently verify the immutable model checkpoint.

complete trajectories prevent an isolated estimate of reasoning's effect; full curves and coverage are in Appendix J.1.

Single-turn repair regressions and Oracle-label coverage are detailed in Appendix J and Appendix F.

5.5 Observed reward disagreement during policy optimization

We use the Qwen3-4B-tagged DeepSeek-testbench run as the main RL case. It supplies 20 joint validation checkpoints through step 100 on the held-out 100-task set. Figure 3 shows testbench pass, formal proof, their aggregate difference, and the rate of outputs that both pass the testbench and receive an Oracle counterexample. The early (steps 5–20) and late (85–100) windows were chosen after inspecting the trajectory; their bootstrap intervals are exploratory.

Averaging each metric over the four checkpoints in each window, held-out testbench pass rises from 11.51% to 14.58%, a 3.08-point change (95% task-bootstrap interval: 1.39 to 4.91). Oracle proof moves from 10.32% to 10.62%, a 0.30-point change (interval: -1.39 to 2.08). Consequently, the proxy-minus-proof gap grows from 1.19 to 3.97 points, a 2.78-point increase (interval: 0.89 to 4.71). The fraction of all generated outputs that pass the testbench yet receive a formal counterexample rises from 4.32% to 6.50%, a 2.18-point increase (interval: 1.07 to 3.47). Changes use unrounded rates. A formal counterexample establishes functional disagreement under the adopted equivalence model; an unresolved Oracle outcome does not.

Some improvement in proxy pass accompanies compilation recovery: compile failures fall from 45.54% early to 37.25% late. Nevertheless, among outputs without a compilation failure, the testbench-pass/counterexample rate also rises from 7.92% to 10.36%. The observation is therefore not explained solely by a growing compilable fraction. It shows more rewarded, demonstrably incorrect outputs, while it does not establish a decline in overall functional correctness.

The reward and testbenches remain fixed across these checkpoints. The increase in testbench-pass/formal-counterexample outputs therefore identifies a growing rewarded error mode in the policy's sampled outputs under a simple, reproducible pass/fail objective; it does not require inferring that a particular testbench loophole was learned.

The supporting 8B trajectory and its interpretation are in Appendix G.1 and Appendix F.

6 FAILURE ANALYSIS

6.1 WHY FINITE REWARDS MISS BUGS

The coverage of each reward suggests concrete mechanisms for the observed false positives. LLM-generated testbenches may omit input transitions; differential simulation may miss rare sequences; shallow BMC may miss a mismatch beyond its bound. These mechanisms are possible explanations for the aggregate results, not a measured taxonomy.

One directly inspectable example is the 8-bit RAM task `0003__64833__ram`. A Qwen3-32B reasoning candidate (`sample_00.v`) passes the specification-prompted DeepSeek testbench, but GoldDiff rejects it and both BMC and the hidden Oracle find counterexamples. The golden RAM holds its read address until the next clock edge; the candidate changes its read address immediately. The testbench checks the output only after clock edges, so it misses this difference between edges. Appendix E shows the RTL and a concrete input sequence. This example illustrates a testbench coverage gap; it does not show that RL learned this particular error.

Circuit-class analysis and broader implications appear in Appendices F and O.

7 LIMITATIONS

Additional scope and compute limitations are in Appendix P.

Oracle is stronger than the exposed proxies, but its verdicts still depend on the equivalence model, reset assumptions, supported constructs, timeouts, and infrastructure. Not every candidate receives a binary label. We therefore report coverage and distinguish counterexamples from unsupported or failed tool runs.

The RL evidence consists of one two-epoch trajectory per tagged model. The 4B and 8B runs differ in model setting and rollout count, and no matched Oracle-reward control is reported. About 64% of 4B and 72% of 8B validation outputs receive binary Oracle labels; unresolved outcomes remain distinct from formal counterexamples. The `rl_val` split is monitored every five steps, so it is a validation set rather than an untouched final test.

8 CONCLUSION

Inference-time search can make automated RTL generation more capable, but its success depends on the reward being optimized. Across 50,948 candidates, we find that model scale and reasoning create better candidate pools, while verifier quality determines how much of that capability is recovered. Weak testbench rewards can report high visible success and still select formally incorrect designs. This pattern appears with both Grok- and DeepSeek-labeled testbench teachers and under both specification and golden prompt sources. Tuned BMC and differential simulation recover most of the hidden oracle ceiling, and their quality depends strongly on verification effort. Multi-turn repair shows the same pattern under adaptive feedback in both direct and reasoning modes, including reasoning-mode proxy passes with formal counterexamples. During two epochs of testbench-reward RL, the 4B-tagged run shows a widening proxy–Oracle gap and a growing fraction of rewarded formal counterexamples; the 8B-tagged run shows a persistent discrepancy over the same budget. These results do not show a clear fall in overall Oracle proof rate. Reliable hardware optimization should report visible reward, hidden correctness, selection regret, verifier effort, and oracle coverage together.

REPRODUCIBILITY STATEMENT

The main paper defines the candidate pools, proxy rewards, hidden oracle, optimization regimes, and evaluation metrics. The appendix specifies the verification and outcome policies, prompt structure, task-bootstrap procedure, and RL analysis limits.

AI USE DISCLOSURE

Generative models are part of the experimental method: Qwen3 generated RTL candidates, and Grok and DeepSeek generated testbenches. AI assistance was also used to draft and revise manuscript text, organize literature, develop and modify analysis and plotting code, inspect results. The authors are responsible for the final claims, citations, code, and artifacts.

REFERENCES

Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 1999.

Aaron R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, 2011.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations*, 2023. arXiv:2207.10397.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024. arXiv:2304.05128.

Zhirong Chen, Kaiyan Chang, Zhuolin Li, Cangyuan Li, Xinyang He, Chujie Chen, Mengdi Wang, Haobo Xu, Yinhe Han, Huawei Li, and Ying Wang. ChipSeek: Optimizing Verilog generation via EDA-integrated reinforcement learning. *arXiv preprint arXiv:2507.04736*, 2025.

DeepSeek-AI. DeepSeek-V4-Flash. `https://huggingface.co/deepseek-ai/DeepSeek-V4-Flash`, 2026.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *Nature*, 645:633–638, 2025. arXiv:2501.12948.

Niklas Eén, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design*, 2011.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning*, 2023. arXiv:2210.10760.

Razine Moundir Ghorab, Emanuele Parisi, Cristian Gutierrez-Gomez, Miquel Alberti-Binimelis, Miquel Moreto, Dario Garcia-Gasulla, and Gokcen Kestor. NotSoTiny: A large, living benchmark for RTL code generation. *arXiv preprint arXiv:2512.20823*, 2025.

Lukas Helff, Quentin Delfosse, David Steinmann, Ruben Härle, Hikaru Shindo, Patrick Schramowski, Wolfgang Stammer, Kristian Kersting, and Felix Friedrich. LLMs gaming verifiers: RLVR can lead to reward hacking. *arXiv preprint arXiv:2604.15149*, 2026.

En-Ming Huang, Yu-Hung Kao, Ren-Hao Deng, Wei-Po Hsin, Yao-Ting Hsieh, Cheng Liang, Hsiang-Yu Tsou, Mu-Chi Chen, Yu-Kai Hung, Shao-Chun Ho, Po-Hsuang Huang, Shih-Hao Hung, and H. T. Kung. Structured testbench generation for LLM-driven HDL design and verification-oriented data curation. *arXiv preprint arXiv:2606.12983*, 2026.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination-free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

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 *Advances in Neural Information Processing Systems*, 2022. arXiv:2207.01780.

Kezhi Li, Min Li, Xiangyu Wen, Shibo Zhao, Jieying Wu, Junhua Huang, and Qiang Xu. FormalRTL: Verified RTL synthesis at scale. *arXiv preprint arXiv:2603.08738*, 2026.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023a. arXiv:2305.01210.

Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, et al. ChipNeMo: Domain-adapted LLMs for chip design. *arXiv preprint arXiv:2311.00176*, 2023b.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. VerilogEval: Evaluating large language models for Verilog code generation. In *IEEE/ACM International Conference on Computer-Aided Design*, 2023c. arXiv:2309.07544.

Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. RTLCoder: Outperforming GPT-3.5 in design RTL generation with our open-source dataset and lightweight solution. *arXiv preprint arXiv:2312.08617*, 2023d.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. RTLLM: An open-source benchmark for design RTL generation with large language model. In *Asia and South Pacific Design Automation Conference*, 2024. arXiv:2308.05345.

William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. In *International Conference on Learning Representations*, 2022. arXiv:2201.03544.

Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da Tsai, Mingjie Liu, Wenfei Zhou, Brucek Khailany, and Haoxing Ren. Comprehensive Verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on RTL design and verification. *arXiv preprint arXiv:2506.14074*, 2025.

Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. AutoBench: Automatic testbench generation and evaluation using LLMs for HDL design. In *ACM/IEEE International Symposium on Machine Learning for CAD*, 2024. doi: 10.1145/3670474.3685956.

Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. CorrectBench: Automatic testbench generation with functional self-correction using LLMs for HDL design. In *Design, Automation and Test in Europe Conference*, 2025. doi: 10.23919/DATE64628.2025.10992873.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, 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.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward hacking. In *Advances in Neural Information Processing Systems*, 2022. arXiv:2209.13085.

SpaceXAI. Introducing Grok 4.6. `https://x.ai/news/grok-4-6`, 2026.

Fu Teng, Miao Pan, Xuhong Zhang, Zhezhi He, Yiyao Yang, Xinyi Chai, Mengnan Qi, Liqiang Lu, and Jianwei Yin. VeriRL: Boosting the LLM-based Verilog code generation via reinforcement learning. *arXiv preprint arXiv:2508.18462*, 2025.

Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated Verilog RTL code generation. In *Design, Automation and Test in Europe Conference*, 2023a. arXiv:2212.11140.

Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. AutoChip: Automating HDL generation using LLM feedback. *arXiv preprint arXiv:2311.04887*, 2023b.

Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. RTLFixer: Automatically fixing RTL syntax errors with large language models. *arXiv preprint arXiv:2311.16543*, 2024. DAC 2024.

Jing Wang, Shang Liu, Wenji Fang, Yuchao Wu, Yugao Zhu, and Zhiyao Xie. RTL-BenchLS: A large-scale benchmark for RTL reasoning and generation with large language models. *arXiv preprint arXiv:2606.08976*, 2026a.

Yiting Wang, Guoheng Sun, Wanghao Ye, Gang Qu, and Ang Li. VeriReason: Reinforcement learning with testbench feedback for reasoning-enhanced Verilog generation. In *Great Lakes Symposium on VLSI*, pp. 948–953, 2026b. doi: 10.1145/3787109.3815310.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

YosysHQ. Symbiyosys. `https://github.com/YosysHQ/sby`, 2026. Software repository; accessed 2026-09-26.

Bingchen Zhao, Dhruv Srikanth, Yuxiang Wu, and Zhengyao Jiang. SpecBench: Measuring reward hacking in long-horizon coding agents. *arXiv preprint arXiv:2605.21384*, 2026.

Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Yang Zhao, Pengwei Jin, Shuyao Cheng, Shengwen Liang, Xishan Zhang, Rui Zhang, Zidong Du, Qi Guo, Xing Hu, and Yunji Chen. QiMeng-CodeV-R1: Reasoning-enhanced Verilog generation. In *Advances in Neural Information Processing Systems*, 2025.

A Background: Hardware Description and Verification

This section introduces the RTL verification concepts that underlie our experiments. Appendices C and L cover these concepts in more detail, including what a pass guarantees for each verification procedure.

RTL hardware description. Register-transfer level (RTL) hardware description languages describe digital circuits before physical implementation. An RTL module has typed input and output signals and may contain both combinational logic and state. Combinational outputs depend only on current inputs. Sequential circuits also contain registers or memories that update on clock events, so their outputs can depend on a long history of prior inputs. We call the generated candidate the *design under test* (DUT) and the reference implementation the *golden design*.

Why correctness differs from ordinary program execution. An RTL module denotes hardware that reacts to many possible input sequences, clock events, and reset conditions. Running one example shows only one trace through this behavior. A testbench can therefore pass even when the DUT is wrong on another input or after a longer sequence of state updates. This issue is important for generated hardware because a functional bug that survives verification becomes part of the synthesized circuit.

Simulation and testbenches. A testbench is a simulation program that instantiates the DUT, drives input signals, advances simulated time or clocks, and checks outputs. An LLM-generated testbench embeds a finite set of stimuli and expected values. Differential simulation instead drives the same stimuli into both the DUT and the golden design and compares their outputs. Both methods are empirical: a pass covers only the traces that were executed.

Bounded and stronger formal reasoning. BMC builds a joint gold–DUT model and asks a solver for an input or state sequence that makes their outputs differ within a finite bound. A BMC pass means that no counterexample was found within that bound; it is not an unbounded equivalence

proof. Our hidden Oracle uses a stronger formal-equivalence procedure that combines property-directed reachability and SMT-based proof search. Its goal is to prove output equivalence for all behaviors allowed by the adopted clock, reset, and initialization model. Appendix C provides a more detailed tutorial, and Appendix L gives the exact checker contracts.

B Additional Study Design

B.1 Research questions

- **RQ1:** How do model scale and reasoning affect the pool of correct RTL candidates?
- **RQ2:** How does proxy–oracle agreement change as Best-of-N pressure increases?
- **RQ3:** How does verifier effort affect fidelity, selection regret, and cost?
- **RQ4:** Does iterative repair increase hidden correctness or mainly improve the exposed reward?
- **RQ5:** During RL, how do proxy reward and hidden correctness change across policy checkpoints?

B.2 Task and generator preparation

The full extracted RL training split contains 3,033 tasks. From a 1,000-task preparation pool, we retained 815 tasks whose golden designs passed GoldDiff self-check, Oracle proof, and bounded checking, and whose specification-prompted DeepSeek testbench passed its golden self-check. The 100-task `rl_val` split has no `problem_id` overlap with this training subset. These gates make the training reward well-defined on every selected task, while restricting the task distribution studied by RL. Each task provides a natural-language specification and a golden RTL implementation. Based on the golden RTL, the evaluation set contains 40 combinational, 39 sequential, 9 memory, 8 datapath, and 4 finite-state-machine tasks. These categories are used only for stratified analysis; all headline rates use the full 100-task set.

The baseline includes eight Qwen3 configurations: 4B, 8B, 14B, and 32B parameter models, each in direct and reasoning mode. The model receives the task specification but not the golden implementation. We target 64 candidates per task with a maximum of 16,384 generated tokens. Reasoning-mode sampling uses temperature 0.6, top-p 0.95, and top-k 20. Direct-mode sampling uses temperature 0.7, top-p 0.8, and top-k 20. Candidate i uses random seed i, starting from zero. Generation produced 50,948 candidates in total; one Qwen3-32B reasoning task has 61 rather than 64 extracted RTL candidates because three responses failed RTL extraction. The generator manifests record the model alias, sampling settings, and per-sample seeds.

B.3 RL training configuration

The RL launcher uses veRL GRPO and vLLM sampling in Qwen3 reasoning mode (temperature 0.6, top-p 0.95, top-k 20). For the reported trajectories, the run tags record a 1,024-token prompt cap, a 16,384-token response cap, and 16 rollouts per prompt for 4B or eight for 8B. The launcher sets a batch size and PPO minibatch size of 16, one PPO microbatch per GPU, dynamic token batching, gradient checkpointing, and FSDP parameter, optimizer, and activation offloading. It enables a reference-policy KL loss of type `low_var_kl` with coefficient 0.05, disables KL shaping of the scalar reward, and sets the entropy coefficient to zero. The metrics confirm an actor learning rate of 10^{-6} and KL coefficient 0.05 at every reported step.

B.4 Testbench conditions and golden self-check

At candidate-scoring time, a testbench contains its own stimuli and expected values and is simulated only with the candidate DUT; it does not instantiate a live copy of the golden design. During testbench authoring, however, we require each retained checker to pass on the golden RTL. DeepSeek retries for at most eight attempts and retains only a golden-self-checking bench. The frozen artifact contains 100 specification-prompted and 100 golden-prompted DeepSeek benches. The initial Grok specification set passed on 85 of 100 golden designs. For the remaining 15, the stimuli were retained

Table 3: **Two-by-two LLM testbench design.** Grok 4.6 is the proprietary teacher; DeepSeek V4 Flash is the open-weight teacher. Each is prompted from the specification or from the golden RTL. The final testbenches are separately validated against the golden design before candidate scoring.

Judge ID	Testbench model	Effort	Prompt source	Available
`llmtb_spec`	Cursor Grok 4.6	Extra-high	Specification + ports	100
`llmtb_golden`	Cursor Grok 4.6	Extra-high	Golden RTL + ports	100
`llmtb_ds_spec`	DeepSeek V4 Flash	High	Specification + ports	100
`llmtb_ds_golden`	DeepSeek V4 Flash	High	Golden RTL + ports	100

while expected values or timing were edited against the golden RTL, yielding a 100/100 self-check. We therefore use "specification" and "golden" to describe the information in the generator prompt, not to claim that the final specification-prompted checker was never validated against the golden design. This distinction is important when interpreting the testbench results.

B.5 Verification ladder

Each verifier answers a different question about a candidate. The short descriptions below are enough to read the results; Appendix L gives the exact contracts, coverage, and meaning of a pass.

We use the following verifiers:

1. **Syntax** checks whether the generated RTL parses and compiles.
2. **Grok-TB Spec/Golden** runs a Grok 4.6 testbench prompted from either the specification or golden RTL.
3. **DeepSeek-TB Spec/Golden** runs a DeepSeek V4 Flash testbench under the same two prompt-source conditions.
4. **GoldDiff** performs differential simulation over a controlled random-test budget. We use "cycles" as shorthand for the configured outer-times-inner test steps. The exact clock toggles depend on whether the design is combinational, clocked, or reset-bearing. The headline budget is 10^5 nominal steps. Clocked designs without a detected reset receive eight warm-up toggles.
5. **BMC** searches for an equivalence counterexample through bound k using a gold–candidate miter. The headline sequential bound is $k = 8$; combinational tasks use depth one. Passing BMC means no counterexample was found within the bound.
6. **Oracle** performs the stronger formal-equivalence check used as the hidden reference.

The effort study evaluates BMC at $k \in \{1, 2, 4, 8, 16, 32\}$ and GoldDiff at $\{10, 10^2, 10^3, 10^4, 10^5, 5\times10^5\}$ nominal steps. The tuned headline comparison uses BMC at $k = 8$ and GoldDiff at 10^5 nominal steps. The frozen syntax records use a 60-second timeout. The frozen testbench, GoldDiff, and BMC records use 120-second limits.

Oracle constructs a gold–candidate miter and uses SymbiYosys in `mode prove`. Its engines are ABC PDR and SMTBMC with Bitwuzla. The sequential proof depth is 32, with depth one for combinational designs. Generated proof jobs use a 300-second sequential limit and a 120-second combinational limit. The stored pool contains mixed outer-wrapper limits of 150 and 300 seconds, which we preserve in the coverage report rather than describing as one uniform timeout. Reset ports are constrained in the generated miter according to the detected polarity, and candidate outputs are asserted equal to the golden outputs. A SymbiYosys `PASS` is mapped to `PROVED_EQUIVALENT`; `FAIL` is mapped to `COUNTEREXAMPLE`. Timeout, unsupported constructs, and infrastructure failures are retained as separate outcomes rather than functional counterexamples.

