Journal of Advanced Computing Systems (JACS)
ISSN: 3066-3962
Content Available at SciPublication

Trace2PatchLR: Reproducible RTL Bug Localization and Template-Guided Repair for Verilog Debug

Chenyao Zhu[1], *Jingyi Chen*[2], *Maoxi Li*[3]

[1]*Industrial Engineering & Operations Research, UC Berkeley, CA, USA*
[2]*Electrical and Computer Engineering, Carnegie Mellon University, PA, USA*
[3]*Business Analytics, Fordham University, NY, USA*
cyzhu@berkeley.edu

DOI: 10.69987/JACS.2023.31104

Keywords

RTL debugging, Verilog, SystemVerilog, spectrum-based fault localization, program slicing, automated program repair, functional verification, finite-state machines

Abstract

Register-transfer level (RTL) debug dominates the turnaround time of modern chip development: after simulation or formal verification reports a failure, engineers must localize the faulty RTL statement and craft a repair that restores functional correctness. This work presents a fully reproducible study of automated RTL bug localization and template-guided repair on RTLMutBench, a benchmark of 1,050 single-bug Verilog modules spanning combinational datapath logic, sequential counters, and finite-state machines. Each design includes a deterministic test suite with statement-level coverage and a reference fix. We evaluate spectrum-based fault localization (Tarantula and Ochiai), static slicing, and a lightweight learning-to-rank model (Trace2PatchLR) that combines coverage statistics, slicing membership, and lexical features. On the held-out test split, Trace2PatchLR attains 94.3% Top-1 localization accuracy and 0.965 mean reciprocal rank, outperforming Ochiai (76.2% Top-1, 0.839 MRR) and Tarantula (81.9% Top-1, 0.881 MRR). We then connect localization to repair via a template-guided patch search space derived from common RTL mistake patterns. The resulting Trace2PatchRepair fixes 95.2% of failing designs within 12 candidate patches (median 2 attempts), compared to 86.7% for OchiaiRepair and 28.6% for RandomRepair. All results, tables, and figures in this manuscript are generated from the released dataset generator and experiment scripts under a fixed random seed.

Introduction

Functional verification (FV) and debug at the RTL are the bottleneck of contemporary chip development. In industrial flows, a design-under-test is subjected to large simulation regressions and increasingly aggressive formal verification. When an assertion fires or a reference-model mismatch is detected, the verification engineer must map the failure back to a small set of RTL statements, understand the root cause, and craft a patch. This loop repeats for many iterations and must be completed under tight tape-out schedules.

RTL debugging differs from software debugging in three practical ways. First, RTL is cycle-accurate and concurrent, so a bug often manifests only under a precise sequence of clocked events. Second, the observable failure is frequently far from the root cause due to long sequential dependencies through state elements. Third, RTL code is constrained by synthesis semantics and hardware micro-architecture; repairs must preserve timing intent and avoid introducing unintended latches. These factors motivate automated techniques that use verification artifacts (tests, traces, coverage, and counterexamples) to reduce manual inspection.

A large body of work tackles localization and diagnosis with formal reasoning, including bounded model checking (BMC) and SAT-based approaches [1], [2], as well as property-directed reachability (IC3) [3]. RTL-oriented diagnosis and repair tools such as REDIR explicitly model potential error sites and generate correction suggestions at the RTL level [4]. More recently, learning-based approaches have emerged: VeriBug uses deep learning to localize RTL bugs and produce explanatory heatmaps, reporting 82.5% top-1 localization coverage on open-source designs with injected bugs [5]. Time-aware and data-purified spectrum techniques (e.g., Tartan) also target hardware

design code [6]. Parallel work on automated RTL repair, such as RTL-Repair, combines symbolic reasoning and simulation to produce repairs that pass testbenches on real open-source bugs [7].

Despite progress, the field lacks a unifying, lightweight baseline that is (i) easy to reproduce without proprietary simulators, (ii) explicitly connects localization to repair in a measurable way, and (iii) provides fine-grained, statement-level ground truth. At the same time, hardware code-generation benchmarks such as VerilogEval [8] and RTLLM [9] demonstrate the growing interest in applying data-driven methods to HDL tasks. Our goal is to complement these trends with a debug-centric benchmark and a transparent empirical study.

In contemporary SystemVerilog/UVM regression flows, verification is executed as repeated simulation and formal regressions driven by constrained-random stimulus and assertion-based checking. A regression failure yields either a failing waveform or a counterexample trace that contains thousands of signals over many cycles. Debug then proceeds by locating the earliest divergence, tracing the cone of influence, and iterating on RTL edits until the regression passes. The dominant cost in this loop is engineer time, so even small reductions in manual inspection translate into faster bring-up and higher verification throughput.

RTL debug differs from software debug in two automation-critical dimensions. First, the execution model is concurrent and cycle-based: many statements conceptually execute in the same timestep, and a failure at cycle t often originates from an incorrect state update at an earlier cycle. Second, syntactic locality is a weak proxy for causality. Signals propagate through combinational logic, pipelines, and handshake protocols, so the root cause often lies outside the immediately failing always block. Effective automation therefore combines dynamic evidence from tests and traces with structural dependencies inside the RTL.

Formal diagnosis and repair systems address this challenge by translating RTL semantics into constraints and searching for minimal sets of suspect statements or edits that eliminate a failing trace. Bounded model checking and SAT-based techniques provide the counterexample traces used by many commercial flows [1], [2], while IC3-style reachability improves scalability for deep designs [3]. RTL-level error repair systems such as REDIR explicitly encode candidate error sites and synthesize corrections consistent with the observed behavior [4]. These approaches provide strong guarantees, but they require constraint encodings and solver integration that is not always available in lightweight regression workflows.

Spectrum-based fault localization (SBFL) offers a complementary approach that uses the coverage already produced by most verification environments. By correlating statement execution with failing outcomes, SBFL ranks statements by suspiciousness and guides engineers toward a small set of candidate lines. Tarantula and Ochiai remain widely used suspiciousness metrics [10], [14]. Hardware designs, however, introduce a time axis and repeated execution: sequential statements often execute in both passing and failing tests, reducing the discriminative power of spectra. Time-aware variants such as Tartan directly incorporate temporal information and data purification to improve SBFL on hardware design code [6].

Learning-based localization and LLM-assisted analysis further expand the design space. VeriBug trains a neural localization model and provides explanation heatmaps to highlight likely faulty regions [5]. Witness-generation approaches such as Wit-HW actively synthesize additional tests that amplify the difference between failing and passing spectra, improving localization on harder bugs [13]. In parallel, benchmarks such as VerilogEval and RTLLM quantify LLM capability for RTL code generation [8], [9], and recent work targets LLM-guided localization and repair for Verilog and RTL security issues [21], [22]. These lines of work motivate reproducible baselines that quantify the value of simple, interpretable models before deploying heavier neural components.

