LEARNING A SYMBOLIC DECOMPILER

Anonymous authors
Paper under double-blind review

ABSTRACT

Decompilation reverses the transformations of compilers. These transformations are deterministic but non-bijective, often discarding names, types, and structure from high-level languages. Traditional decompilers infer plausible high-level information through heuristic analyses that require expert labor to write and maintain. Recent work instead fine-tunes large language models to generate source code directly or uses LLM agents to iteratively refine the output of existing decompilers. Both approaches make decompilation knowledge hard to accumulate and inspect: neural models encode it implicitly in weights, where individual behaviors are hard to trace or repair; agents typically reason about each program independently, leaving no residue for future tasks. Symbolic rules offer complementary properties: they are inspectable, traceable, precisely repairable, and reusable across programs, but have traditionally been difficult to learn autonomously. To address this, we present RuleCrafter, an agentic framework that decomposes decompilation into substeps and learns Datalog rules for each. Without LLM calls at evaluation, the best learned rule base matches or exceeds traditional decompilers in re-execution, and rule bases learned on one corpus apply to the others without modification.

1 INTRODUCTION

Decompilers are systems which take object files (*e.g.*, binaries or bytecode) and produce an equivalent source-level representation. Decompilation and related binary-lifting techniques support reverse engineering (Schwartz et al., 2013), binary instrumentation (Altinay et al., 2020), and malware analysis (Yakdan et al., 2016). However, production decompilers (such as IDA Pro with Hex-Rays (Hex-Rays, 2024), Ghidra (National Security Agency, 2019), and Binary Ninja (Vector 35, 2024)) are complex software artifacts that combine control-flow reconstruction, type recovery, and code generation (Schwartz et al., 2013; Yakdan et al., 2016). This complexity arises because decompilers must produce correct and *readable* code while dealing with the ambiguity introduced by the compilation process.

We introduce a new methodology for *learning* rule-based decompilers. Learning has been applied with increasing frequency to tackle subproblems in decompilation. For example, DIRE uses a neural approach to decompiled identifier naming (Lacomis et al., 2019), illustrating how learning can productively harmonize with an otherwise-symbolic decompiler: the learning guides correctness-invariant decisions (variable naming) in an otherwise-explainable process.

Recent interest has shifted to LLM-guided decompilation (Tan et al., 2024; Hu et al., 2024), wherein a language model reconstructs source code or refines decompiler output. These LLM-based methods are attractive due to their ability to produce readable code. However, applying an LLM to each input incurs repeated inference cost, and its reconstruction decisions are less explicit than those of a rule-based decompiler. Recompilation can guide iterative repair, but successful compilation alone does not establish behavioral equivalence (Wong et al., 2025).

We present RuleCrafter, an approach that learns a reusable, rule-based procedure for reconstructing C directly from x86-64 assembly. Decompilation steps that procedure requires constructing analyses works together: recovered types inform expressions, memory accesses constrain assignments, and control-flow analysis determines loops. A failure in the resulting C does not supply the correct implementation of any one analysis. Datalog provides a useful representation for this task: program properties are relations, clauses derive new facts from them, and recursive inference expresses anal-

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.

yses such as reachability and dominance. Its use in program analysis and decompilation is already established (Smaragdakis & Bravenboer, 2010; Flores-Montoya & Schulte, 2020; Liu et al., 2026a).

We investigate training executable Datalog programs: *the LLM is the learner, while the rule base is the learned decompiler*. RuleCrafter decodes x86-64 instructions from ELF files into relational facts and supplies these facts to workers together with stage interfaces, concise task guidelines, and a C AST grammar. Starting from an empty rule base, workers implement analyses and reconstruction rules, introducing intermediate relations and pure Rust helpers as needed. A worker examines generated C and its diagnostics, traces an output fact to the clauses that produced it, and revises those clauses. The revised base can then be tested on other training programs. Fixed relational interfaces let the implementation grow while preserving how its stages compose. Section 2 illustrates reconstruction with a schematic loop example.

The learned procedure spans the path from decoded instructions to C. The fixed loader supplies instruction operands, register and flag effects, symbol and relocation evidence, and the machine's calling convention. Learned rules recover basic blocks and control-flow edges, track stack locations and register definitions, infer operations and C signatures, and construct and restructure candidate ASTs. The supplied execution machinery evaluates these rules, selects declared alternatives through MaxSMT, and validates and prints the resulting AST.

We use GPT-6 Astra, Opus 5.5, and DeepSeek V4.1 Flash as rule-learning workers. Table 2 compares the resulting RuleCrafter rule bases with neural refiners and traditional decompilers on HumanEval, MBPP, and ExeBench. RuleCrafter uses model calls to develop an executable reconstruction procedure that can then run on new inputs without the model.

Our contributions are:

1. **Learning a symbolic decompiler from program feedback.** LLM workers build reusable Datalog rules from an empty base, guided by program feedback and provenance.
2. **Competitive reconstruction with no model calls at evaluation.** The best rule base competes with established decompilers in re-execution on MBPP and ExeBench.
3. **Learned rules transfer across corpora.** Rule bases learned on HumanEval, MBPP, or ExeBench reconstruct programs from the other corpora without edits.

2 Motivation and End-to-End Example

Feedback on a failed decompilation should improve the logic used on future inputs. Repairing one C output leaves the analyses that produced it unchanged. Learning that logic is demanding: traditional decompilers combine algorithm design, reconstruction heuristics, and shared program representations, and revisions must preserve how these parts work together.

Symbolic programs expose this logic as executable clauses over program facts. Decoded instructions, recovered control-flow edges, and analysis results share a relational representation; rules connect them to reconstructed C syntax. RuleCrafter supplies the execution engine and stage interfaces, allowing the LLM to add and revise analyses within a stable architecture. For example, a worker can extend loop-membership analysis while preserving the relation consumed by reconstruction. Datalog provenance links output to producing clauses, identifying definitions to inspect when reconstruction fails.