B.6 Repair runs and message construction

Qwen3-32B reasoning-mode repair was run on the same 100 tasks under GoldDiff, BMC, four teacher-testbench conditions, and an Oracle-feedback control. The corresponding run summaries

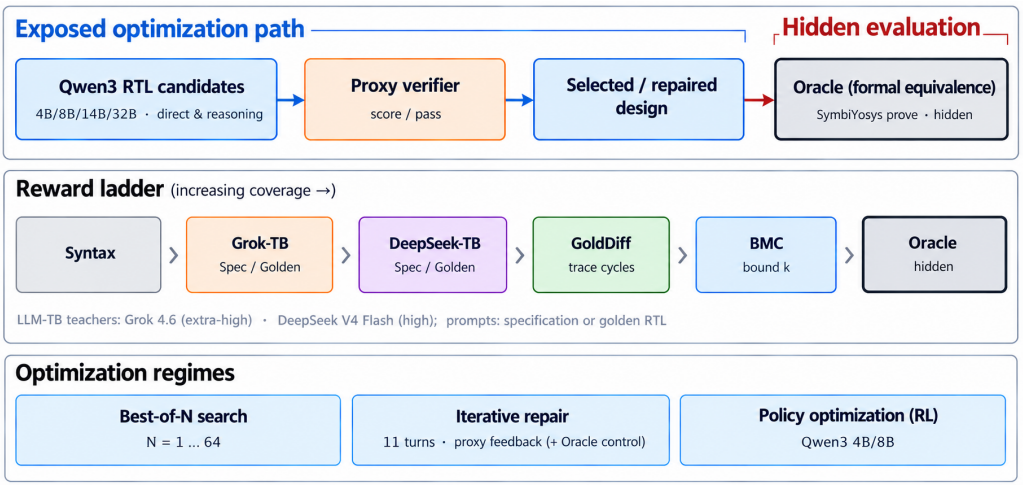


Figure 4: **Study design.** Candidate RTL is optimized using an exposed proxy verifier. A hidden formal-equivalence oracle evaluates selected candidates and repair turns as well as sampled RL checkpoints. Verifier effort increases from finite testbenches to longer simulation, bounded checking, and the hidden oracle.

record 82–90 complete trajectories per condition. All 6,115 saved RTL files in the six proxy-feedback runs were subsequently scored by Oracle. The analysis follows the recorded trajectory manifests: 135 saved files absent from those manifests are excluded from the outcome curves. A failed extraction has no RTL to score; incomplete trajectories retain successes observed before failure. Direct and reasoning modes use different decoding settings and token limits, so their outcomes are descriptive rather than a controlled causal comparison of reasoning alone.

At each repair turn, the model sees the original system and specification messages, its most recent RTL response, and only the newest feedback message. The feedback names the assigned verifier, turn, status, compile outcome, and available diagnostic summary, then asks for a corrected module with unchanged ports. The full earlier conversation is omitted by the sliding-window policy; Appendix M specifies the message construction.

B.7 Oracle outcomes and statistics

Oracle can return a proof, a counterexample, a timeout, an unsupported outcome, or an infrastructure error. In the Qwen3-32B reasoning pool, 5,340 of 6,397 candidates receive binary oracle labels: 2,210 are proved equivalent and 3,130 produce counterexamples. The remaining outcomes are 697 infrastructure errors, 336 unsupported cases, and 24 timeouts. We report coverage beside candidate-level metrics and separate tool failures from functional counterexamples.

Headline task-level rates use 100 tasks. For the main Qwen3-32B reasoning selection comparisons, we use 10,000 paired task-bootstrap resamples and also measure sensitivity to candidate-prefix and tie order (Appendix Q). RL figures use all 504 logged outputs at each checkpoint for sample-level rates. Pointwise bands use 1,000 task-bootstrap resamples; early-to-late intervals use 3,000 resamples of the 100 validation tasks. These intervals capture task variation, not training-seed variation or post-hoc selection of analysis windows. Candidate-level precision, recall, FPR, and MCC use only binary-labeled Oracle candidates. Repair tables in the main paper use the conservative fixed denominator of 100 tasks; valid-label rates are reported separately in the appendix.

C Hardware and RTL Primer for ML Readers

This appendix is written for readers with an ML background who may not work with hardware design. It expands the short primer in Appendix Section A. No result in the main paper depends on electronics knowledge beyond the definitions below.

C.1 FROM RTL SOURCE CODE TO A CIRCUIT

RTL is a programming notation for digital hardware. A Verilog module declares input and output bit-vectors and describes how outputs and internal state are computed. Unlike an ordinary program that runs once from start to finish, a hardware module is continuously active. Its logic reacts whenever inputs or state change.

A *combinational* circuit has no memory and can be written abstractly as

$$y = g(x), \tag{9}$$

where current outputs y depend only on current inputs x. Examples include adders, multiplexers, and decoders. A *sequential* circuit has state:

$$s_{t+1} = f(s_t, x_t), \qquad y_t = g(s_t, x_t). \tag{10}$$

Registers and memories store s_t, a clock defines when updates occur, and a reset places the design in a known initial state. Counters, controllers, pipelines, and memories are sequential. Two sequential designs can agree for many cycles and diverge only after a particular input history, which is why short tests are incomplete.

RTL is later synthesized into a gate-level implementation and eventually fabricated or programmed onto hardware. A functional error in the RTL can therefore become a physical error in the resulting circuit. The experiments in this paper study functional behavior, not timing, power, placement, or physical design.

C.2 SPECIFICATION, GOLDEN DESIGN, AND DUT

Each benchmark task contains a natural-language specification and a benchmark-provided reference implementation, which we call the *golden RTL*. The LLM produces another RTL module, the *design under test* (DUT). The goal is not textual similarity. Many RTL programs can implement the same circuit. The goal is behavioral equivalence: for every allowed input and state trajectory, the DUT outputs should match the golden outputs under the chosen clock, reset, and initialization assumptions.

C.3 HOW THE VERIFIER LADDER DIFFERS

Table 4 gives a conceptual comparison for readers who do not work with hardware verification. The checks differ in both the evidence they use and the claim justified by a pass.

Table 4: **Conceptual verifier ladder.** "Golden required" means the checker uses the reference RTL while scoring a candidate. LLM testbenches may use the golden during authoring but do not instantiate it during candidate scoring.

Verifier	Operation	Golden required at scoring	Coverage	Meaning of pass
Syntax	Parse and elaborate DUT	No	Source legality	Legal RTL for Icarus
LLM testbench	Run generated directed tests	No	Finite chosen traces	Accepted success marker
GoldDiff	Random gold–DUT co-simulation	Yes	Finite random traces	Outputs agree on run traces
BMC	Solver search through bound k	Yes	All traces up to bound	No CEX within bound
Oracle	Formal proof search	Yes	Allowed unbounded behaviors	Equivalence proved by engines

Syntax. The syntax checker invokes Icarus Verilog to parse and elaborate the DUT. It catches malformed or structurally invalid RTL. It does not apply inputs or check function, so a well-formed but completely wrong circuit passes.

LLM-generated testbench. A testbench is another Verilog program. It instantiates the DUT, sets input signals, optionally toggles a clock or reset, and compares outputs with expected values. The evaluator records a pass when DUT-plus-testbench compilation succeeds and simulation output contains an accepted success marker. Its coverage is limited by the stimuli and expected values chosen by the teacher LLM.

GoldDiff. GoldDiff instantiates both the DUT and golden design, applies the same random inputs to both, and compares their outputs. This removes the need for the testbench author to write expected output values. It is still finite simulation: a rare input or long temporal dependency can remain untested.

Bounded model checking. BMC creates a mathematical gold–DUT miter and asks a solver whether an output mismatch is reachable within k state transitions. Unlike simulation, it quantifies over all inputs within that bound. It can produce a counterexample trace when a mismatch exists. A pass is weaker than a proof because a mismatch may require more than k transitions.

Oracle. Oracle uses SymbiYosys (YosysHQ, 2026) `mode prove` with ABC PDR (Eén et al., 2011) and SMTBMC. These engines attempt an inductive or reachability-based proof rather than stopping at the public BMC bound. The oracle is still conditional on the encoded model: clock detection, reset polarity, initialization, two-state treatment of undefined values, supported Verilog constructs, and timeouts all affect its coverage. We therefore report non-binary oracle outcomes explicitly.

D LLM Testbench Generation Details

D.1 Grok 4.6 conditions

The Grok folders are `grok-4.6-xhigh-from-spec` and `grok-4.6-xhigh-from-golden`. We use this family as a state-of-the-art proprietary teacher at extra-high effort. The first prompt contains the task specification and module ports; the second contains the golden RTL and ports. Neither prompt contains both sources. Both use the same Grok 4.6 extra-high-effort generator, so their difference is the information source rather than model identity.

The original specification-prompted Grok benches passed their golden self-check on 85 of 100 tasks. The other 15 had expected-value, timing, or interpretation errors. Their existing stimuli were kept, while expected values or timing were edited in place against `golden.v`; no golden-prompted bench was copied into this condition. The final 100 benches used for candidate scoring all pass on the golden RTL. The golden-prompted set independently contains 100 benches and also passes 100/100 golden self-checks.

As a sensitivity check on the main Qwen3-32B reasoning pool, the 85 unedited tasks have Oracle Pass@64 of 72/85 and Grok-Spec visible/proved BoN@64 of 79/85 and 52/85; 24/85 Grok-Spec selections both pass the bench and receive an Oracle counterexample. On the 15 edited tasks, the corresponding counts are 10/15, 14/15, 9/15, and 3/15. The proved selection rate is therefore similar in the two groups (61.2% and 60.0%), and the aggregate 61% result is not confined to the edited tasks. These groups differ in task difficulty and the 15 were selected by initial self-check failure. The pre-edit benches are not retained in the frozen scoring pool, so this split does not estimate the causal effect of editing them.

D.2 DeepSeek V4 Flash conditions

The DeepSeek folders are `deepseek_v4_flash_from_spec` and `deepseek_v4_flash_from_golden`. We use this family as a state-of-the-art open-weight teacher. Generation uses high reasoning effort in a closed loop with at most eight attempts; only a bench that passes the golden self-check is retained. The frozen artifact contains 100 specification-prompted and 100 golden-prompted benches. As with Grok, the specification condition does not show golden RTL in its prompt and the golden condition does not show the natural-language specification.