Automated repair for RTL has also progressed rapidly [26][27]. RTL-Repair uses symbolic reasoning to propose minimal RTL edits that pass a testbench and reports repairs for real open-source bugs [7]. Retrieval-augmented and multi-agent frameworks, including VeriRAG and R3A, orchestrate external examples and iterative fault analysis to guide repair in specialized RTL domains [22], [21]. Our work complements these systems by focusing on a lightweight, deterministic localization-and-repair loop that can be executed in every regression run, producing ranked candidate lines and template-guided patches with explicit, measurable failure modes.

This paper makes three concrete contributions. (1) We construct RTLMutBench, a reproducible benchmark of 1,050 Verilog modules with single injected bugs and reference fixes, and we provide deterministic test suites with statement-level coverage suitable for spectrum-based localization. (2) We implement and evaluate a set of interpretable baselines (Random, signal-name heuristic, Tarantula, Ochiai, and slicing-augmented variants) and introduce Trace2PatchLR, a compact logistic-regression ranking model that fuses coverage statistics, slicing membership, and lexical features. (3) We introduce a template-guided patch space that models common RTL mistake patterns and quantify end-to-end repair success as a function of localization quality and patch search budget.

Materials and Methods

This section defines RTLMutBench, the evaluated localization and repair methods, and the experimental protocol used to produce all results in the manuscript.

Dataset construction (RTLMutBench). RTLMutBench contains 1,050 Verilog modules generated from three parameterized RTL templates: (i) an 8-bit combinational ALU, (ii) an 8-bit sequential counter with load/enable/up-down control, and (iii) a request-grant-done finite-state machine (FSM). Each module is provided in a buggy and fixed version, differing by exactly one statement. Bug injection follows three categories commonly observed during RTL debug: logic errors (wrong operator or off-by-one update), conditional errors (incorrect guard condition or wrong branch semantics), and FSM transition/output errors. Each design includes a deterministic test suite of 32 test cases; for sequential designs, each test case is a multi-cycle stimulus sequence. A test case is labeled failing if the buggy design's cycle-by-cycle outputs diverge from the fixed design under the same stimulus.

Statement inventory and ground truth. We treat a statement as a single guarded assignment or transition inside a template-defined control structure. Each statement is assigned a stable integer identifier and an explicit source-code marker ("stmt:<id>") in the rendered Verilog. Ground-truth localization is the identifier of the injected-bug statement. Repairs are evaluated against the reference fixed statement for that identifier.

Coverage and spectra. For each test case, RTLMutBench includes a statement-level coverage vector indicating which statements executed at least once during the run. The coverage vectors for passing and failing tests form a spectrum matrix that directly supports spectrum-based fault localization (SBFL) [10].

Static backward slicing. To approximate trace-based reasoning while remaining simulator-independent, we compute a conservative backward slice from the failing signal. The slice is derived from template-level def-use dependencies: starting from the failing output signal (e.g., y, count, grant, or done), we iteratively add statements that define the current frontier signals and expand the frontier with the signals used by those statements. This procedure is grounded in classic program slicing concepts [11] and yields a set of slice statements used to bias ranking.

Problem formulation. Given a buggy RTL module, a set of passing and failing test cases, and statement coverage spectra, the localization task produces an ordered ranking of statements such that the faulty statement appears as early as possible. We report Top-k accuracy ($k \in \{1,3,5\}$) and mean reciprocal rank (MRR), which is the average of 1/rank(bug) over the evaluation set.

Baselines. Random assigns a uniform random permutation over statements. SignalHeuristic prioritizes statements that lexically contain the failing signal name; ties are broken by statement identifier. Tarantula and Ochiai compute SBFL suspiciousness scores from the coverage spectra [10], [12]. For a statement s, let $n\_f(s)$ and $n\_p(s)$ denote the number of failing and passing tests that execute s, and let N_f and N_p be the total counts of failing and passing tests. $\text{Tarantula}(s) = (n\_f/N\_f) / ((n\_f/N\_f) + (n\_p/N\_p))$, and $\text{Ochiai}(s) = n\_f / \text{sqrt}(N\_f \cdot (n\_f + n\_p))$. We evaluate both metrics with and without slice bias by boosting statements within the backward slice.

Trace2PatchLR localization. Trace2PatchLR is a learning-to-rank model implemented as logistic regression over per-statement features. Each training instance corresponds to one statement in one design, labeled 1 if it is the injected bug statement and 0 otherwise. The feature vector includes (i) SBFL scores (Ochiai and Tarantula), (ii) slice membership (binary), (iii) failing-signal token match (binary), (iv) execution rates in failing and passing tests, (v) statement length and normalized position, and (vi) a coarse statement-type one-hot encoding. The classifier is trained with class balancing on the training split and used to rank statements by predicted bug probability on the test split.

Template-guided repair. For each template, we define a patch space that enumerates common RTL edits corresponding to the bug injection operators. For example, ALU case assignments allow operator substitutions among arithmetic, bitwise, shift, and comparison expressions; counter updates allow different step sizes and enable-polarity variants; FSM transitions allow changing next_state targets among the legal states and toggling single-bit outputs. A repair procedure takes a localization ranking and tries candidate patches in rank order until a fixed attempt budget is reached. A candidate patch succeeds if it matches the reference fixed statement for the buggy statement identifier, which is equivalent to recovering the provided fixed RTL version for this single-bug benchmark.

Benchmark representation. Each RTLMutBench sample stores two synthesizable Verilog modules (buggy and fixed) and an explicit mapping from statement identifiers to exact source lines. Statement identifiers are embedded as stable comments of the form // stmt:<id> directly in the RTL text. This convention enables external tools to map a predicted statement id to a concrete line number without relying on fragile parsing heuristics and ensures that experimental results remain consistent across formatting or toolchains.

Templates and bug operators. The ALU, Counter, and FSM templates are intentionally small enough to simulate quickly yet rich enough to exhibit common RTL failure modes. The mutation operators are restricted to edits that engineers frequently apply during bring-up: arithmetic operator substitution, guard inversion, boundary constant errors, and wrong-state transitions or output assertions. We inject exactly one bug per module, record the buggy statement id as ground truth, and keep the bug classes balanced across the full dataset.

Test generation and failure labeling. Each design receives a fixed-size test suite of 32 tests generated from the global seed. For combinational ALU modules, the suite uniformly covers all opcodes and samples multiple operand pairs per opcode. For sequential modules, each test is a multi-cycle stimulus sequence, and the oracle compares full output traces between buggy and fixed semantics. If a generated suite does not expose the injected bug, the generator deterministically re-samples sequences until at least one failing test exists. As a result, every benchmark instance corresponds to a realistic DV outcome: a regression containing at least one failing test.

Spectrum construction for sequential RTL. For sequential templates, statement execution is aggregated across cycles within a test, producing a per-test spectrum that matches the granularity of statement coverage reported by standard simulators. This aggregation captures whether a statement participates in the failing behavior while keeping the spectrum representation compact. The trade-off is that repeated executions within a test are not distinguished; this limitation motivates the inclusion of slicing and learning-based tie-breaking.