LLM workers generate the rules from an initially empty rule base. They receive training inputs and interfaces specifying each substep's task, readable inputs, owned outputs, and vocabulary relations in which facts may be stated. The input facts describe decoded instructions and operands, registers, flags, data, and ABI locations. Basic blocks, control-flow edges, and loops must be derived by learned rules. Further rules choose C declarations, expressions, and assignments, preserve memory-effect ordering, and organize branches into control structures. Workers propose clauses and intermediate relations, execute the rules, and inspect derived facts, generated C, or failure diagnostics.

The loop in Figure 1(a–c) illustrates the logic to be learned: identifying the blocks that repeat and preserving the exit test's position. Assume that `x` contains a `'/'` within its readable bounds and that `num` is a separate buffer large enough for the preceding characters. Panel (a) shows a schematic CFG

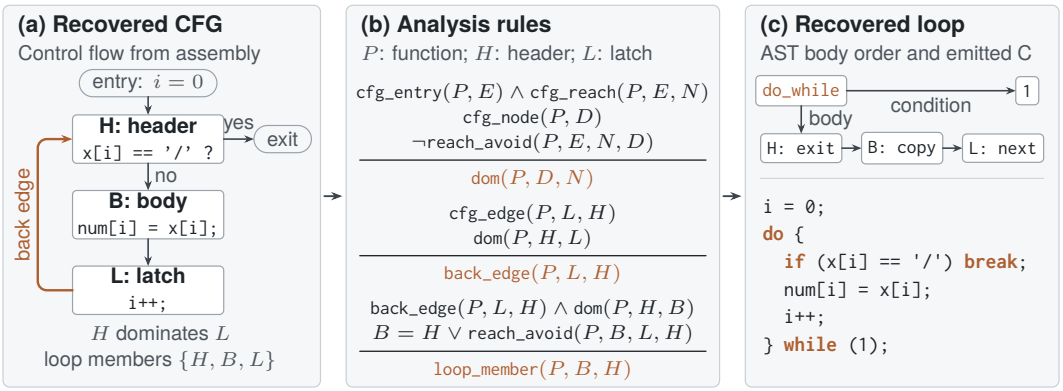


Figure 1: Schematic loop reconstruction (§2). (a) Recovered control-flow graph (CFG) with an exit test at the loop header; the loader supplies instructions, and learned rules derive the CFG. (b) Illustrative auxiliary dominance, back-edge, and loop-membership clauses (¬: Datalog negation); derived heads are shown in orange. (c) AST body order and corresponding C, with the exit guard before the copy and index update.

recovered from assembly: entry initializes `i` to zero; H tests for the delimiter and exits if found. Otherwise, B copies the character and L increments the index before returning to H.

Panel (b) uses illustrative auxiliary relations: `cfg_reach` is reflexive reachability, and `reach_avoid`(P, U, V, D) holds when a path from U to V avoids D, including at its endpoints. A zero-length path is allowed when $U = V \neq D$. The first rule derives dominance (`dom`), including self-dominance, by ruling out entry paths that avoid a candidate dominator. Thus H dominates L, and the edge $L \to H$ is a back edge. The membership rule includes H and reachable blocks that can reach L without revisiting H; its disjunction abbreviates two Datalog clauses. Zero-length reachability includes L, yielding exactly $\{H, B, L\}$ in this example. These relations are illustrative choices available to workers, not prescribed analyses.

Restructuring rules use these control-flow relationships to organize an initially unstructured C abstract syntax tree (AST) fragment into the loop sketched in panel (c). Its body places the header's test and exit guard before the copy and index update. The `do-while (1)` form preserves this order: when `x[0]` is `'/'`, it exits without copying or incrementing. Moving the guard after the copy would incorrectly write the delimiter. The fragment copies only the prefix and adds no null terminator. Panel (c) shows the intended body order and corresponding C; AST construction, restructuring, selection, and printing are detailed in Section 3.1.

Figure 2 shows the assembly-to-C pipeline and its learning architecture: dashed arrows show feedback and rule edits, while solid arrows show execution on new inputs without LLM calls.

3 Architecture and Learning

RuleCrafter reconstructs C from x86-64 assembly decoded from ELF inputs. The learned decompiler is a rule base $\mathcal{R}$ containing Datalog clauses, vocabulary facts, and pure embedded Rust functions. Its supplied architecture Γ fixes the machine description, assembly loader, substep interfaces, C AST grammar, evaluation schedule, MaxSMT selector, and emitter. Learning changes the rules within these boundaries. We instantiate the LLM worker policy with GPT-6 Astra, Opus 5.5, or DeepSeek V4.1 Flash; the model parameters remain fixed.

The trainable rule base starts at $\mathcal{R}_0 = \varnothing$. Workers generate the initial rules from the supplied interfaces and assembly facts, then add, revise, or delete entries using emission diagnostics, Clang errors, corpus test outcomes, KLEE comparisons and counterexamples, and R2I readability scores. Each published batch changes the same shared program, from which a checkpoint is selected for evaluation.