D.3 Scoring contract

All four final testbench sets are static during candidate evaluation. The DUT is compiled with the selected `tb.v` using Icarus Verilog. A pass is recorded when compilation succeeds and simulation output contains an accepted success marker. The current evaluator does not independently count

assertions or recompute the mismatch total, so the generated testbench is part of the trusted reward implementation. The golden design is not co-simulated at scoring time. Thus these rewards measure whether a candidate passes a fixed teacher-authored test suite, not whether it matches the golden on arbitrary random inputs.

E Illustrative Testbench Blind Spot

For task `0003__64833__ram`, the frozen DeepSeek specification testbench passes Qwen3-32B reasoning candidate `sample_00.v`, while the frozen GoldDiff, BMC, and Oracle evaluations reject it; BMC and Oracle return counterexamples. The relevant implementation difference is:

```
golden:    address_reg <= address;   // at posedge clk
           assign q = memory[address_reg];
candidate: assign q = mem[address];
```

The testbench writes `0xAB` to address `0x00` and `0xCD` to address `0xFF`. It checks the output after a rising edge with the address held steady, but never checks immediately after an address change. To expose the bug, clock in address `0x00`, then change the address to `0xFF` before the next rising edge. The golden output remains `0xAB`; the candidate output becomes `0xCD`. We confirmed both outputs with a direct simulation; the frozen testbench reports zero errors for both implementations. This is a mismatch missed by the fixed checks in the specification-generated testbench. The source files and recorded verdicts are in the frozen task and candidate artifacts.

F Additional Results and Analyses

F.1 Numerical overview

The completed experiments give a consistent result. Better generators create better candidate pools. Oracle Pass@64 reaches 82% for Qwen3-32B in reasoning mode, and Qwen3-4B reasoning already matches 32B direct at both Pass@1 and Pass@64. Whether search recovers that capability depends strongly on the verifier. Tuned BMC and GoldDiff select designs with 78% and 76% oracle correctness. LLM-generated testbench variants select Oracle-proved designs on only 61–66 of 100 tasks, although 84–93 selected designs pass the testbench used to select them. Reducing BMC depth from $k = 8$ to $k = 2$ raises its false-positive rate from 6.7% to 37.0% and reduces hidden BoN@64 from 78% to 66%. In direct-mode repair, GoldDiff reaches 38% visible and 37% hidden cumulative success, while Grok-TB Spec reaches 48% visible but only 26% hidden success. The Oracle feedback control reaches 30% cumulative success with a zero public–hidden gap. Under Qwen3-4B-tagged testbench-reward training, the held-out proxy–Oracle gap grows by 2.78 percentage points between early and late checkpoint windows (95% task-bootstrap interval: 0.89 to 4.71); the share of outputs that pass the testbench but receive a formal counterexample grows by 2.18 points (1.07 to 3.47). Oracle proof changes by only 0.30 points (interval includes zero).

F.2 One-shot and sampling details

One-shot capability rises with size, but not smoothly. At Pass@1, direct generation improves from 5% at 4B to 8%, 10%, and 14% at 8B, 14B, and 32B. Reasoning generation improves from 14% to 19%, 20%, and 33%. The largest one-shot jump is therefore not a uniform step in parameter count: it appears when the 32B model uses reasoning. Direct-mode Pass@64 is almost unchanged from 4B to 8B (both 20%) before rising to 30% and 45%. These comparisons are within one model family and should not be read as a universal parameter-scaling law.

Sampling expands the proved task set, especially for reasoning. At Pass@64, reasoning-mode coverage is 45%, 47%, 61%, and 82% from 4B through 32B. Reasoning improves Pass@64 at every size, and the absolute advantage over direct generation grows with scale: $+25$ points at 4B, $+27$ at 8B, $+31$ at 14B, and $+37$ at 32B. The Pass@64–Pass@1 search gain is likewise larger in reasoning mode: 31, 28, 41, and 49 points, compared with 15, 12, 20, and 31 points for the matching direct configurations.

Table 5: **Oracle candidate-pool capability.** Each configuration is evaluated on 100 tasks. "Proved" is the candidate-level proof rate. Pass@N is task-level with a fixed 100-task denominator. "Direct" denotes no-thinking generation. Labeled-candidate counts and the full Pass@N curves are in Tables 9 and 10.

Generator	Candidates	Proved	Pass@1	Pass@8	Pass@64
Qwen3-4B direct	6,304	334 (5.3%)	5%	14%	20%
Qwen3-4B reasoning	6,283	736 (11.7%)	14%	29%	45%
Qwen3-8B direct	6,374	504 (7.9%)	8%	13%	20%
Qwen3-8B reasoning	6,395	871 (13.6%)	19%	34%	47%
Qwen3-14B direct	6,396	540 (8.4%)	10%	19%	30%
Qwen3-14B reasoning	6,399	1,373 (21.5%)	20%	40%	61%
Qwen3-32B direct	6,400	918 (14.3%)	14%	29%	45%
Qwen3-32B reasoning	6,397	2,210 (34.5%)	33%	68%	82%

Table 6: **Tuned proxy selection across generators.** Hidden is Oracle BoN@64; visible is the exposed proxy pass rate of the selected candidate with missing winners counted as zero. All rates use 100 tasks. BMC uses $k = 8$ and GoldDiff uses the nominal 10^5-step budget.

	Oracle	BMC		GoldDiff	
Generator	Pass@64	Hidden	Visible	Hidden	Visible
Qwen3-4B direct	20%	20%	28%	13%	17%
Qwen3-4B reasoning	45%	43%	49%	35%	36%
Qwen3-8B direct	20%	20%	30%	12%	17%
Qwen3-8B reasoning	47%	42%	50%	40%	43%
Qwen3-14B direct	30%	26%	33%	22%	22%
Qwen3-14B reasoning	61%	53%	64%	53%	61%
Qwen3-32B direct	45%	40%	50%	35%	40%
Qwen3-32B reasoning	82%	78%	86%	76%	84%

The 32B reasoning curve also shows where additional samples help most. Pass@N rises from 33% at $N = 1$ to 68% at $N = 8$, then only from 68% to 82% between $N = 8$ and $N = 64$. Most of the recoverable task coverage is therefore already present in a modest prefix, and $N = 8$ on this pool already exceeds the Pass@64 of every weaker generator. Direct 32B continues to pick up late solutions (35% at $N = 32$, 45% at $N = 64$), so sampling remains useful. The strongest reasoning pool is simply the setting in which a selector has the most correct designs to find.

Correct candidates become denser, not only more tasks become solvable. The candidate-level proved rate rises from 5.3% for 4B direct to 14.3% for 32B direct, and from 11.7% for 4B reasoning to 34.5% for 32B reasoning. Reasoning therefore roughly doubles to triples the density of proved designs at each size, not only the number of tasks that are ever solved. On Qwen3-32B reasoning, 2,210 of 6,397 candidates are proved equivalent. That density is what later lets a faithful selector recover most of the 82% Pass@64 ceiling.

Coverage does not reverse the ranking. Candidate-level labeling is high for seven of the eight pools (91.1–92.7%) and lower for 32B reasoning (83.5%), mainly because of more infrastructure errors (697 versus 182 on 32B direct). The conservative Pass@N metric treats unlabeled candidates as not proved, so the 82% ceiling is a lower bound on available correctness. At $N = 64$, 94 of 100 tasks still have at least one binary oracle label, matching the other reasoning pools. The ranking is therefore not an artifact of missing labels on a few tasks.

Table 6 shows that tuned formal and simulation proxies benefit from the stronger pools rather than collapsing as capability grows. BMC remains within 0–8 points of the Oracle Pass@64 ceiling

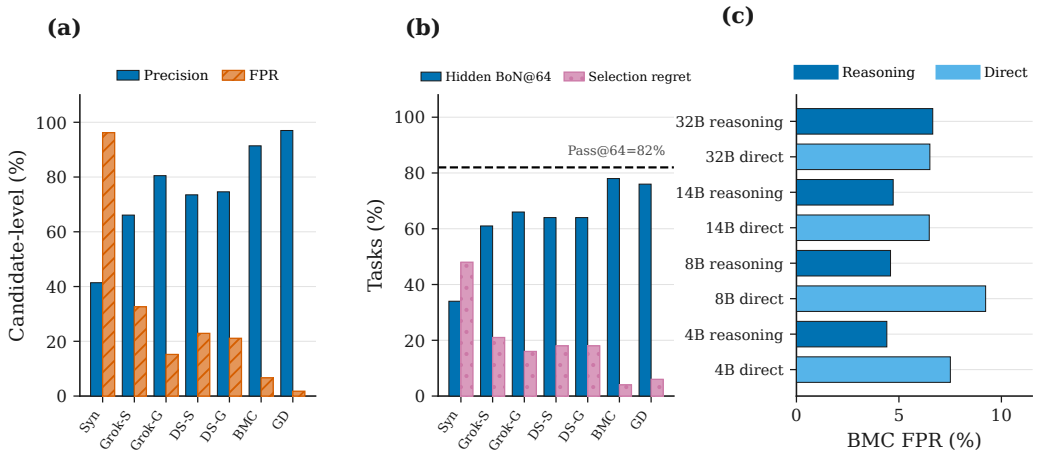


Figure 5: **Verifier quality across the reward ladder.** Candidate-level precision and false-positive rate predict whether a verifier can select the correct design at task level. Error rates vary with both the verifier and the generator distribution.

across all eight configurations. GoldDiff remains within 6–10 points. The absolute number of correct selections therefore grows strongly with model capability: BMC rises from 20% on 4B direct to 78% on 32B reasoning, and GoldDiff rises from 13% to 76%. At the same time, reasoning does not monotonically increase selection regret. Generator capability and verifier fidelity are related but distinct axes. Appendix Figure 7 gives the full cross-model breakdown.