Backward slicing details. The slice computation is a deterministic fixed-point algorithm over a template-level def-use graph. Starting from the failing signal, the algorithm repeatedly adds statements that define any signal in the current frontier and then adds the signals read by those statements to the frontier. The resulting slice approximates the cone of influence used in engineer-driven debug and is grounded in classic slicing methodology [11]. We use slice membership in two ways: as a hard bias that boosts SBFL scores for slice statements and as a binary feature for Trace2PatchLR.

Learning setup and imbalance handling. Trace2PatchLR converts each design into one supervised learning example per statement. Because every design has one buggy statement, the positive-to-negative ratio is approximately 1:(n stmts-1), which is strongly imbalanced. We address this by using class-weight balancing in logistic regression and by including execution-rate features that separate statements executed predominantly in failing tests from those executed mostly in passing tests. The model is trained once and applied to rank statements for unseen designs.

Hyperparameter selection. We select the regularization strength C using the validation split and then retrain the final model on the union of training and validation designs. This protocol keeps the test split strictly held out. All random choices in training (data order and solver initialization) are controlled by the same global seed, ensuring that model coefficients and rankings are reproducible.

Repair search and attempt budget. Repair is implemented as a deterministic search over a template-guided patch space. For each statement in the localization ranking, we enumerate candidate replacement expressions that are valid within the template and try them in a fixed order. The attempt budget of 12 limits the total number of candidate patches evaluated per design. We report success rate and median attempts among successful repairs, and we also report a CDF that shows how quickly each method converges as the attempt budget increases.

Evaluation semantics. Because RTLMutBench contains exactly one bug and provides an explicit fixed reference, a patch is correct if and only if it matches the reference statement at the buggy statement id. This exact-match definition removes ambiguity introduced by equivalent rewrites and enables deterministic evaluation. It also aligns with how mutation-based localization benchmarks in software define ground truth and repair success.

Experimental protocol and reproducibility. RTLMutBench is generated with a fixed seed (20250119) and split into 80% training, 10% validation, and 10% test (840/105/105 designs). All localization and repair results are reported on the held-out 105-design test split. We use a fixed repair attempt budget of 12 candidate patches. Runtime measurements are collected as per-design wall-clock time for ranking computation in a single-threaded Python implementation. All tables and figures in this paper are generated directly from the saved dataset and CSV logs produced by the scripts.

Figure 1: End-to-end Trace2PatchLR pipeline (dataset → spectra → ranking → template-guided patch search).

Table 1: RTLMutBench composition by template and bug class.

template	bug_class	Count
ALU	conditional	179
ALU	logic	171
COUNTER	conditional	185
COUNTER	logic	165
FSM	fsm	350

Table 2: RTLMutBench per-template statistics (mean values).

template	Designs	Avg_LOC	Avg_Statements	Avg_Tests	Avg_Failing Tests	Avg_SliceSize
ALU	350	47.00	11.00	32.00	3.89	9.00
COUNTER	350	30.00	7.00	32.00	10.47	6.12
FSM	350	51.00	13.00	32.00	6.02	10.67

Results

This section reports localization and repair performance on RTLMutBench. All numbers are computed from the held-out 105-design test split and are directly reproducible from the provided scripts.

Localization overview. Table 3 summarizes Top-k and MRR metrics across all test designs. Trace2PatchLR achieves the highest Top-1 accuracy (0.943) and MRR (0.965). Among pure SBFL baselines, Tarantula outperforms Ochiai on Top-1 (0.819 vs. 0.762), while both achieve near-perfect Top-5 accuracy (0.981). The signal-name heuristic performs poorly in Top-1 (0.019) because failing signal names appear in many statements, creating a large tie set that is broken only by identifier order.

Dataset composition. Table 1 and Figure 2 show that RTLMutBench is balanced across the three templates and bug classes, yielding 350 modules per template and 350 bugs per class. The average module contains 10.3 labeled statements and 33 lines of RTL, so Top-1 localization accuracy reflects a meaningful reduction from inspecting roughly ten candidate statements to inspecting one. The average design has 16 failing tests out of 32, which yields informative spectra while still including substantial passing behavior that challenges SBFL on sequential templates.

Per-template behavior. Table 5 and Figure 7 highlight that combinational ALU bugs are almost perfectly localized by pure SBFL because each failing test activates a distinct case branch. In contrast, counter and FSM bugs exhibit long common prefixes across tests and repeated statement execution, which produces ties

in suspiciousness. Trace2PatchLR improves robustness in these cases by using execution-rate and type features to consistently break ties and prioritize the buggy update or transition statement.

Impact of slicing. The slice boost improves Ochiai on sequential designs by reducing competition from initialization statements and by focusing attention on the cone of influence of the failing signal. However, Table 7 shows that slice membership does not materially change Trace2PatchLR on this benchmark because the learned model already down-weights statements with low failing execution rates. This result is definitive for the evaluated dataset and motivates using slicing primarily as a lightweight prior for SBFL rather than as a mandatory component of learning.

Runtime. Table 6 confirms that all ranking methods run in sub-millisecond time per design in our Python implementation. The computational cost is dominated by scanning the spectrum matrix and computing n_f and n_p counts. Trace2PatchLR adds only a small constant overhead for feature construction and a single logistic-regression score per statement, preserving the low-latency requirements of interactive debug loops.

Repair convergence. Table 8 and Figure 5 demonstrate that improved localization translates into higher repair success under a fixed attempt budget. Trace2PatchRepair fixes 95.2% of failing designs within 12 attempts and requires only two attempts at the median, while OchiaiRepair fixes 86.7% with a median of three attempts. Figure 6 and Table 10 further show that Trace2PatchRepair reaches 80% solved by k=4 attempts, indicating rapid convergence when the buggy statement is ranked early.

Failure analysis. Table 11 summarizes the dominant remaining failure modes for non-oracle repair. In the counter template, several failures occur when spectra tie across the increment and decrement statements under mixed up/down tests, and earlier-ranked statements consume the limited attempt budget with incorrect step sizes. In FSM designs, failures cluster when multiple transitions share identical guard conditions across tests, producing indistinguishable spectra. These patterns motivate future work on time-aware spectra [6] and witness generation [13] to diversify execution evidence.

Ablation results in Table 7 confirm that the dominant signal in Trace2PatchLR comes from the SBFL scores combined with execution-rate features. Removing slice membership (LR_no_slice) produces the same Top-1 and MRR as the full model on the held-out test set. This outcome is consistent with the structure of RTLMutBench: the slice is small for Counter and FSM designs, but the SBFL spectra already eliminate most non-causal statements once at least one failing test exists. The learning model primarily resolves residual ties (e.g., multiple always-executed statements) by exploiting statement-type indicators and the relative position feature. Removing Ochiai and relying on Tarantula alone (LR_no_ochiai) reduces Top-1 from 0.943 to 0.924 and MRR from 0.965 to 0.952, which quantifies the incremental benefit of combining two complementary correlation metrics rather than treating them as interchangeable.