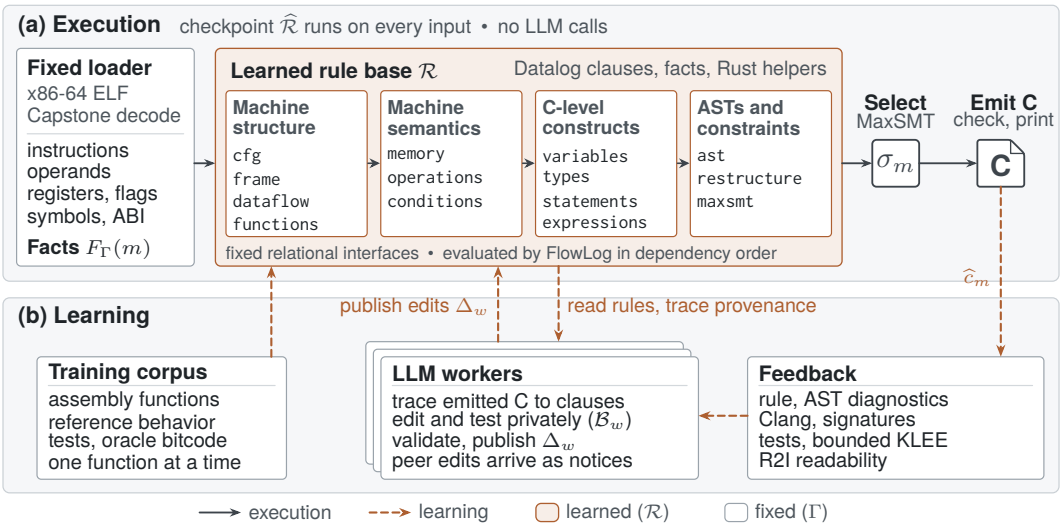


Figure 2: RuleCrafter overview. (a) Execution on one function: a fixed loader decodes an ELF into facts; the learned rule base $\mathcal{R}$, fourteen substeps in dependency order (§3.3), derives candidate ASTs and selection constraints; a fixed MaxSMT solver selects among declared alternatives under the learned MaxSMT constraints, and a fixed emitter checks the selected graph and prints C with learned spelling tables (§3.1). Tinted boxes are learned; white boxes are supplied and fixed. The loader supplies no basic blocks, edges, or loops: CFG recovery is learned. (b) Learning: LLM workers run training functions through the same pipeline, trace emitted C to the clauses that produced it, and publish rule edits to the shared program using feedback on the output (§3.2). A checkpoint of $\mathcal{R}$ reconstructs new inputs without LLM calls.

3.1 From assembly to C

The fixed loader uses Capstone (Quynh, 2014) for structured decoding of instructions, operands and widths, explicit and implicit register accesses, and flag effects. The loader also exposes symbol, relocation, and data evidence. A machine description supplies the x86-64 Linux System V register file and ABI vocabulary. The loader supplies no basic blocks, CFG edges, or loop annotations. For assembly-text inputs, comments and unwind annotations are removed before assembly and decoding. The ELF contract requires bounded function symbols, retained data symbols, and, for linked executables, retained static relocations; unsupported encodings or relocations yield explicit refusals.

Each target function is reconstructed separately, with direct callees and referenced storage available as context. Let m denote this scoped assembly input and $F_\Gamma(m)$ its finite fact base. With Γ fixed, each $\mathcal{R}$ defines a reconstruction procedure $\mathcal{D}_\mathcal{R}$:

$$\begin{aligned} E_m &= \mathrm{Eval}_{\Gamma,\mathcal{R}}(F_\Gamma(m)), \\ \sigma_m &= \mathrm{Select}_\Gamma(E_m), \\ \widehat{c}_m &= \mathcal{D}_\mathcal{R}(m) = \mathrm{Emit}_\Gamma(E_m, \sigma_m). \end{aligned} \tag{1}$$

The FlowLog engine (Zhao et al., 2025; Yu et al., 2026) evaluates substeps in dependency order, normally to a stratified fixed point. AST construction derives complete candidate graphs as `ast_node`/`ast_edge` facts. Restructuring repeatedly derives successor node and edge snapshots; a fixed driver compares them for exact convergence and publishes `c_ast`/`c_ast_edge`. These graphs and the learned constraint rows form part of E_m.

After Datalog evaluation, one MaxSMT selection phase produces σ_m. Substeps declare decision variables in relations such as `ast_decision`, pairing them with candidate values in `ast_alternative`; the selector recognizes these relation suffixes across substeps. The fixed boundary enforces the declared domains, while learned `maxsmt` rules supply hard consistency constraints and weighted soft preferences. A candidate with no declared alternative is included unconditionally, so a single candidate need not supply any constraint rows. Unconstrained choices are

reported as such. The solver selects among rule-produced candidates; it does not establish their semantic correctness.

The fixed emitter validates the selected graph and prints C using learned spelling tables and the supplied grammar. It checks graph structure and requires exactly one selected instruction-origin edge for every target instruction. Calls, control transfers, and program-visible writes cannot be discharged by assigning their origins to the function root; private frame writes require supporting frame evidence. Every cast must already be present in the learned AST. Refusals return diagnostics; an accepted AST can still produce incorrect C, whose behavior is assessed during training.

3.2 Learning from feedback

The training corpus pairs assembly inputs with reference behavior and checking procedures. For each input m, workers access the extracted facts $F_\Gamma(m)$. Relational and emission diagnostics identify broken interfaces or AST structure; compiler diagnostics expose invalid C; corpus tests expose incorrect concrete executions. The KLEE (Cadar et al., 2008) comparison module checks candidate signatures and searches for differences in return values and modeled memory effects. Its reference is LLVM bitcode compiled from the reference C; this bitcode remains outside the workers' input facts and queries. R2I (Eom et al., 2024) supplies readability feedback. The workers use these separate reports to choose actions, without a scalar reward combining them.

Feedback is derived from the reconstructed programs, while an update changes the rules that produced them. To connect the two, workers inspect generated C, query intermediate relations, and trace an output line or fact to its producing clauses. A trace identifies derivations to inspect, including upstream dependencies; the worker proposes an edit and requests new feedback to assess its effect. Testing other training functions then exposes improvements and regressions wherever the edited clauses apply.

The current learning interface presents one training function at a time. `try_rule` evaluates a private edit batch, `try_emit` exposes emitted C (optionally an individual substep's graph), and `try_oracle` and `try_r2i` inspect its behavior and readability. `check` evaluates the current function on the shared base. The worker advances when C is emitted and compiled, no signature mismatch is reported, available shipped tests pass, and scored readability reaches the configured threshold, or when its check budget is exhausted. KLEE's bounded verdict is reported alongside this decision and does not gate advancement. Advancing a function therefore does not establish behavioral equivalence.

3.3 Substeps and shared rule learning

Fourteen substeps recover machine structure and values (1–7), typed statements and expressions (8–11), and C ASTs and selection constraints (12–14), in dependency order (Figure 2(a)). Appendix B lists their interfaces. Each declares readable inputs, owned outputs, vocabulary facts, and a task guideline; the AST interface also supplies the printer's graph grammar. Workers choose intermediate analyses, auxiliary relations, and pure Rust helpers. Ground facts are restricted to declared vocabulary, and publication rejects guarded enumerations of input-specific program text.

Raw instruction details reach substeps 1–7. Variables and statements receive instruction identities; AST construction receives symbol and literal metadata plus upstream results. Types, expressions, restructuring, and MaxSMT receive no raw assembly facts.