F.3 Teacher and prompt-source contrasts

The two-by-two testbench design separates model family from prompt source. Giving Grok the golden RTL instead of the specification improves hidden BoN@64 from 61% to 66% and reduces candidate-level FPR from 32.6% to 15.2%. For DeepSeek, both prompt conditions reach 64% hidden BoN@64; golden prompting does not improve selection and only slightly reduces FPR from 22.9% to 21.1%. Grok Golden still leads DeepSeek (66% vs. 64%), but DeepSeek Spec now leads Grok Spec (64% vs. 61%). All four testbench rewards remain well below tuned BMC and GoldDiff. The public–hidden gap is therefore not explained by one teacher model or one prompt source.

F.4 Verifier diagnostics explain the selection gap

Figure 5 summarizes candidate-level diagnostics supporting the task-level selection results. On the Qwen3-32B reasoning pool, tuned GoldDiff has 97.0% precision and a 1.8% false-positive rate. Tuned BMC has 91.4% precision and a 6.7% false-positive rate. Across generators, the BMC false-positive rate ranges from 4.4% to 9.2% under Oracle. The stronger proxies therefore remain imperfect, but their errors are much less frequent than those of weak or shallow checks.

F.5 Full verifier-effort diagnostics

For every effort setting we reselect from the frozen candidate pool using the baseline rule: take the highest binary proxy score among candidates with a usable verdict, breaking ties by sample index. A task without a usable verdict has no selected design and counts as not proved on the fixed 100-task denominator. Appendix Q reports the recomputation source.

The effort pipeline treats a missing or failed proxy outcome on an Oracle-binary candidate as proxy-negative for candidate-level diagnostics. The baseline diagnostic table drops that proxy row instead. Thus GoldDiff FPR is 1.7% in Table 7 and 1.77% in Table 1; these candidate diagnostics have different inclusion rules even though the task-level selection rule is now aligned.

Table 7: **Verifier-effort study on Qwen3-32B reasoning.** Candidate metrics use the common Oracle binary-label set in the full-pool effort study. Hidden BoN@64 uses the baseline no-winner selection policy on a fixed 100-task denominator. CPU is mean process-tree CPU time per candidate.

Verifier	Effort	Precision	FPR	MCC	Hidden BoN@64	CPU (s)
BMC	$k = 1$	60.3%	46.4%	0.566	61%	1.358
BMC	$k = 2$	65.5%	37.0%	0.637	66%	1.490
BMC	$k = 4$	88.7%	9.0%	0.894	77%	1.697
BMC	$k = 8$	91.4%	6.7%	0.920	78%	1.549
BMC	$k = 16$	92.6%	5.5%	0.916	77%	2.143
BMC	$k = 32$	93.8%	4.5%	0.925	76%	2.217
GoldDiff	10 cycles	65.4%	29.0%	0.479	58%	0.169
GoldDiff	10^2 cycles	82.1%	11.9%	0.663	72%	0.478
GoldDiff	10^3 cycles	89.2%	6.6%	0.728	74%	0.517
GoldDiff	10^4 cycles	93.5%	3.8%	0.765	76%	0.771
GoldDiff	10^5 cycles	97.0%	1.7%	0.794	76%	1.426
GoldDiff	5×10^5 cycles	97.5%	1.3%	0.785	75%	15.107

F.6 Repair interpretation

For the direct-mode results in this paragraph, the cumulative metric records whether a task has been solved at least once; it does not imply that the final design remains correct. At turn 10, single-turn oracle success is 18 tasks for GoldDiff, 17 for BMC, and 11 for Grok-TB Spec. This difference between cumulative and current correctness shows that later repairs can regress a previously correct design. Oracle scoring is also incomplete on some turns: depending on the feedback condition, only 83–88 tasks have at least one binary oracle label cumulatively through turn 10; current turn-10 binary coverage is 79–85 tasks. The main counts conservatively use all 100 tasks, while Appendix J reports the valid-label denominators. The direct-mode proxy and Oracle trajectories for all feedback conditions are shown in Appendix Figure 9; reasoning-mode trajectories and their completion coverage are in Figure 10 and Table 15.

F.7 Supporting RL interpretation

Under the same two-epoch budget, the Qwen3-8B-tagged run has a positive proxy–Oracle gap and rewarded formal counterexamples throughout its reported trajectory (Appendix G.1). Its change within this budget is smaller, so we do not treat it as an independent confirmation of the 4B gap-growth estimate. Different rollout counts prevent a controlled model-size comparison. The task-bootstrap intervals therefore describe these observed runs, not generalization over training runs.

F.8 Circuit complexity

The benchmark contains 40 combinational, 39 sequential, 9 memory, 8 datapath, and 4 finite-state-machine tasks. Table 8 reports Oracle Pass@64 and tuned BMC BoN@64 within each class (denominator = tasks in the stratum). The weighted Pass@64 across classes recovers the headline 82% rate. Combinational and datapath tasks are largely recovered by BMC; the sequential class shows the largest Pass@64–BMC gap (76.9% vs. 66.7%). Memory and FSM rates are exploratory because their sample sizes are small. Figure 6 visualizes the Oracle and BMC columns.

G Complete Figure Inventory

The following figures preserve the important analyses that may not fit in the main nine-page paper:

1. Full comparison of Grok/DeepSeek and specification/golden testbench conditions across model generators.

Table 8: **Oracle circuit-class rates on Qwen3-32B reasoning.** Pass@64 and BMC BoN@64 use the stratum task count as the denominator.

Circuit class	n	Pass@64	BMC BoN@64	GoldDiff BoN@64
Combinational	40	85.0%	85.0%	82.5%
Sequential	39	76.9%	66.7%	64.1%
Memory	9	77.8%	77.8%	77.8%
Datapath	8	87.5%	87.5%	87.5%
FSM	4	100%	100%	100%

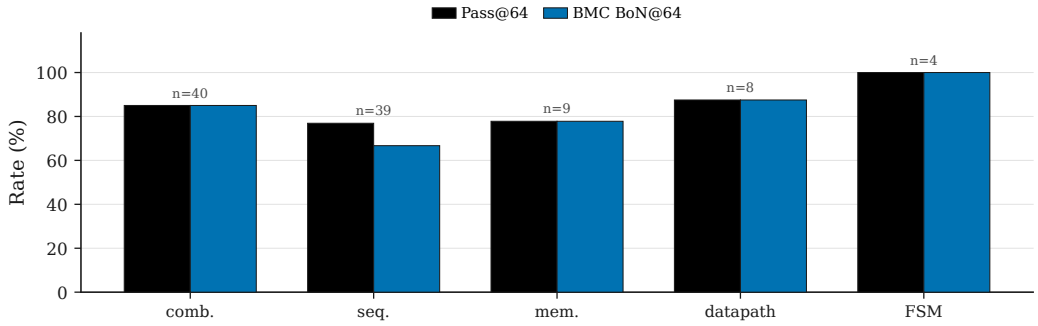


Figure 6: **Correctness by circuit class.** Oracle Pass@64 and tuned BMC BoN@64 on the Qwen3-32B reasoning pool. Counts above the bars give the number of tasks in each class; small memory and FSM strata are exploratory.

2. Full Pass@N curves for all eight generators (Table 10).
3. Hidden BoN@64 by proxy and generator.
4. Public–hidden gap at $N = 64$ by model size and reasoning mode.
5. Proxy precision and false-positive rate for every generator.
6. Hidden BoN@N curves for all BMC and GoldDiff effort tiers (Figure 8).
7. Candidate-level precision, FPR, and mean CPU cost for each effort tier (Table 7).
8. Public and hidden repair curves for all feedback conditions in both direct and reasoning modes, with reasoning-mode completion coverage.
9. Single-turn and cumulative direct-mode repair correctness.
10. The full Qwen3-8B-tagged RL proxy, Oracle, gap, and rewarded-counterexample curves (Figure 11).
11. Circuit-class results with exact sample sizes.

G.1 Supporting Qwen3-8B RL trajectory

The Qwen3-8B-tagged DeepSeek-testbench run has 20 joint validation checkpoints across the reported two epochs. Comparing steps 5–20 with 85–100, held-out testbench pass changes from 18.80% to 21.18%, while Oracle proof changes from 13.44% to 14.78%. The proxy-minus-proof gap is positive in both windows (5.36 and 6.40 percentage points), and testbench-pass/formal-counterexample outputs account for 7.19% and 7.84% of all outputs. These descriptive differences are smaller than in the 4B case and do not establish a replicated widening effect.

H Complete Oracle Coverage

Table 9 reports every Oracle outcome in the baseline pools. Binary-labeled candidates are the sum of proved and counterexample outcomes. Task-level Pass@N keeps the fixed 100-task denominator, while candidate-level diagnostic metrics use only binary labels. Table 10 reports the complete Oracle