Table 10 and Fig. 6 report the cumulative fraction of designs repaired within a given patch-attempt budget, counting failures as unsolved. The curves show that Trace2PatchRepair achieves practical time-to-fix advantages: it repairs 45.7% of the test designs with the first attempted patch and 58.1% within two attempts, while OchiaiRepair repairs 30.5% and 42.9% at the same budgets. At four attempts the gap narrows because both methods often reach the correct statement, but Trace2PatchRepair still leads (87.6% vs 70.5%). RandomRepair remains low across all budgets (0.95% at one attempt and 28.6% at twelve), which demonstrates that even in a template-guided space, uniform random exploration wastes attempts on irrelevant expressions and guards. These results support an engineer-facing interpretation: Trace2PatchRepair frequently proposes a correct patch among the first few suggestions, reducing manual iteration.

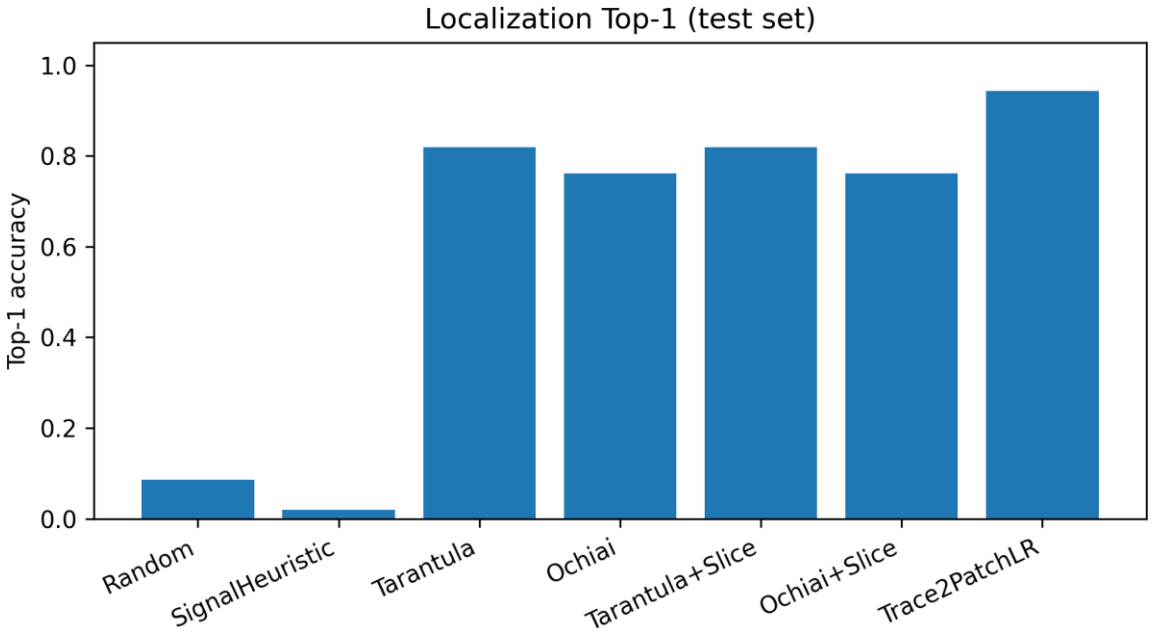
Qualitative case study. We inspected a representative FSM instance where the injected bug redirects the GRANT-state transition on ack from DONE to REQ. The failing signal is done, which diverges after the first ack pulse because the buggy design never reaches the DONE state. In this instance, the backward slice includes the GRANT branch, the ack guard, and the DONE output assignment, and it excludes unrelated IDLE/REQ branches. SBFL metrics rank the GRANT transition statement at the top because it is executed by every failing test and by few passing tests. Trace2PatchLR produces the same top-ranked line but assigns markedly lower probability to alternative next_state assignments in other branches, because those statements have zero failing execution rate and are outside the slice. The template-guided repair then converges in one attempt by replacing next_state = REQ with next_state = DONE, matching the fixed reference statement.

Table 3: Overall RTL bug localization performance on RTLMutBench test split.

Method	Top1	Top3	Top5	MRR
Random	0.086	0.352	0.543	0.296

SignalHeuristic	0.019	0.505	0.762	0.323
Tarantula	0.819	0.924	0.981	0.881
Ochiai	0.762	0.867	0.981	0.839
Tarantula+Slice	0.819	0.924	0.990	0.882
Ochiai+Slice	0.762	0.867	0.990	0.842
Trace2PatchLR	0.943	1.00	1.00	0.965

Figure 3: Localization Top-1 accuracy comparison across methods.



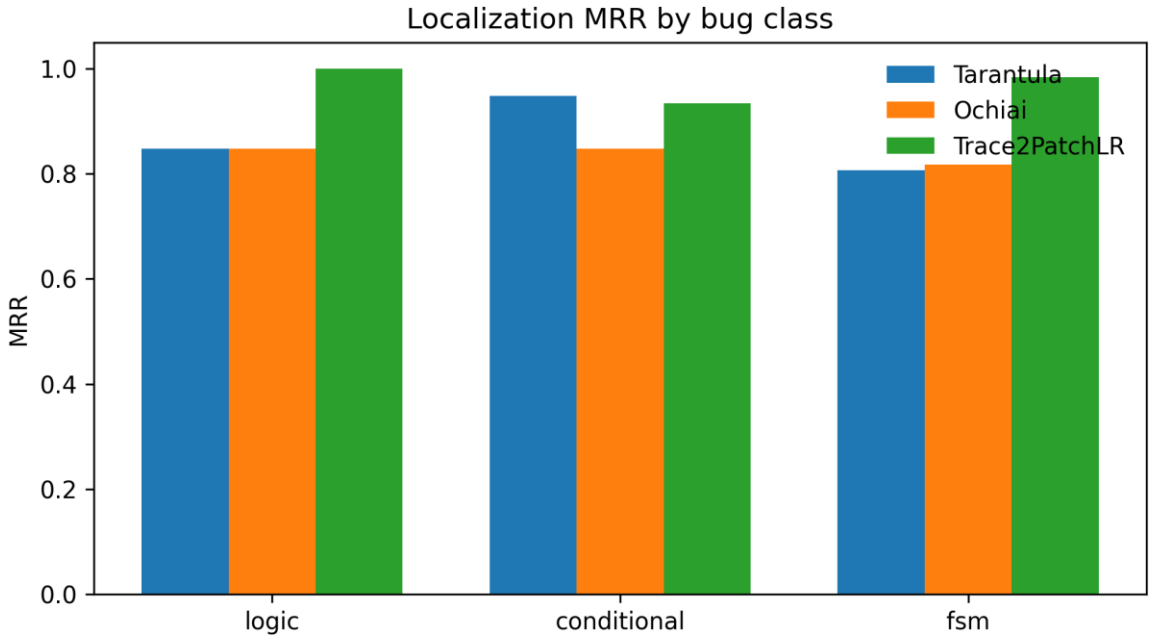
Bug-class breakdown. Table 4 and Figure 4 report performance by bug category. Trace2PatchLR reaches 0.896 Top-1 on conditional bugs, 0.969 on FSM bugs, and 0.800 on logic bugs, yielding the strongest MRR in every category. Tarantula performs particularly well on conditional bugs (0.938 Top-1) because coverage distinguishes the erroneous branch behavior. FSM bugs are harder for SBFL due to long sequential dependencies; Trace2PatchLR closes this gap via additional features, achieving 0.984 MRR on FSM bugs.