The contracts distinguish whole written values (`value`) from bit-range evidence (`value_part`, `use_site`), and ABI recovery from C signature inference, with unresolved choices recorded in `signature_ambiguity`. Semantic effects remain separate from instruction ownership: assigning one owner does not discard an instruction's effects.

Workers implement assigned substeps using their training input, tool feedback, and peer notices. Published edits are visible to all workers; the shared rule base persists when interaction histories are summarized. Let $\mathcal{R}_t$ denote this base after t published batches. Worker w can test a private candidate $\mathcal{B}_w$ formed from batch Δ_w and snapshot $\mathcal{R}_s$; publication applies the batch atomically to

Table 1: Learned program size and cumulative rule edits by model and training corpus.

Model	Training corpus	Clauses	Invented relations	Rust helpers	Rule edits
GPT-6 Astra	HumanEval	2,211	582	6	2,497
	MBPP	1,636	498	7	2,024
	ExeBench	3,402	940	5	3,660
Opus 5.5	HumanEval	1,710	468	1	2,115
	MBPP	1,342	357	1	1,515
	ExeBench	2,211	650	3	2,502
DeepSeek V4.1 Flash	HumanEval	2,228	578	4	2,754
	MBPP	3,563	919	4	4,110
	ExeBench	3,772	1,190	5	4,378

the current base:

$$\mathcal{B}_w = \mathrm{Apply}(\mathcal{R}_s, \Delta_w), \tag{2}$$

$$\mathcal{R}_{t+1} = \mathrm{Apply}(\mathcal{R}_t, \Delta_w), \qquad s \le t. \tag{3}$$

Each published addition, revision, or deletion counts as one *rule edit*; an entry may contain clauses, vocabulary facts, or helper code.

Publication checks rule and interface consistency; promotion also checks trial completion and emission regressions. Compilation and test success are not publication requirements, and trials made stale by concurrent edits are not automatically rerun.

Evaluation fixes a checkpoint $\widehat{\mathcal{R}} = \mathcal{R}_{t^\star}$ containing the active clauses, vocabulary facts, and helpers. It runs as $\mathcal{D}_{\widehat{\mathcal{R}}}$ on new assembly inputs without the LLM or worker histories.

4 Evaluation

We evaluate on HumanEval, MBPP, and ExeBench and ask three questions: how RuleCrafter compares with existing decompilers (§4.1), how feedback leads to a rule repair during learning (§4.2), and how learned rule bases transfer across corpora (§4.3). We measure readability with R2I (Eom et al., 2024), a relative metric on a 0–100 scale, and correctness with re-execution (R-E), the percentage of cases whose decompiled C passes the corpus's tests. The validation experiments execute fixed rule bases without LLM calls; the repair case study examines a recorded learning interaction. Table 1 reports the nine learned rule bases' sizes and cumulative rule edits; Appendix A gives training-set sizes and model usage and cost figures.

4.1 Decompiler comparison

Table 2 compares RuleCrafter with neural refiners and traditional decompilers, including angr (Shoshitaishvili et al., 2016; Basque et al., 2024) and Kuna (Noelo Lab, 2026). For RuleCrafter, it reports each model's rule base learned on ExeBench; §4.3 evaluates the rule bases learned on the other corpora. The traditional decompilers are evaluated at O0–O3, with R2I normalized over the common parseable inputs: 36, 210, and 4,962 for HumanEval, MBPP, and ExeBench.

Neural-system values follow a different protocol and serve as reference points. Their HumanEval and MBPP R-E and all R2I values are the O0–O3 averages reported by Tan et al. (2025, Tables 1 and 2), whose R2I uses its own normalization and assigns zero to parsing failures. That paper does not report ExeBench R-E, so we measure it for LLM4Decompile, Idioms, and SK^{2}Decompile, using Ghidra rather than IDA pseudocode for LLM4Decompile; GPT-5-mini's ExeBench R-E is unavailable ("–").

RuleCrafter's GPT-6 Astra rule base exceeds some traditional decompilers in re-execution on MBPP and ExeBench. It trails SK^{2}Decompile on all three; SK^{2}Decompile runs a 6.7B-parameter model on IDA pseudocode for every input, whereas RuleCrafter makes no model calls at evaluation. DeepSeek's rule base has higher R2I than any traditional decompiler on HumanEval and MBPP, but

Table 2: Decompiler comparison on HumanEval, MBPP, and ExeBench using R2I readability and re-execution (R-E) scores (0–100, higher is better). Neural R2I and HumanEval/MBPP R-E are from Tan et al. (2025); all other reported scores are from our evaluation.

		HumanEval		MBPP		ExeBench	
System	Input	R2I	R-E	R2I	R-E	R2I	R-E
Neural systems (model inference per input)							
GPT-5-mini	IDA C	42.33	56.75	44.14	47.23	45.25	–
LLM4Decompile	IDA/Ghidra C	68.71	41.71	69.19	43.05	57.02	49.8
Idioms	IDA C	53.10	29.81	55.73	24.01	60.40	46.5
SK^{2}Decompile (6.7B)	IDA C	**70.27**	**69.00**	**71.66**	**59.63**	**71.50**	**67.0**
Non-neural decompilers							
angr (9.2.102)	Native code	44.38	6.82	45.27	18.60	60.16	30.64
Ghidra (12.0.3)	Native code	48.55	37.88	53.28	38.50	62.04	45.13
Hex-Rays (8.3)	Native code	47.40	40.15	54.59	46.51	**73.89**	**47.63**
Kuna (v1.553)	Native code	50.99	**42.42**	53.88	39.02	62.45	42.34
GPT-6 Astra	x86-64 asm	33.16	41.67	36.19	**47.23**	47.27	44.53
DeepSeek V4.1 Flash	x86-64 asm	**78.43**	3.79	**76.79**	7.57	63.15	26.49
Opus 5.5	x86-64 asm	46.44	25.76	51.77	29.47	58.46	30.79