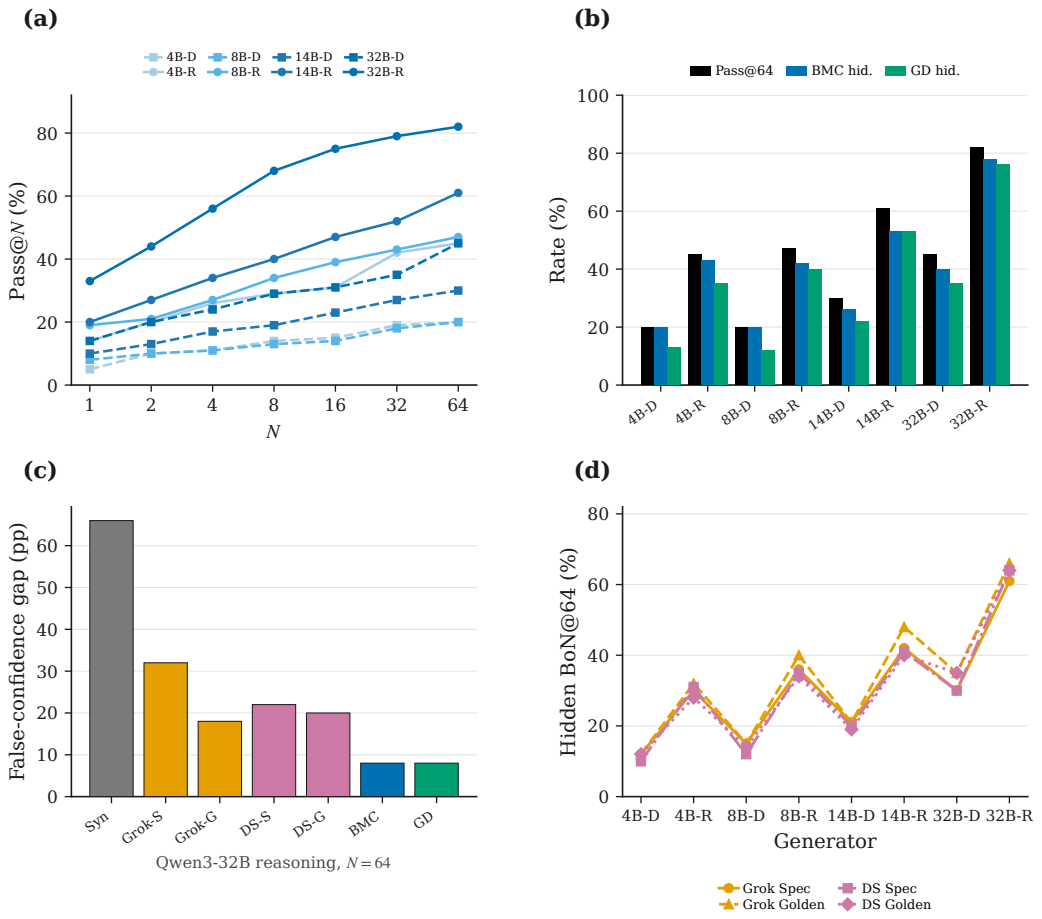


Figure 7: **Full cross-model results.** This composite retains the complete eight-generator comparison behind the main-paper summary.

Pass@N curves used in Section 5.1. Table 11 contrasts fixed-denominator and winner-conditional visible pass rates for the main selection pool.

Table 9: **Oracle outcomes for all candidate pools.**

Generator	Total	Proved	CEX	Infra.	Unsupported	Timeout	Labeled
Qwen3-4B direct	6,304	334	5,454	94	410	12	5,788
Qwen3-4B reasoning	6,283	736	5,091	80	368	8	5,827
Qwen3-8B direct	6,374	504	5,344	85	434	7	5,848
Qwen3-8B reasoning	6,395	871	5,042	112	364	6	5,913
Qwen3-14B direct	6,396	540	5,305	186	358	7	5,845
Qwen3-14B reasoning	6,399	1,373	4,492	155	343	36	5,865
Qwen3-32B direct	6,400	918	4,913	182	369	18	5,831
Qwen3-32B reasoning	6,397	2,210	3,130	697	336	24	5,340

I Cross-Model Proxy Diagnostics

Table 12 reports tuned BMC and GoldDiff quality on each generator distribution. Precision, FPR, and MCC are computed only on candidates with a binary Oracle label. The number of proxy-scored, binary-labeled candidates can differ by proxy because missing proxy outcomes are excluded.

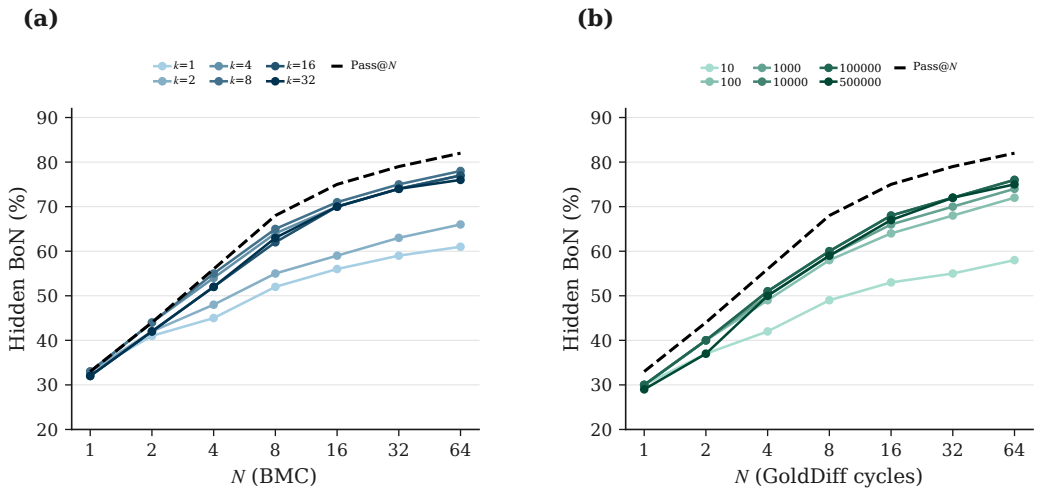


Figure 8: **Hidden correctness across search size and verifier effort.** Panels show Best-of-N Oracle proof rates for every tested (A) BMC bound and (B) GoldDiff simulation budget. The main-paper Figure 2 shows the $N = 64$ endpoint, false-positive rates, and CPU cost.

Table 10: **Complete Oracle Pass@N curves.** Each entry is the number of tasks, out of 100, for which the first N candidates contain at least one proved-equivalent design. Unlabeled candidates count as not proved. At $N = 64$, 93–95 tasks have at least one binary oracle label in every pool.

Generator	@1	@2	@4	@8	@16	@32	@64
Qwen3-4B direct	5	10	11	14	15	19	20
Qwen3-4B reasoning	14	20	26	29	31	42	45
Qwen3-8B direct	8	10	11	13	14	18	20
Qwen3-8B reasoning	19	21	27	34	39	43	47
Qwen3-14B direct	10	13	17	19	23	27	30
Qwen3-14B reasoning	20	27	34	40	47	52	61
Qwen3-32B direct	14	20	24	29	31	35	45
Qwen3-32B reasoning	33	44	56	68	75	79	82

J Repair Denominators and Full Curves

The direct-mode main repair table uses a fixed denominator of 100 tasks and counts any task without a proof as not proved. Table 13 gives the alternative valid-label denominator at turn 10. This view produces higher oracle percentages because infrastructure and timeout outcomes are excluded; we therefore do not mix these rates with fixed-denominator values. Table 14 gives the fixed-denominator trajectory at every direct-mode repair turn.

J.1 Reasoning-mode repair

Table 15 and Figure 10 report the available Qwen3-32B reasoning-mode trajectories. All counts use the fixed 100-task denominator, with any success before a trajectory fails retained. The gap is proxy success minus Oracle proof and is conservative when the Oracle returns an unresolved outcome. The last column instead counts tasks with at least one *same-turn* proxy pass and formal counterexample. For Grok-TB Spec this occurs on 38 tasks, versus 7 for GoldDiff. Thus the larger Grok-TB Spec gap is supported by direct functional mismatches, not only by unresolved Oracle cases. The two modes have different decoding settings; these numbers should not be interpreted as the isolated effect of reasoning.

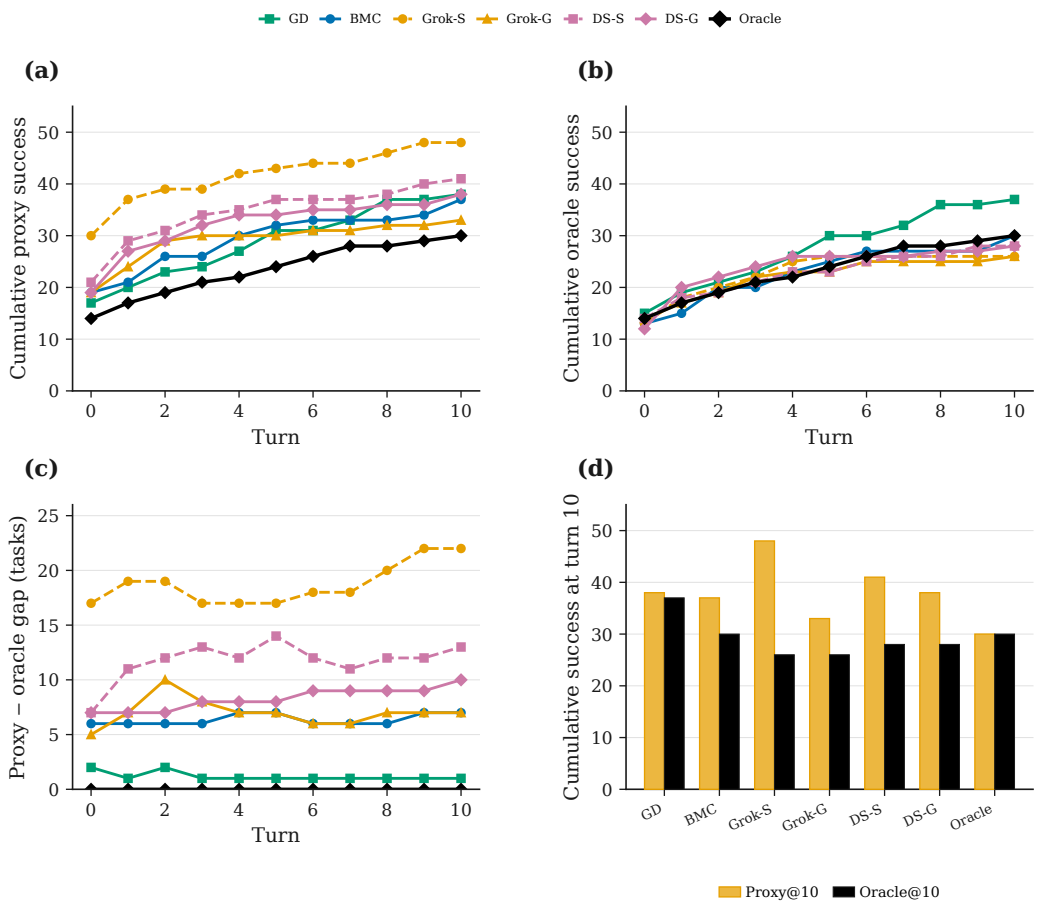


Figure 9: **Full Qwen3-32B direct-mode iterative-repair results.** Proxy and Oracle trajectories, their gap, and turn-10 outcomes are shown for every feedback condition.