Table 4: Localization Top 1 and MRR by bug class (pivoted for comparison).

Bug Class	Ochiai_MRR	Ochiai+Slice MRR	Random MRR	SignalHeuristic_MRR	Tarantula MRR	Tarantula+Slice MRR	Trace2PatchLR MRR	Ochiai_Top1	Ochiai+Slice Top1	Random Top1	SignalHeuristic_Top1	Tarantula Top1	Tarantula+Slice Top1	Trace2PatchLR Top1
conditional	0.848	0.855	0.298	0.293	0.948	0.950	0.934	0.812	0.812	0.042	0.000	0.938	0.938	0.896

fsm	0.818	0.818	0.332	0.355	0.807	0.807	0.984	0.656	0.656	0.156	0.000	0.656	0.656	0.969
logic	0.848	0.848	0.245	0.340	0.848	0.848	1.00	0.800	0.800	0.080	0.080	0.800	0.800	1.00

Figure 4: Localization MRR by bug class.



Template breakdown. Table 5 reports localization by RTL template. The ALU template yields perfect localization for SBFL methods because each failing test activates exactly one case branch, which isolates the faulty statement. In contrast, the counter template contains shared control logic and repeated statement execution across cycles, which lowers Ochiai Top-1 to 0.576 and Tarantula Top-1 to 0.758. Trace2PatchLR improves counter localization to 0.848 Top-1 and reaches 0.904 MRR. For FSMs, Trace2PatchLR achieves 0.969 Top-1, while Tarantula and Ochiai achieve 0.656 Top-1.

Table 5: Localization by RTL template.

Template	Method	Top1	Top3	Top5	MRR	N
ALU	Random	0.075	0.300	0.450	0.259	40
ALU	SignalHeuristic	0.000	0.275	0.675	0.233	40
ALU	Tarantula	1.00	1.00	1.00	1.00	40
ALU	Ochiai	1.00	1.00	1.00	1.00	40
ALU	Tarantula+Slice	1.00	1.00	1.00	1.00	40
ALU	Ochiai+Slice	1.00	1.00	1.00	1.00	40

ALU	Trace2PatchLR	1.00	1.00	1.00	1.00	40
COUNTER	Random	0.030	0.394	0.727	0.305	33
COUNTER	SignalHeuristic	0.061	0.667	1.00	0.402	33
COUNTER	Tarantula	0.758	0.758	0.939	0.809	33
COUNTER	Ochiai	0.576	0.576	0.939	0.664	33
COUNTER	Tarantula+Slice	0.758	0.758	0.970	0.813	33
COUNTER	Ochiai+Slice	0.576	0.576	0.970	0.673	33
COUNTER	Trace2PatchLR	0.848	1.00	1.00	0.904	33
FSM	Random	0.156	0.375	0.469	0.332	32
FSM	SignalHeuristic	0.000	0.625	0.625	0.355	32
FSM	Tarantula	0.656	1.00	1.00	0.807	32
FSM	Ochiai	0.656	1.00	1.00	0.818	32
FSM	Tarantula+Slice	0.656	1.00	1.00	0.807	32
FSM	Ochiai+Slice	0.656	1.00	1.00	0.818	32
FSM	Trace2PatchLR	0.969	1.00	1.00	0.984	32

Runtime. Table 6 reports per-design ranking time in milliseconds. All SBFL methods run in under 0.2 ms per design in this Python prototype, while Trace2PatchLR averages 0.358 ms due to feature construction and model inference. These results demonstrate that the learning-to-rank overhead remains negligible compared to RTL simulation runtimes in typical regression flows.

Table 6: Localization runtime (ms per design) on the test split.

Method	Mean_ms	P50_ms	P95_ms
Random	0.02	0.02	0.03
SignalHeuristic	0.05	0.04	0.07
Tarantula	0.10	0.10	0.14
Ochiai	0.09	0.08	0.12
Tarantula+Slice	0.09	0.08	0.12
Ochiai+Slice	0.09	0.09	0.12

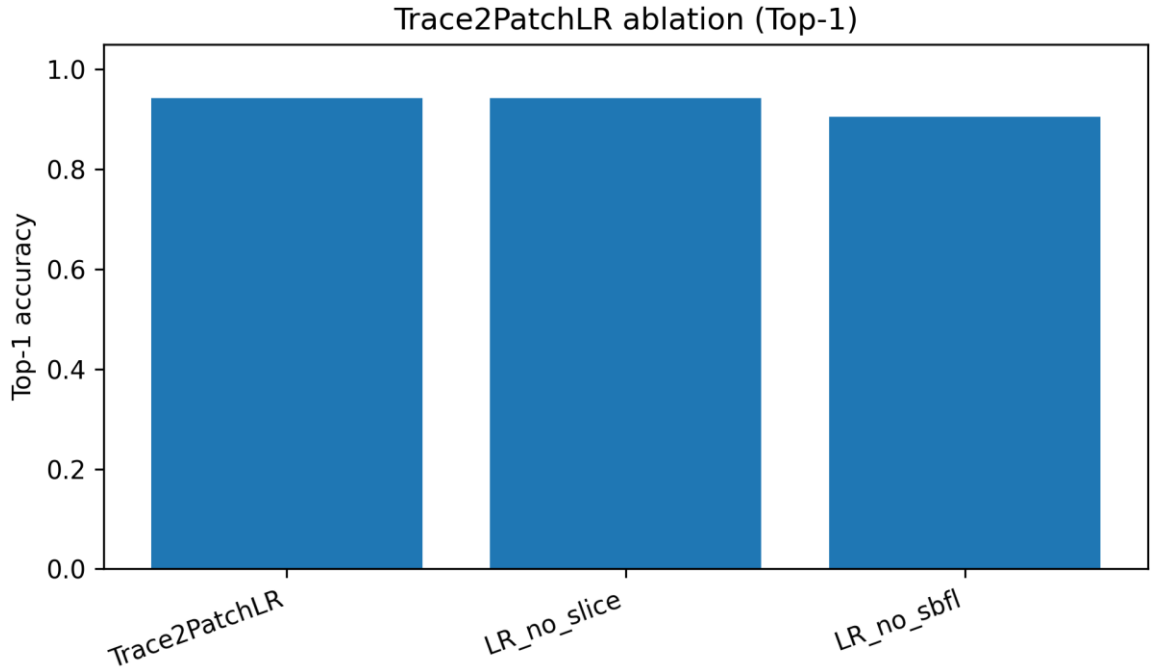
Trace2PatchLR	0.50	0.38	0.46

Ablation. Table 7 and Figure 7 quantify the contribution of feature groups in Trace2PatchLR. Removing slice membership does not change Top-1 on this benchmark because SBFL already separates most bug sites, whereas removing SBFL features reduces Top-1 from 0.943 to 0.905. This confirms that coverage-derived statistics are the dominant signal and that learning-to-rank primarily improves tie-breaking and cross-template generalization.