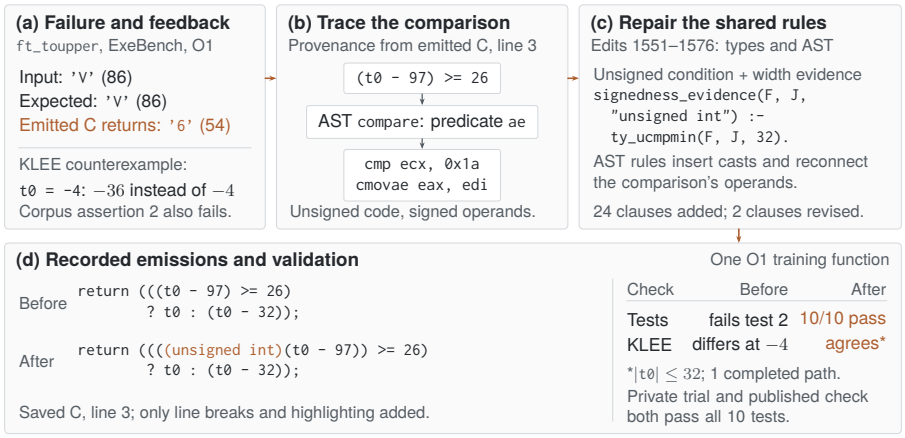


Figure 3: Recorded repair by the Opus ExeBench worker. The unsigned `cmovae` predicate survives reconstruction, but its C operands have signed type. Provenance identifies the comparison rules; a 26-edit batch adds width-based unsigned casts. The saved trial and published check pass the function's tests and agree with the reference within KLEE's stated bound. This trace measures one training function.

its output rarely passes tests. Across models, readability and correctness diverge: no RuleCrafter rule base leads on both.

4.2 Rule repair from feedback

Figure 3 follows a recorded repair by the Opus 5.5 worker on the ExeBench training function `ft_toupper`, compiled at O1. Its assembly uses an unsigned range test to distinguish lowercase letters from other inputs. The reconstructed C retains the comparison operator but uses signed operands: for input `'V'` (86), `t0 - 97` is negative, so the code incorrectly subtracts 32 and returns `'6'` (54). The corpus harness fails on this input, and KLEE independently finds a counterexample at `t0 = -4`.

The worker calls `trace_fact` on emitted line 3 and its comparison fact, then inspects the rules connecting the comparison's operands. The AST already carries the unsigned condition code `ae`, but its operands are plain `int` expressions. At rule base 1550, the worker tests 24 new clauses and two revisions: type rules recover the operand width and publish `signedness_evidence`,

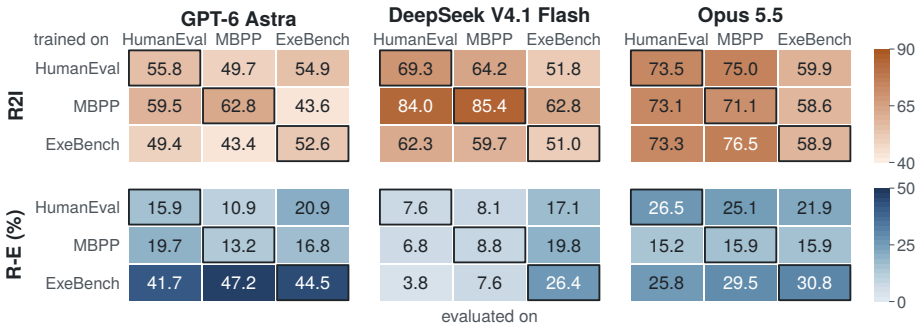


Figure 4: Cross-corpus transfer. Rows indicate the training corpus and columns the evaluation corpus; outlined cells use the same corpus for both. Top: R2I readability (0–100), using a different comparison pool from Table 2. Bottom: test suite pass rate (R-E, %).

while AST rules insert unsigned casts and reconnect the operands. The resulting C casts `t0 - 97` to `unsigned int`, preserving `'V'`. The worker publishes this batch as edits 1551–1576; the diagnostic and trial interval contains no intervening peer edit.

Both the private trial and the subsequent check of the published base compile without diagnostics and pass all ten recorded assertion evaluations. KLEE reports agreement on one completed path with $|\texttt{t0}| \leq 32$; the separate corpus tests cover upper- and lowercase characters outside this bound. These observations document the repair of one training function; Appendix C reproduces its assembly and complete saved C outputs. The following experiments examine reuse of complete rule bases across corpora.

4.3 Transfer across corpora

Figure 4 evaluates each model's three rule bases across all three validation corpora without rule edits. R2I is normalized within each model and evaluation corpus, whereas our scores in Table 2 use inputs parseable by all traditional decompilers.

We summarize transfer by the *in-distribution advantage*: on each corpus, the score of the rule base learned there minus the mean score of the two learned elsewhere, averaged over the three corpora. This balanced contrast cancels additive rule-base and corpus effects. For GPT-6 Astra, DeepSeek V4.1 Flash, and Opus 5.5, the advantages are $+7.0$, $+4.4$, and -1.6 R2I points (mean $+3.3$), and -1.6, $+3.7$, and $+2.2$ R-E points (mean $+1.4$), respectively.

Learning on the evaluation corpus need not yield the best rule base. On HumanEval, GPT-6 Astra's ExeBench base passes 41.7% of cases, versus 15.9% for its HumanEval base. Readability and correctness also transfer differently: its ExeBench base has the highest R-E on every corpus but the lowest R2I on HumanEval and MBPP. DeepSeek's in-distribution bases lead its R-E on every corpus, although all three pass fewer than 9% of HumanEval and MBPP cases. Each base comes from one learning run, so training-corpus effects remain confounded with run-to-run variation.

5 Related Work

Symbolic Learning with LLMs ASP systems refine initial theories with solver feedback and regression tests (Eiter et al., 2024; 2026), or learn rules from scratch that execute without further LLM calls (Ruiz et al., 2026). ReaComp compiles reasoning traces into reusable synthesizers (Naik et al., 2026). RuleCrafter learns cooperating decompilation stages with fixed relational interfaces, feedback on generated C, and rule provenance.