Table 11: **Fixed versus conditional visible pass on Qwen3-32B reasoning at $N = 64$.** The main paper uses fixed visible pass and matched gap. Conditional pass is retained for comparison with earlier generated tables.

Selector	Winners	Fixed visible	Conditional visible	Hidden	Matched gap
Syntax	100	100%	100.0%	34%	66 pp
Grok-TB Spec	99	93%	93.9%	61%	32 pp
Grok-TB Golden	97	84%	86.6%	66%	18 pp
DeepSeek-TB Spec	99	86%	86.9%	64%	22 pp
DeepSeek-TB Golden	97	84%	86.6%	64%	20 pp
BMC ($k = 8$)	95	86%	90.5%	78%	8 pp
GoldDiff (10^5 nominal)	99	84%	84.9%	76%	8 pp

K Verifier Cost Accounting

We record elapsed wall time, process-tree CPU time, peak memory, and timeout caps. Table 16 summarizes tuned-verifier costs. CPU time is the preferred quantity for the effort frontier; wall time is machine- and scheduler-dependent and is not comparable across hosts.

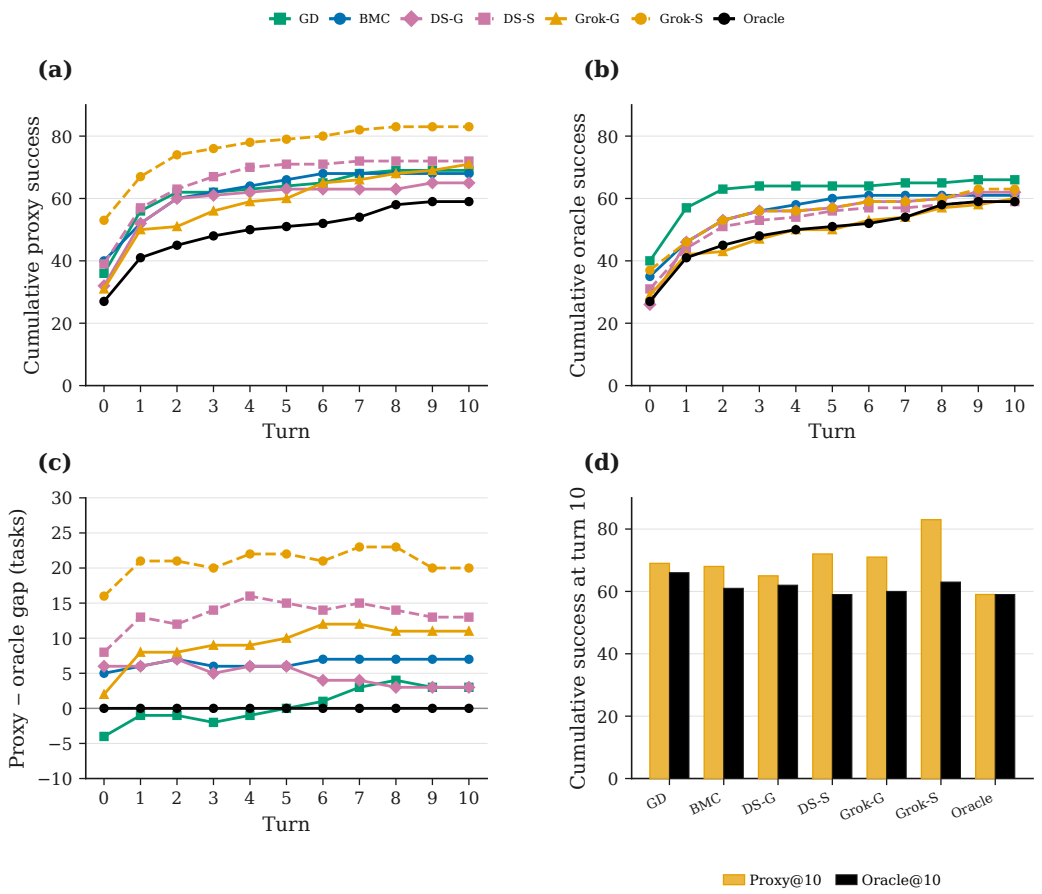


Figure 10: **Qwen3-32B reasoning-mode iterative repair.** Observed cumulative proxy success, Oracle proof, and their gap through ten feedback turns use a fixed denominator of 100 tasks. Only 82–90 trajectories complete per condition; missing turns contribute no additional success.

L Detailed Verifier Configuration

This appendix gives the exact checker contracts behind the short verifier ladder in Section 4.3. Table 4 explains what a pass means conceptually; Table 17 records the headline settings. Frozen run records and per-candidate manifests are authoritative when they differ from the current configuration.

Oracle uses Bitwuzla for SMTBMC, a 300-second sequential timeout, and a 120-second combinational timeout. The generated Yosys script applies `proc`, `memory`, `opt`, `prep`, and `setundef -zero -undriven -init`. Reset assumptions are generated from the detected polarity. The frozen manifests preserve generated commands and task-specific configuration but not historical Yosys, SymbiYosys, ABC, Bitwuzla, or Icarus version strings. The reported outcomes are conditional on these solver and undefined-value semantics.

M Generation, Repair, and RL Prompts

The initial-generation user message is the task specification, followed by "The top module name must be ..." when that metadata field is available. It does not expose the golden implementation. The frozen system message is:

```
You are an expert Verilog RTL designer.
Implement exactly one synthesizable Verilog-2001/SystemVerilog
```

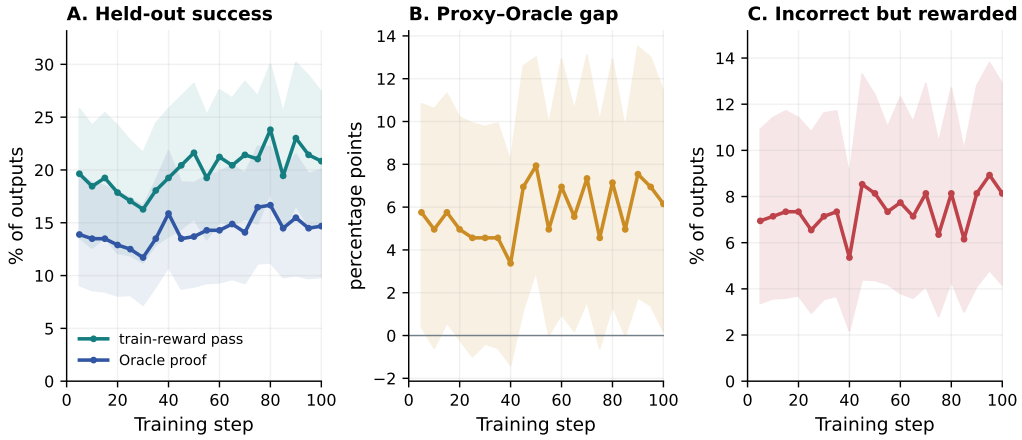


Figure 11: **Supporting 8B-tagged two-epoch RL trajectory.** Panels show (A) testbench pass and Oracle proof, (B) their aggregate gap, and (C) testbench-passing outputs with Oracle counterexamples through step 100. Each checkpoint has 504 logged outputs from the same 100 validation tasks; bands are pointwise 95% task-bootstrap intervals and exclude training-seed uncertainty. The different rollout count makes this a supporting observation rather than a controlled 4B–8B comparison.

Table 12: **Tuned BMC and GoldDiff diagnostics under Oracle.** BMC uses $k = 8$ and GoldDiff uses 10^5 cycles.

Generator	Proxy	Precision	FPR	MCC	Labeled rows
Qwen3-4B direct	BMC	55.30%	7.51%	0.7151	3,927
	GoldDiff	97.24%	0.22%	0.8717	3,480
Qwen3-4B reasoning	BMC	81.90%	4.41%	0.8848	4,132
	GoldDiff	97.06%	0.57%	0.8645	3,046
Qwen3-8B direct	BMC	58.88%	9.23%	0.7311	4,318
	GoldDiff	98.83%	0.14%	0.9171	4,050
Qwen3-8B reasoning	BMC	82.78%	4.59%	0.8887	4,790
	GoldDiff	97.32%	0.47%	0.8383	4,443
Qwen3-14B direct	BMC	62.65%	6.48%	0.7654	4,873
	GoldDiff	99.05%	0.09%	0.9018	3,749
Qwen3-14B reasoning	BMC	88.35%	4.72%	0.9175	5,060
	GoldDiff	91.68%	2.60%	0.8316	4,288
Qwen3-32B direct	BMC	77.58%	6.51%	0.8516	4,915
	GoldDiff	98.82%	0.24%	0.8913	4,700
Qwen3-32B reasoning	BMC	91.37%	6.65%	0.9235	5,330
	GoldDiff	97.00%	1.77%	0.8228	5,103

```
module that matches the specification.
Rules:
- Use the module name and ports given in the spec. Do not rename them.
- Output only the Verilog source in a ```verilog fence after you
  finish reasoning.
- Do not write a testbench. Do not instantiate a golden reference.
  Do not copy comments from a hidden solution.
- If a clock or reset is listed, use it. Do not add extra
  top-level ports.
```

Line wrapping above is typographic; the source prompt has the same words.

Table 13: **Qwen3-32B direct-mode cumulative repair at turn 10 with valid-label denominators.**

Feedback	Proxy numerator	Proxy denominator	Oracle numerator	Oracle denominator
GoldDiff	38	100	37	87
BMC	37	91	30	88
Oracle	30	100	30	100
Grok-TB Spec	48	100	26	88
Grok-TB Golden	33	100	26	83
DeepSeek-TB Spec	41	100	28	84
DeepSeek-TB Golden	38	100	28	83

Table 14: **Qwen3-32B direct-mode fixed-denominator cumulative repair curves.** Each entry is proxy-passing tasks / Oracle-proved tasks out of 100. For Oracle feedback, both counts use the Oracle verdict.