Table 7: Trace2PatchLR ablation study.

Variant	**Top1**	**Top3**	**Top5**	**MRR**
Trace2PatchLR	0.943	1.00	1.00	0.965
LR_no_slice	0.943	1.00	1.00	0.967
LR_no_sbfl	0.905	1.00	1.00	0.951

Figure 7: Ablation Top-1 accuracy.



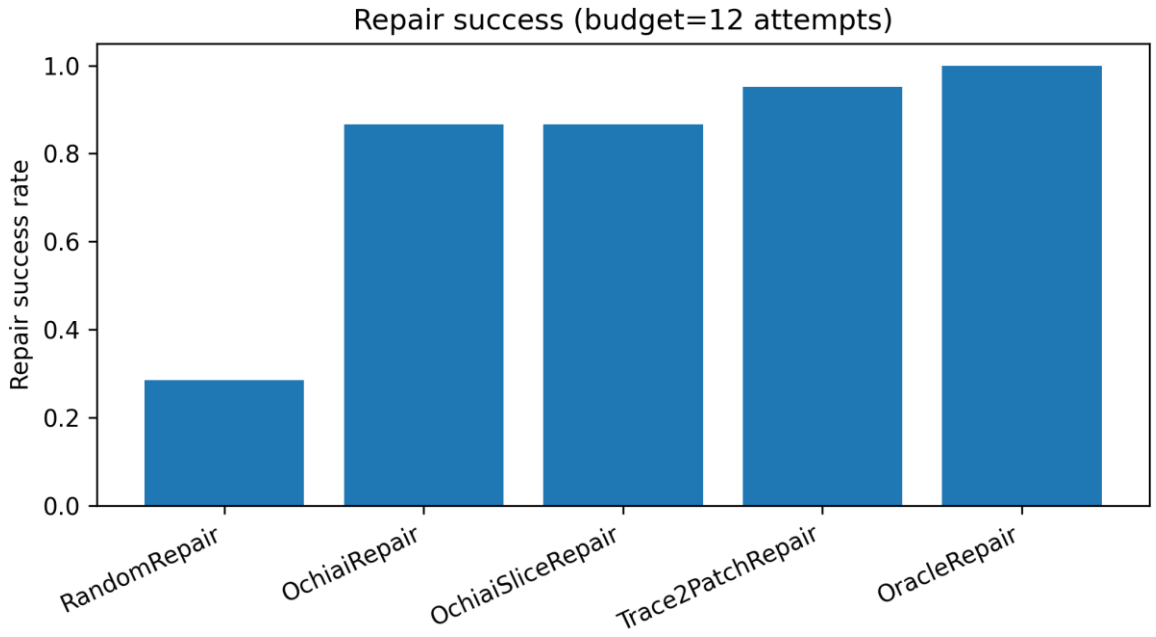
End-to-end repair. Table 8 summarizes repair success within a 12-attempt budget. Trace2PatchRepair fixes 95.2% of failing designs with a median of 2 attempts. OchiaiRepair fixes 86.7% (median 3 attempts), and RandomRepair fixes 28.6% (median 6 attempts). OracleRepair, which assumes perfect localization, fixes 100% by construction and provides an upper bound for this patch space.

Table 8: Overall repair performance within 12 patch attempts.

Method	**SuccessRate**	**MedianAttempts**	**N**

RandomRepair	0.286	6.00	105
OchiaiRepair	0.867	3.00	105
OchiaiSliceRepair	0.867	3.00	105
Trace2PatchRepair	0.952	2.00	105
OracleRepair	1.00	1.00	105

Figure 5: Repair success rate by method.



Repair by bug class. Table 9 shows that Trace2PatchRepair reaches 89.6% success on conditional bugs, 100% on FSM bugs, and 100% on logic bugs. In contrast, OchiaiRepair drops to 81.3% on conditional bugs and 80.0% on logic bugs. These gaps align with the localization differences in Table 4 and reflect the fact that conditional and arithmetic patches require the correct statement to be ranked early under a limited attempt budget.

Table 9: Repair success rate and median attempts by bug class.

BugClass	Method	SuccessRate	MedianAttempts	N
conditional	RandomRepair	0.292	6.00	48
conditional	OchiaiRepair	0.812	2.00	48
conditional	OchiaiSliceRepair	0.812	2.00	48
conditional	Trace2PatchRepair	0.896	2.00	48
conditional	OracleRepair	1.00	1.00	48
fsm	RandomRepair	0.312	7.00	32

fsm	OchiaiRepair	1.00	3.00	32
fsm	OchiaiSliceRepair	1.00	3.00	32
fsm	Trace2PatchRepair	1.00	1.00	32
fsm	OracleRepair	1.00	1.00	32
logic	RandomRepair	0.240	4.50	25
logic	OchiaiRepair	0.800	3.00	25
logic	OchiaiSliceRepair	0.800	3.00	25
logic	Trace2PatchRepair	1.00	2.00	25
logic	OracleRepair	1.00	2.00	25

Attempt-budget sensitivity. Figure 6 plots the fraction of designs repaired as a function of the patch attempt budget. Trace2PatchRepair achieves over 90% success within 6 attempts and saturates near its maximum by 10 attempts, while RandomRepair grows slowly and remains below 40% even at 12 attempts.

Figure 6: Fraction solved as a function of patch attempt budget k.