Reverse Engineering Prior work recovers control flow (Schwartz et al., 2013; Yakdan et al., 2015; Basque et al., 2024) and instructions (Bauman et al., 2018; Ye et al., 2023). Declarative analyses support disassembly and decompilation (Flores-Montoya & Schulte, 2020; Grech et al., 2019; 2022); Formulog combines Datalog and SMT for static analysis (Bembenek et al., 2020). HyRES adds

LLM predictions to Datalog for data-structure recovery (Sha et al., 2026); Superset Decompilation preserves candidates and provenance across declarative passes (Liu et al., 2026a).

Machine Learning for Decompilation SLaDe and LLM4Decompile-End translate assembly; LLM4Decompile-Ref refines pseudocode (Armengol-Estapé et al., 2023; Tan et al., 2024). Idioms predicts code and types; SK^{2}Decompile separates structural and identifier recovery (Dramko et al., 2025; Tan et al., 2025). DeGPT refines pseudocode; D-LiFT fine-tunes with correctness feedback (Hu et al., 2024; Zou et al., 2025). BED, DecLLM, and Decaf refine individual reconstructions using recompilation or execution (Schulte et al., 2018; Wong et al., 2025; Shypula et al., 2026). Constraint-guided agents and AutoDecompiler use compilation and tests (Zhang et al., 2026; Liu et al., 2026b). D-Helix checks decompilation symbolically; Alive2 validates LLVM translations within bounds (Zou et al., 2024; Lopes et al., 2021). RuleCrafter updates executable rules reused across inputs and uses bounded KLEE comparisons for feedback (Cadar et al., 2008).

6 Discussion and limitations

RuleCrafter makes decompilation process inspectable through executable rules and provenance, allowing repairs to change the procedure reused on later inputs. This reuse remains bounded by the supplied interfaces and execution machinery.

The current input contract covers x86-64 Linux System V ELF with retained symbols and relocations; stripped binaries and PIE are outside this contract. Assembly may not determine source-level parameter order across ABI classes, unused parameters, or types uniquely. The interfaces retain such ambiguity, while the oracle can refuse a proposed C signature before behavioral comparison. AST coverage and MaxSMT consistency do not establish semantic equivalence, and tests and bounded symbolic checks provide limited evidence.

RuleCrafter demonstrates that executable, symbolic Datalog rules for binary decompilation can be learned autonomously through feedback-guided LLM workers without relying on active model inference at evaluation time. By maintaining provenance and decoupling reusable analysis rules from input instances, the system produces explainable reconstructions that transfer across benchmarks and achieve re-execution performance competitive with established decompilers.

AI Use Disclosure

In this work, we used generative AI tools (Claude Code, Codex, and ChatGPT) for designing and refining the research methodology and experiments, implementing methods, and interpreting results. We did not use generative AI tools to develop the conceptual framework.

Synthetic dataset generation, dataset cleaning or reformatting, natural-language translation, formulating mathematical claims, qualitative or thematic data analysis, proposing or refining hypotheses, providing critical ingredients for proving mathematical claims, and assisting with proof writing are not applicable to this work. We use existing datasets directly and do not present mathematical proofs.

Additionally, we used generative AI tools for creating and revising scientific figures; developing and editing software; drafting the manuscript and improving its readability; finding, summarizing, and analyzing relevant literature; exploring research topics and gaps; retrieving information and relevant research literature; and suggesting the paper's structure, title, and wording.

We have reviewed all AI-assisted work. The experiments were run under human supervision, and we have inspected generated code to make sure it aligns with the design and correctly implemented. We used LLMs to refine text and writing, and have gone through manual checks. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

References

Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dąbrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, H. Bos, and M. Franz. *BinRec: dynamic binary lifting and recompilation*. 2020.

Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael F. P. O'Boyle. SLaDe: A Portable Small Language Model Decompiler for Optimized Assembly, 2023. URL https://arxiv.org/abs/2305.12520.

Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 361–378. USENIX Association, 2024. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/basque.

Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. 2018.

A. Bembenek, M. Greenberg, and Stephen Chong. Formulog: Datalog for smt-based static analysis. *Proceedings of the ACM on Programming Languages*, 4:1 – 31, 2020.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Operating Systems Design and Implementation (OSDI)*, pp. 209–224, 2008. URL https://www.usenix.org/legacy/event/osdi08/tech/full_papers/cadar/cadar.pdf.

Luke Dramko, Claire Le Goues, and Edward J. Schwartz. Idioms: Neural decompilation with joint code and type definition prediction. 2025.

Thomas Eiter, Jan Hadl, Nelson Higuera, and Johannes Oetsch. Declarative knowledge distillation from large language models for visual question answering datasets, 2024. URL https://arxiv.org/abs/2410.09428.

Thomas Eiter, Nelson Higuera Ruiz, and Johannes Oetsch. Distilling answer-set programming rules from llms for neurosymbolic visual question answering, 2026. URL https://arxiv.org/abs/2606.03269.

Haeun Eom, Dohee Kim, Sori Lim, Hyungjoon Koo, and Sungjae Hwang. R2i: A relative readability metric for decompiled code. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi: 10.1145/3643744. URL https://doi.org/10.1145/3643744.

Antonio Flores-Montoya and Eric Schulte. Datalog Disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1075–1092. USENIX Association, 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya.

Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *International Conference on Software Engineering (ICSE)*, pp. 1176–1186, 2019. doi: 10.1109/ICSE.2019.00120. URL https://yanniss.github.io/gigahorse-icse19.pdf.

Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. Elipmoc: Advanced Decompilation of Ethereum Smart Contracts. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1), 2022. doi: 10.1145/3527321. URL https://yanniss.github.io/elipmoc-oopsla22.pdf.

Hex-Rays. *IDA Pro and Hex-Rays Decompiler*. Hex-Rays SA, 2024. URL https://hex-rays.com/ida-pro/.

Peiwei Hu, Ruigang Liang, and Kai Chen. DeGPT: Optimizing Decompiler Output with LLM. In *Network and Distributed System Security Symposium (NDSS)*, 2024. URL https://www.ndss-symposium.org/ndss-paper/degpt-optimizing-decompiler-output-with-llm/.

Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 628–639, 2019.

Chang Liu, Yihao Sun, Thomas Gilray, and Kristopher Micinski. Superset Decompilation, 2026a. URL https://arxiv.org/abs/2603.28002.