Turn	GoldDiff	BMC	Oracle	Grok Spec	Grok Golden	DeepSeek Spec	DeepSeek Golden
0	17/15	19/13	14/14	30/13	19/14	21/14	19/12
1	20/19	21/15	17/17	37/18	24/17	29/18	27/20
2	23/21	26/20	19/19	39/20	29/19	31/19	29/22
3	24/23	26/20	21/21	39/22	30/22	34/21	32/24
4	27/26	30/23	22/22	42/25	30/23	35/23	34/26
5	31/30	32/25	24/24	43/26	30/23	37/23	34/26
6	31/30	33/27	26/26	44/26	31/25	37/25	35/26
7	33/32	33/27	28/28	44/26	31/25	37/26	35/26
8	37/36	33/27	28/28	46/26	32/25	38/26	36/27
9	37/36	34/27	29/29	48/26	32/25	40/28	36/27
10	38/37	37/30	30/30	48/26	33/26	41/28	38/28

For `sliding-window-v1` repair turn $t > 0$, the message list is `[system, specification, last assistant RTL, latest verifier feedback]`. The feedback starts "Your previous RTL failed the public [verifier] verifier," then reports the verifier ID, turn, status, and compile result. Depending on the verifier, it includes a simulation error summary, selected mismatch examples, or counterexample diagnostics. It ends with these fixed instructions: keep the same module name and ports, do not add a testbench or golden reference, return only corrected Verilog in a fenced block, and do not repeat unchanged RTL unless correct. The configured repair run also includes the previous RTL block inside the feedback message. Only the assigned verifier's feedback is shown to that condition.

RL preparation uses the system message

```
You are an expert Verilog RTL designer.
Given a natural-language hardware specification,
implement a correct,
synthesizable Verilog module that matches the specified interface and behavior.

Rules:
- Output a single Verilog module implementation.
- Match module name, port names, directions, and widths from the specification.
- Prefer clear synchronous logic when a clock/reset is specified.
- Do not invent extra top-level ports.
- Put the final Verilog in one fenced code block.
```

followed by the task specification; its output contract is a single fenced Verilog module matching the named interface. The training subset has 815 tasks and excludes all 100 `rl_val` task IDs.

N Oracle Coverage and Outcome Policy

For task-level Pass@N, BoN@N, and fixed-denominator repair counts, the denominator is always 100 tasks. Only `PROVED_EQUIVALENT` is counted as oracle success; all other outcomes are not

Table 15: **Qwen3-32B reasoning-mode cumulative repair outcomes at turn 10.** Completion varies by feedback condition; proxy and Oracle counts use all 100 tasks. CEX tasks have a proxy-passing RTL with an Oracle counterexample at the same turn.

Feedback	Complete	Proxy	Oracle	Gap	CEX tasks
GoldDiff	83	69	66	3	7
BMC	86	68	61	7	10
Oracle	90	59	59	0	–
DeepSeek-TB Golden	85	65	62	3	23
DeepSeek-TB Spec	87	72	59	13	28
Grok-TB Golden	82	71	60	11	20
Grok-TB Spec	89	83	63	20	38

Table 16: **Cost of tuned verifiers on Qwen3-32B reasoning.** Large wall/CPU ratios for syntax and Grok-TB Spec indicate waiting or contention and should not be interpreted as intrinsic compute.

Verifier	Median wall (s)	Median CPU (s)	Total CPU (h)	Timeout rate
Syntax	3.718	0.033	0.07	0.00%
Grok-TB Spec	4.746	0.047	0.11	0.97%
GoldDiff (10^5)	0.665	0.644	2.53	0.36%
BMC ($k = 8$)	0.825	0.750	2.75	0.30%

proved. For candidate-level precision, recall, FPR, and MCC, only `PROVED_EQUIVALENT` and `COUNTEREXAMPLE` form the binary label set. Timeout, unsupported, unknown, and infrastructure-error outcomes are excluded and the remaining coverage is reported. Table 9 provides the full baseline outcome counts, and Table 13 shows the effect of valid-label denominators in repair.

O Discussion and Implications

Why this is a foundation-model problem. The hardware domain supplies unusually strong correctness tools, but the central question is about foundation-model post-training. A teacher model creates a reward-bearing artifact, a generator or policy is optimized against that artifact, and a stronger hidden evaluator measures whether reward improvement transfers to the intended task. Comparing Grok- and DeepSeek-generated testbenches shows that the issue is not limited to one teacher family. Comparing direct and reasoning Qwen3 generators shows that reward reliability must also be tested as the optimized model becomes more capable.

Generation and selection are separate problems. Larger and reasoning-enabled models improve the candidate pool—in this family, 4B reasoning already matches 32B direct—but stronger generation does not remove the need for a reliable selector. Pass@N should therefore be reported together with the hidden correctness of the selected candidate. Reporting only proxy pass can confuse available capability with recovered capability.

Verifier effort is part of the optimization method. The reward configuration is not a minor implementation detail. Shallow BMC and short simulation traces change which candidate is selected and which repair is accepted. Moderate effort recovers much of the candidate-pool ceiling, while the highest tested efforts show smaller gains. This suggests that verifier budgets should be chosen using both computational cost and hidden selection quality.

Implications for RLVR. An executable reward is valuable because it scales, but executability does not guarantee complete semantic coverage. The risk grows when reward optimization changes the policy distribution. RLVR studies should therefore evaluate fixed checkpoints with a hidden reference that is stronger than the training reward. Training reward, hidden correctness, missing-label coverage, and tool failures should be reported separately. Our 4B-tagged training trajectory illustrates this need directly: more outputs pass the optimized testbench and receive formal counterexamples, even though the proof rate does not show a clear decline. Golden self-check establishes

Table 17: **Headline verifier configuration.**

Verifier	Input or comparison	Headline effort	Pass meaning
Syntax	Candidate RTL	60 s frozen timeout	Parses and elaborates
Grok-TB Spec/Golden	Grok-generated TB	120 s timeout	Testbench passes
DeepSeek-TB Spec/Golden	DeepSeek-generated TB	120 s timeout	Testbench passes
GoldDiff	Candidate vs golden traces	10^5 nominal test steps	Traces agree
BMC	Gold–candidate miter	$k = 8$ sequential, 1 combinational	No CEX in bound
Oracle	Gold–candidate miter	depth 32, mixed outer limits	SymbiYosys PASS

that a testbench accepts the reference RTL, but it cannot establish that the bench rejects every incorrect candidate. The formal counterexamples rewarded by our frozen, golden-self-checked benches show why that distinction matters when the policy is optimized against them.

Toward multi-fidelity verification. Our results motivate a practical design: use syntax and inexpensive simulation for broad filtering, stronger GoldDiff or BMC for promising candidates, and a formal oracle for periodic audits. We do not claim that a particular routing policy is optimal because we have not directly evaluated one. The result is a design principle: verification effort should increase where optimization pressure or uncertainty is high.

P Additional Limitations

Our study uses one model family and 100 headline tasks. The results may not transfer directly to other generators, larger industrial modules, or different hardware-description styles. Each generator configuration contributes one candidate pool, so additional generation seeds would improve estimates of sampling variation.

The circuit categories are imbalanced. In particular, the memory and FSM subsets are too small for strong class-level claims. Repair uses Qwen3-32B; the main comparison is direct mode, while reasoning-mode repair has incomplete trajectory coverage and different decoding settings. These outcomes may not capture repair behavior at other model sizes.

Some cost measurements were collected on different hosts or under different load. Final Pareto claims will use controlled CPU measurements.

Preparing a self-checked reward for each task, sampling RTL candidates, compiling and simulating them, and periodically running formal validation all raise the cost of a training run. The 4B recipe alone requests 16 prompts times 16 rollouts per update and scores 504 validation outputs every five steps. Our available compute limited recipe searches, matched reward controls, and seed replication. Together with the verification-based 815-task filter, this bounds the result to the observed within-run reward disagreement; it does not establish the best training recipe or an average effect across tasks and seeds.

Q Additional Statistical Analysis

For the Qwen3-32B reasoning pool, we resample the 100 tasks 10,000 times while preserving all candidates and proxy outcomes within each task. Hidden BoN@64 is 78% for BMC (95% task-bootstrap interval: 70–86%), 76% for GoldDiff (68–84%), and 64% for DeepSeek-TB Spec (55–73%). Paired hidden-success differences are 14 points for BMC minus DeepSeek-TB Spec (7–22), 12 points for GoldDiff minus DeepSeek-TB Spec (4–20), and 17 points for BMC minus Grok-TB Spec (9–25). Grok-TB Golden minus Grok-TB Spec is 5 points with an interval of -2 to 12, so that within-teacher difference is descriptive.

We also permuted candidate order independently within each task 1,000 times, reselecting from random prefixes and breaking equal-score ties by the new order. Table 18 gives mean hidden-success counts and central 95% ranges over the permutations. These ranges measure order sensitivity in this fixed pool, not training or task-sampling uncertainty.

Table 18: **Candidate-order sensitivity on 100 tasks.** Entries are mean hidden-success task counts over 1,000 random orders, with central 95% ranges in parentheses. Equal proxy scores are resolved by the randomized order.

N	Grok-TB Spec	DeepSeek-TB Spec	BMC	GoldDiff
16	61.0 (55–66)	61.0 (55–67)	71.8 (67–76)	66.6 (61–71)
32	62.7 (58–67)	63.7 (59–68)	76.2 (73–79)	71.3 (68–75)
64	63.5 (59–68)	65.8 (62–70)	78.8 (77–81)	74.7 (72–78)

For the effort study, the frozen per-candidate verdicts were reselected with the baseline no-winner rule at all six BMC and GoldDiff effort settings. The only BoN@64 value changed by this reconciliation is BMC $k = 32$, from 77% to 76%; the other displayed effort values are unchanged. For the RL figures, the 100 validation tasks are the resampling units, preserving each task's multiple outputs. Pointwise 95% bands use 1,000 resamples per checkpoint. Early-to-late 95% intervals use 3,000 resamples preserving task identity across checkpoints. They do not estimate training-seed variability, uncertainty from selecting windows after viewing the curves, or uncertainty in the unrecorded model identities.