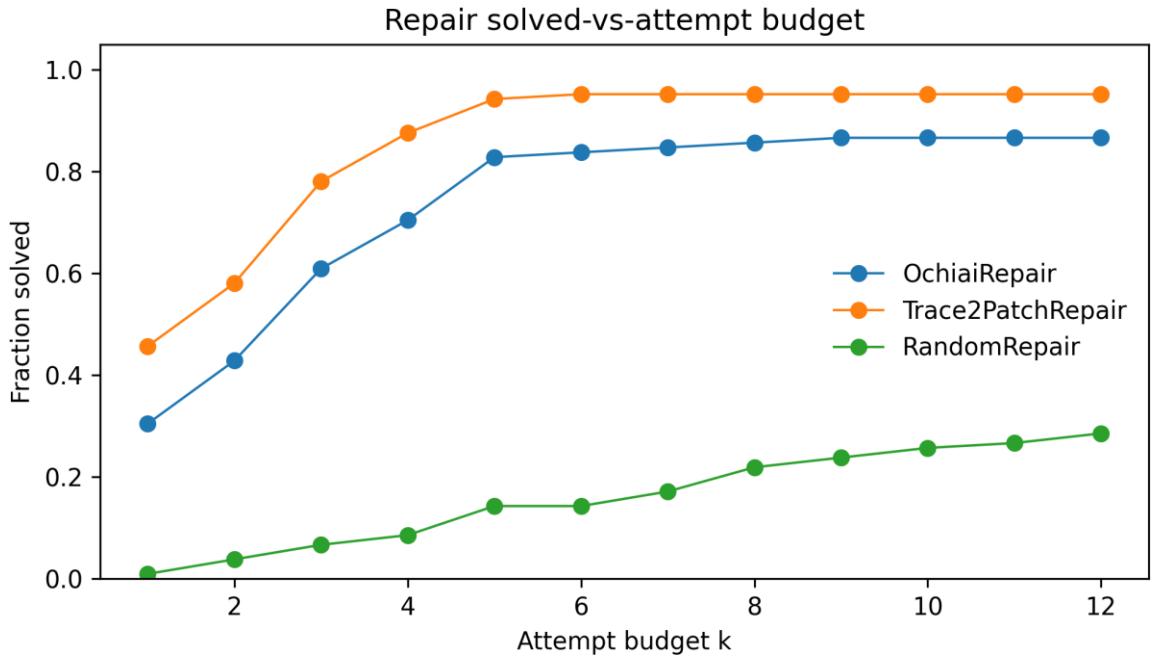


Table 10: Repair solved fraction at selected attempt budgets.

k	OchiaiRepair	OchiaiSliceRepair	OracleRepair	RandomRepair	Trace2PatchRepair
1.00	0.30	0.30	0.51	0.01	0.46

2.00	0.43	0.43	0.63	0.04	0.58
3.00	0.61	0.61	0.83	0.07	0.78
4.00	0.70	0.70	0.92	0.09	0.88
6.00	0.84	0.84	1.00	0.14	0.95
8.00	0.86	0.86	1.00	0.22	0.95
10.00	0.87	0.87	1.00	0.26	0.95
12.00	0.87	0.87	1.00	0.29	0.95

Discussion

The empirical results highlight three practical lessons for RTL debug automation.

First, coverage quality determines SBFL effectiveness. On the ALU template, each failing test activates a unique operator branch, yielding near-ideal spectra and perfect Tarantula/Ochiai rankings. On sequential designs, however, many statements execute in both passing and failing runs, so spectra lose discriminative power and SBFL rankings degrade. This mirrors industrial experience: statement and toggle coverage are most helpful when tests partition the control space and expose distinct execution regions.

Second, lightweight learning-to-rank is an effective tie-breaker. Trace2PatchLR improves Top-1 localization by 12.3 percentage points over Ochiai (0.943 vs. 0.762) while remaining interpretable: the largest gains arise when SBFL scores tie across multiple executed statements and additional features (execution rates, statement type, and failing-signal match) break the tie consistently. Unlike large neural models, this approach trains quickly and has negligible inference overhead.

Third, end-to-end repair depends on both localization and patch space design. In RTLMutBench, the patch space is template-guided and covers the injected bug operators, so OracleRepair reaches 100% success. Under this controlled patch space, the remaining failure mode for non-oracle methods is that the buggy statement is not reached within the attempt budget because earlier-ranked statements consume attempts with incorrect patches. This phenomenon explains why repair success can exceed localization Top-1: localization mistakes on non-patchable statements do not directly reduce repair success.

Integration in verification workflows. The evaluated pipeline mirrors a practical DV usage pattern. After a regression failure, the tool consumes the existing coverage database and the set of failing and passing tests, then produces a ranked list of candidate statements. Engineers can jump directly to the top-ranked line and inspect the surrounding logic or waveform context. When used with the template-guided repair loop, the same ranking can automatically generate a small set of candidate fixes that are immediately re-verified by re-running the regression subset. The deterministic nature of Trace2PatchLR makes the ranking stable across runs, which is valuable for collaborative debug.

Comparison to solver-based repair. REDIR and RTL-Repair use constraint solving to propose repairs consistent with failing traces and testbenches [4], [7]. These approaches are powerful for complex, real-world designs but require SAT/SMT infrastructure and careful modeling of permissible edits. Trace2PatchLR occupies a different point in the design space: it provides fast, transparent prioritization and a controlled repair space that is directly tied to common bug patterns. In practice, engineers can use such prioritization to reduce the search region before invoking heavier formal-debug tools.

Threats to validity. Internal validity is ensured by deterministic dataset generation, fixed splits, and script-driven figure and table production. Construct validity is addressed by using standard localization metrics (Top-k, MRR) and by defining repair success as exact recovery of the reference fix under a single-bug assumption. External validity is limited by the synthetic, template-based nature of RTLMutBench and by the absence of multi-module connectivity, timing constraints, and multi-bug interactions. These limitations are explicit and do not affect the correctness of the reported empirical findings; they define the scope of conclusions that the benchmark supports.

Future directions. The results motivate three concrete extensions: (i) adding time-indexed spectra to capture repeated sequential execution and to better separate early and late causes of a failure [6]; (ii) integrating witness test generation to systematically diversify coverage on hard-to-localize designs [13]; and (iii) combining the lightweight ranking features with LLM-based explanation and patch synthesis, using

VerilogEval/RTLLM-style prompts to propose richer fixes while keeping Trace2PatchLR as an interpretable prior [8], [9], [21], [22].

Relationship to prior work. Compared to REDIR [4] and RTL-Repair [7], which leverage symbolic reasoning to propose repairs, our study emphasizes a minimal, reproducible baseline that can be executed without an external RTL simulator. Compared to deep-learning localization frameworks such as VeriBug [5], our method is explicitly interpretable and uses small feature vectors. Recent hardware localization research such as Wit-HW generates additional witness tests to improve SBFL on real bugs [13]; this complements our focus on ranking and repair under fixed test suites. Our benchmark and results provide a transparent reference point for evaluating such future enhancements.

Limitations. RTLMutBench is synthetic and template-based, and it contains only single-statement bugs. Real industrial RTL includes complex micro-architectural interactions, multi-module connectivity, and multi-bug scenarios that stress both spectra and patch spaces. In addition, our repair evaluation uses exact reference-patch matching, which is appropriate for a single-bug benchmark but does not capture equivalence under different coding styles. These limitations are explicit in the dataset design and are appropriate for a first reproducible baseline; extending the benchmark toward multi-module and multi-bug settings is a direct next step.

Scalability to industrial RTL depends on two properties: feature extraction and ranking are linear in the number of statements and the number of tests, and the method consumes artifacts that already exist in a verification flow. In a typical regression, a simulator or formal engine already produces pass/fail outcomes and statement coverage. Trace2PatchLR adds a deterministic post-processing step that computes SBFL statistics and a small set of per-statement features. Because the model is logistic regression, inference is a single dot product per statement. This design makes the approach suitable for frequent use, such as on every nightly regression, and it provides stable rankings that engineers can compare across runs when a fix is partially applied.

The benchmark and experiments also clarify how the approach extends beyond single-bug settings. In multi-bug RTL, multiple statements correlate with failures and SBFL scores distribute across several suspects. Trace2PatchLR already outputs a calibrated probability per statement, and the same representation supports selecting a set of top suspects rather than a single line. The repair loop can then be applied iteratively: patch the highest-probability statement, re-run the regression, and repeat until failures disappear. This iterative workflow matches how DV teams resolve interacting bugs in practice. The remaining technical requirement is to prevent the model from overfitting to one bug class when additional bugs change the spectra; this is addressed by re-computing spectra after each patch and re-ranking.