Peipei Liu, Jian Sun, Mingzhe Xing, Yicheng Zeng, Zhaoteng Yan, Lixiao Zhang, Li Chen, and Dan Li. Binary Decompilation LLM with Feedback-Driven Multi-Turn Refinement, 2026b. URL https://arxiv.org/abs/2606.16162.

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded Translation Validation for LLVM. In *Programming Language Design and Implementation (PLDI)*, 2021. doi: 10.1145/3453483.3454030. URL https://web.ist.utl.pt/nuno.lopes/pubs.php?id=alive2-pldi21.

Atharva Naik, Yash Mathur, Prakam, Carolyn Rose, and David Mortensen. Reacomp: Compiling llm reasoning into symbolic solvers for efficient program synthesis, 2026. URL https://arxiv.org/abs/2605.05485.

National Security Agency. Ghidra software reverse engineering framework. https://ghidra-sre.org/, 2019. Accessed: 2025-09-13.

Noelo Lab. Kuna, 2026. URL https://github.com/Noelo-Lab/kuna.

Nguyen Anh Quynh. Capstone: Next-gen disassembly framework. In *Black Hat USA*, 2014.

Nelson Higuera Ruiz, Markus Hofmarcher, and Claudiu Leoveanu-Condrei. Distilling answer set programming theories from large language models, 2026. URL https://arxiv.org/abs/2607.28086.

Eric Schulte, Jason Ruchti, M. Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. 2018.

Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, pp. 353–368. USENIX Association, 2013. ISBN 978-1-931971-03-4. URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz.

Zihan Sha, Hui Shu, Hao Wang, Zeyu Gao, Yang Lan, and Chao Zhang. Hyres: Recovering data structures in binaries via semantic enhanced hybrid reasoning. *ACM Transactions on Software Engineering and Methodology*, 35(3):1–28, 2026.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157, 2016.

Alexander Shypula, O. Bastani, and Edward J. Schwartz. Decaf: Improving neural decompilation with automatic feedback and search. *ArXiv*, abs/2605.11501, 2026.

Y. Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. pp. 245–251, 2010.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. LLM4Decompile: Decompiling Binary Code with Large Language Models, 2024. URL https://arxiv.org/abs/2403.05286.

Hanzhuo Tan, Weihao Li, Xiaolong Tian, Siyi Wang, Jiaming Liu, Jing Li, and Yuqun Zhang. Sk2decompile: Llm-based two-phase binary decompilation from skeleton to skin. *ArXiv*, abs/2509.22114, 2025.

Vector 35. Binary Ninja: A reverse engineering platform. https://binary.ninja/, 2024. Accessed: 2025-09-13.

Wai Kin Wong, Daoyuan Wu, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Decllm: Llm-augmented recompilable decompilation for enabling programmatic use of decompiled code. *Proceedings of the ACM on Software Engineering*, 2:1841 – 1864, 2025.

Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Network and Distributed System Security Symposium (NDSS)*, 2015. URL https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/no-more-gotos-decompilation-using-pattern-independent-control-flow-structuring-and-semantics/.

Khaled Yakdan, Sergej Dechand, E. Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 158–177, 2016.

Yapeng Ye, Zhuo Zhang, Qingkai Shi, Yousra Aafer, and X. Zhang. D-arm: Disassembling arm binaries by lightweight superset instruction interpretation and graph modeling. *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2391–2408, 2023.

Zhenghong Yu, Hangdong Zhao, Wanzhu Hou, and Paraschos Koutris. Flowlog: Re-thinking datalog for fast and extensible static analysis, 2026. URL https://arxiv.org/abs/2607.23971.

Yifan Zhang, Xiaohan Wang, Yueke Zhang, Yu Huang, and Kevin Leach. Constraint-Guided Multi-Agent Decompilation for Executable Binary Recovery, 2026. URL https://arxiv.org/abs/2604.23940.

Hangdong Zhao, Zhenghong Yu, Srinag Rao, Simon Frisk, Zhiwei Fan, and Paraschos Koutris. Flowlog: Efficient and extensible datalog via incrementality, 2025. URL https://arxiv.org/abs/2511.00865.

Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave (Jing) Tian. D-Helix: A Generic Decompiler Testing Framework Using Symbolic Differentiation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 397–414. USENIX Association, 2024. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/zou.

Muqi Zou, Hongyu Cai, Hongwei Wu, Zion Leonahenahe Basque, Arslan Khan, Berkay Celik, Dave (Jing) Tian, Antonio Bianchi, Ruoyu (Fish) Wang, and Dongyan Xu. D-LiFT: Improving LLM-based Decompiler Backend via Code Quality-driven Fine-tuning, 2025. URL https://arxiv.org/abs/2506.10125.

A TRAINING CORPORA AND MODEL USAGE

Table 3 gives the sizes of the three training pools. Each distinct function contributes four compiled variants, at O0, O1, O2, and O3, for a total of 998 training functions and 3,992 variants. These counts describe the available training inputs; a worker may revisit a function as the shared rule base changes. They are separate from the common-parseable evaluation subsets used for R2I in §4.1.

Table 3: Training corpus sizes from the all-optimization-level manifests.

Corpus	Training functions	O0–O3 variants
HumanEval	131	524
MBPP	739	2,956
ExeBench	128	512
Total	998	3,992

The manifests are the `corpus.json` files in the `humaneval`, `mbpp`, and `exebench` subdirectories of `learn_run/allopts/`. Workers receive assembly facts and feedback on emitted C. Reference C is used by the checking tools, and its LLVM bitcode is not part of the facts visible to workers (§3.2). During learning, if a message is flagged and the model response is cut off, we force a switch to the next training functions.

Table 4 reports input and output token usage and estimated API costs for the GPT-6 Astra, Opus, and DeepSeek learning pools. These costs concern learning; evaluating a frozen rule base makes no LLM calls.

Table 4: Token usage in millions (M) and estimated API cost in USD by dataset. Opus input counts are estimated as reported total tokens minus output tokens.

Dataset	Input tokens (M)	Output tokens (M)	Cost
GPT-6 Astra			
HumanEval	208.9	1.2	$394
MBPP	179.7	1.1	$352
ExeBench	1,079.7	3.5	$1,920
Opus			
HumanEval	671.4	3.6	$336
MBPP	590.1	2.9	$264
ExeBench	1,141.2	4.8	$469
DeepSeek			
HumanEval	1,562.3	30.9	$94.40
MBPP	2,399.2	37.9	$144.84
ExeBench	2,212.0	40.3	$143.93

Source artifact. The source release includes the Python implementation, YAML interfaces, and three exported ExeBench rule bases in `learned-rules/`. Its package specification requires Python 3.12 or newer and pins Capstone 5.0.8, Z3 4.16.0.0, and PyYAML 6.0.3. Compiler and oracle checks use LLVM 18 and a compatible KLEE installation. The reference Datalog evaluator is included; FlowLog and Rust compilation are external dependencies.

B SUBSTEP INTERFACES

Table 5 lists all 110 declared output relations of the fourteen substeps in execution order, from `rlcrafter/asm/steps/`. The YAML files also declare upstream inputs, vocabulary facts, and task guidelines; workers may add auxiliary relations and pure Rust helpers.

Table 5: Declared output relations by substep. Auxiliary relation names are learned.

Substep	Published output relations
1. `cfg`	`block`, `block_member`, `block_last`, `cfg_edge`, `jump_table`, `jump_table_target`, `jump_table_dispatch`, `function_exit`
2. `frame`	`stack_height`, `frame_pointer`, `frame_slot`, `frame_access`, `frame_address`, `frame_bookkeeping`, `saved_register`, `frame_canary`
3. `dataflow`	`location`, `value`, `value_part`, `use_site`, `reaches`, `copy`
4. `functions`	`abi_parameter`, `function_returns`, `function_result`, `call_target`, `indirect_call`, `recovered_argument`, `call_result`, `external_name`, `library_function`, `library_parameter`, `library_variadic`
5. `memory`	`access`, `access_address`, `access_object`, `address_term`, `memory_object`, `points_to`, `object_escapes`, `object_extent`, `constant_load`, `no_store_between`
6. `operations`	`operation`, `operand`, `constant`, `holds`, `store_value`, `absorbed`, `inert`
7. `conditions`	`condition`, `condition_producer`, `condition_value`, `condition_select`, `flag_value`, `switch_selector`, `switch_case`, `switch_default`, `switch_guard`
8. `variables`	`live_range`, `interferes`, `coalesceable`, `variable`, `variable_value`, `variable_use`
9. `types`	`c_type`, `unsigned_type`, `pointer_qualifier`, `struct_type`, `struct_declaration`, `struct_field`, `array_type`, `pointer_target`, `variable_type`, `signedness_evidence`, `pointee_evidence`, `function_signature`, `parameter`, `call_argument`, `return_value`, `external_signature`, `external_parameter`, `external_variadic`, `signature_ambiguity`
10. `statements`	`statement`, `statement_operand`, `statement_successor`, `statement_case`, `statement_callee`, `statement_effect`, `statement_owns`, `function_owns`
11. `expressions`	`value_use`, `foldable`, `binary_symbol`, `unsigned_binary_symbol`, `compare_symbol`, `unsigned_compare_symbol`, `plain_cast`, `wrapping_binary`
12. `ast`	`ast_node`, `ast_edge`, `ast_decision`, `ast_alternative`
13. `restructure`	`c_ast`, `c_ast_edge`
14. `maxsmt`	`maxsmt_problem`, `maxsmt_var`, `maxsmt_hard`, `maxsmt_soft`

Raw evidence routing. The routing file `rlcrafter/asm/asm-stages.yaml` gives substeps 1–7 selected instruction, operand, register, flag, memory, and ABI facts. `variables` receives `asm_instruction`; `statements` receives `asm_scope` and `asm_instruction`. AST construction receives only `asm_scope`, `asm_function`, `asm_external`, `asm_object`, `asm_c_label`, `asm_cstring`, and `asm_float` as raw inputs, in addition to its declared upstream relations. Types, expressions, restructuring, and MaxSMT receive no raw assembly relations. `asm_text` is not routed to any substep.

C RECORDED ASSEMBLY AND C OUTPUTS

The following outputs expand the repair in §4.2. The ExeBench function `ft_toupper`, compiled at O1, was presented to the Opus worker under the masked identifier `obj_5b3defe42474`. Its five decoded instructions are:

```
lea ecx, [rdi - 0x61]
lea eax, [rdi - 0x20]
cmp ecx, 0x1a
cmovae eax, edi
ret
```

The saved C before the repair, at rule base 1550, was:

```
int obj_5b3defe42474(int t0)
{
    return (((t0 - 97) >= 26) ? t0 : (t0 - 32));
}
```

The private trial and the subsequent check of the published rule base 1576 emitted the same corrected C:

```
int obj_5b3defe42474(int t0)
{
    return ((((unsigned int)(t0 - 97)) >= 26) ? t0 : (t0 - 32));
}
```

The change preserves the unsigned condition represented by `cmovae`. For input 86 (`'V'`), the first output returns 54 (`'6'`), while the repaired output returns 86. For KLEE's counterexample `t0` $= -4$, the two expressions return -36 and -4, respectively. The repair adds ten `types` clauses and fourteen `ast` clauses and revises two existing AST clauses, for 26 published edits. The new type rules derive unsigned width evidence; the AST rules insert the cast and reconnect operand edges.

Table 6: Recorded checks for the two C outputs. The private trial uses base 1550 plus the 26-edit batch; the published check uses base 1576.

Check	Before repair	Private trial	Published check
Compilation	No diagnostics	No diagnostics	No diagnostics
Corpus tests	Fails assertion 2	10 assertions pass	10 assertions pass
KLEE	Differs at -4	Agrees for $\|t0\| \leq 32$	Agrees for $\|t0\| \leq 32$

Both successful KLEE comparisons report one completed path. Their bound excludes alphabetic inputs, which are instead exercised by the corpus tests; the ten assertion evaluations include repeated inputs. The listings and checks document one training-function repair and do not establish full-domain equivalence or cross-function generalization.