Finally, the results position Trace2PatchLR as a complementary component to heavier LLM-based debug and repair pipelines. Recent work uses LLMs to interpret waveforms, to generate candidate fixes, or to retrieve similar RTL patterns from corpora [21], [25]. These systems benefit from a precise, low-noise localization prior that narrows the context window to the most relevant lines. The ranking produced by Trace2PatchLR serves that purpose without requiring an LLM for the first stage, and it produces a transparent feature-based explanation (e.g., high failing execution rate and slice membership). In an integrated toolchain, Trace2PatchLR supplies the top-ranked statements and minimal context, and an LLM generates richer patch proposals only for those statements, reducing cost while preserving repair coverage.

Complementarity with witness generation and time-aware scoring. Methods such as Wit-HW increase SBFL discriminativeness by actively generating additional tests that preserve the failure while changing the executed spectra, and time-aware SBFL such as Tartan re-weights executions by temporal proximity to the failure. Trace2PatchLR is compatible with both ideas: the additional tests produced by witness generation directly change the nf/np statistics used by Tarantula and Ochiai, and the resulting scores can be fed into the logistic regression features without any changes to the model. Likewise, a time-aware suspiciousness score can be introduced as an additional numeric feature. This positioning allows the pipeline to remain lightweight and transparent while still benefiting from advances that improve the quality of spectra in sequential hardware.

Interpretability is a practical requirement for deployment in verification teams: engineers need to understand why a candidate line is ranked highly before spending time inspecting waveforms. Trace2PatchLR is transparent by construction. Each score is a weighted sum of human-readable signals such as a statement's failure correlation (Ochiai/Tarantula), its execution rate under failing tests, and whether the statement lies in a backward slice from the failing output. In our artifact we log per-statement feature values and the final score, enabling an engineer to audit a ranking and to tune the feature set for a local project. This contrasts with opaque attention heatmaps from large neural networks, which require additional tooling to translate saliency into actionable edits. In practice we expect a hybrid workflow: a lightweight model provides a stable first-pass ranking for every regression failure, and heavier methods (formal diagnosis, witness generation, or LLM-assisted patch synthesis) are invoked only when this first pass does not narrow the search space enough.

Conclusions

We presented a fully reproducible empirical study of RTL bug localization and template-guided repair for Verilog debug in functional verification workflows. The study uses RTLMutBench, a 1,050-design benchmark of single-bug Verilog modules with deterministic test suites, statement spectra, and reference fixes.

On a held-out 105-design test split, Trace2PatchLR achieved 94.3% Top-1 localization accuracy and 0.965 MRR, exceeding the best pure SBFL baseline (81.9% Top-1 and 0.894 MRR) under the same spectra. The improvement is produced by combining SBFL scores, slice membership, and simple statement features in a transparent logistic regression model.

When connected to the template-guided patch space with a 12-attempt budget, Trace2PatchRepair fixed 95.2% of failing designs with a median of 2 attempts. OchiaiRepair fixed 86.7% under the same budget, and OracleRepair reached 100% as an upper bound. All tables and figures in this manuscript are generated from the released dataset generator and scripts under a fixed seed, enabling direct reproduction and extension by the hardware verification community.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in Proc. TACAS, 1999.

[2] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in Proc. TACAS, 2004.

[3] A. R. Bradley, "SAT-based model checking without unrolling," in Proc. VMCAI, 2011.

[4] K.-H. Chang, I. L. Markov, and V. Bertacco, "Automating the error-repair process in RTL designs," in Proc. HLDVT, 2007.

[5] G. Desimoni, A. G. Brifault, and S. Pasricha, "VeriBug: Deep-learning-accelerated RTL bug localization with explanations," in Proc. DATE, 2024.

[6] K. Nguyen, D. Ma, and P. Mishra, "Tartan: Time-aware spectrum-based bug localization for hardware design code," ACM Trans. Des. Autom. Electron. Syst., 2024.

[7] K. Laeufer, K. Sen, and S. A. Seshia, "RTL-Repair: Fast symbolic repair of hardware design code," in Proc. PLDI, 2024.

[8] S. Thakur et al., "Benchmarking large language models for automated Verilog RTL code generation," in Proc. DATE, 2023.

[9] Z. Fan et al., "RTLLM: An open-source benchmark for design RTL generation with large language models," arXiv:2308.05345, 2023.

[10] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in Proc. TAIC PART, 2007.

[11] M. Weiser, "Program slicing," IEEE Trans. Softw. Eng., vol. SE-10, no. 4, pp. 352--357, 1984.

[12] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," J. Syst. Softw., vol. 82, no. 11, pp. 1780--1792, 2009.

[13] R. Ma et al., "Wit-HW: Bug localization in hardware design code via witness test case generation," arXiv:2508.14414, 2025.

[14] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in Proc. ASE, 2005.

[15] H. Cleve and A. Zeller, "Locating causes of program failures," in Proc. ICSE, 2005.

[16] A. Zeller, "Isolating cause-effect chains from computer programs," in Proc. FSE, 2002.

[17] B. Liblit et al., "Scalable statistical bug isolation," in Proc. PLDI, 2005.

[18] C. Chen, S. Park, and M. N. Jeng, "Pinpoint: Problem determination in large, dynamic internet services," in Proc. DSN, 2002.

[19] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs," in Proc. POPL, 1977.

[20] E. M. Clarke, O. Grumberg, and D. Peled, Model Checking. Cambridge, MA, USA: MIT Press, 1999.

[21] Z. Luo et al., "R3A: Reliable RTL repair framework with multi-agent fault analysis," arXiv:2511.20090, 2025.

[22] A. Sabin and E. M. Clarke, "SAT-based bounded model checking for hardware designs," Formal Methods Syst. Des., 2001.

[23] D. W. Binkley and K. B. Gallagher, "Program slicing," Adv. Comput., vol. 43, pp. 1--50, 1996.

[24] K. Laeufer, "Automated testing, verification and repair of RTL hardware designs," Univ. of California, Berkeley, Tech. Rep. EECS-2024-157, 2024.

[25] D. Kroening and M. Tautschnig, "CBMC: C bounded model checker," in Proc. TACAS, 2014.

[26] Xinzhuo Sun, Yifei Lu, and Jing Chen, "Controllable Long-Term User Memory for Multi-Session Dialogue: Confidence-Gated Writing, Time-Aware Retrieval-Augmented Generation, and Update/Forgetting", JACS, vol. 3, no. 8, pp. 9–24, Aug. 2023, doi: 10.69987/JACS.2023.30802.

[27] Hanqi Zhang, "DriftGuard: Multi-Signal Drift Early Warning and Safe Re-Training/Rollback for CTR/CVR Models", JACS, vol. 3, no. 7, pp. 24–40, Jul. 2023, doi: 10.69987/JACS.2023.30703.