CHISEL-ing Back Source Code with AI-enabled Iterative Recovery

Varun Kohli✉
A*STAR Institute of Advanced Intelligence and Computing
Singapore, Singapore
Kohli_Varun@a-star.edu.sg

N Raghava
Coimbatore Institute of Technology
Coimbatore, India
raghava.research.cs@gmail.com

Biplab Sikdar
National University of Singapore
Singapore, Singapore
bsikdar@nus.edu.sg

Dinil Mon Divakaran
A*STAR Institute of Advanced Intelligence and Computing
Singapore, Singapore
Dinil_Divakaran@a-star.edu.sg

Abstract

Decompilation aims to recover high-level, compilable, and semantically equivalent code from binaries. Traditional decompilers produce pseudo-C that is difficult to read and does not compile, while the recent LLM-assisted approaches generate readable, but semantically incorrect code. LLM-aided iterative recovery is an emerging branch of research, but prior works rely on supplied test suites for semantic recovery. In this work, we present Chisel, a test suite-free framework to iteratively recover source code from Ghidra-derived pseudo-C. Chisel uses simple yet effective feedback from a compiler (static analysis) and a coverage-guided fuzzer (differential analysis), augmented by rich observables for grounded divergence detection and feedback, cross-iteration divergence memory, and best candidate retention. We systematically evaluate Chisel for compilation and semantic recovery, feedback oracle soundness, and iteration overhead on 120 ExeBench functions compiled for the x86-64 architecture, across four optimizations (`O0`–`O3`), in both stripped and unstripped variants, using the open-weight Gemma4:31b LLM. Chisel, with all recommended features, achieves an average of 96.1% re-compilability and 79.8% re-executability rates at an average of 2.1 iterations. Significantly, Chisel recovers 26% of first-generation execution errors. At the same time, Chisel's feedback oracle falsely accepts only 9.4% candidates. Lastly, Chisel performs significantly better than two recent prior work on LLM-assisted decompilation.

CCS Concepts

• **Security and privacy → Software reverse engineering**.

ACM Reference Format:
Varun Kohli, N Raghava, Biplab Sikdar, and Dinil Mon Divakaran. 2026. CHISEL-ing Back Source Code with AI-enabled Iterative Recovery. In *2nd Workshop on Software Understanding and Reverse Engineering (SURE '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3847158.3847388

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Request permissions from owner/author(s).
SURE '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3019-1/2026/11
https://doi.org/10.1145/3847158.3847388

1 Introduction

Decompilation, i.e., the recovery of high-level language from compiled code, is valuable for reverse engineering, malware analysis, vulnerability detection and mitigation, analyzing patches and auditing legacy and firmware binaries for which no source is available [5, 17]. The recovery of readable, compilable, and semantically faithful source from machine code is a challenging task, and there has been significant effort in developing faithful decompilers that let an analyst read, recompile, patch, and re-deploy a binary for which no source is available. Traditional decompilers such as Ghidra [1] and IDA Pro [11] reconstruct control flow and data types using hand-built heuristics, which is difficult to read, rarely compilable, and does not guarantee semantic fidelity. The resultant pseudo C is typically obscured by synthetic types, machine-level artifacts and Application Binary Interface (ABI)-level scaffolding. This has led to the development of AI and LLM-based approaches to translate assembly or decompiler output into readable and compilable source code [6, 10, 12, 16, 18], which can be broadly classified into two categories: *Non-iterative decompilers* use finetuned LLMs to recover source code, relying solely on the LLM's intrinsic capability to generate top-1 or top-k candidates without providing feedback for improvement after failed generations [4, 8, 13, 18]. *Iterative decompilers* address these limitations' by using feedback derived from in-loop oracles to guide the LLM towards better generation [7, 12, 19, 20].

However, several methods rely on native test suites to identify semantic divergence that is then passed to the LLM as feedback [7, 19]. This cannot be assumed when reversing binaries derived from unknown source code, and is the biggest limitation of prior work. These works also use a sanitizer to fix memory corruption errors which is not faithful when done asymmetrically only to the candidate, since the original source may also showcase such properties. Some works use symbolic execution as part of a finetuning pipeline or for multi-role generation, but consequently suffer from path explosion, making it challenging to generalize across control-flow types [12, 21, 22]. Further, in-loop feedback coverage is typically limited to compiler errors and return value divergence [19, 20].

To address the above limitations, we develop and present Chisel[1], a test suite-free iterative enhancement decompiler. We make the following contributions:

(1) A lightweight and test-suite-free iterative decompilation framework whose in-loop signals are derived only from the original binary using a compiler and coverage-guided fuzzer,

[1] Controlled Heuristics for Iterative Semantic Extraction and Lifting

Table 1: A comparison of LLM-assisted decompilation methods related to Chisel.

Method	Iterative	test suite-free	In-loop Oracle			Soundness Evaluation	Signature Type Extraction	Rich Observables	Divergence Memory	Best Candidate Retention
			Compiler	Sanitizer	Fuzzer					
LLM4Decompile [18]	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
Nova [13]	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
Idioms [8]	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
DeGPT [12]	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
DecLLM [19]	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗
PCodeTrans [7]	✓	✗	✓	✓	✗	✗	✗	✓	✗	✗
Agent4Decompile [20]	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
Chisel (ours)	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓

aided by rich observables for grounded differential analysis, cross-iteration divergence memory, and best-candidate retention. Our code is available on GitHub[2].

(2) A systematic six-arm ablation of Chisel's iteration harness, building up from single-shot LLM generation to our end-to-end design, highlighting the incremental impact of each components across three axes of evaluation, namely *recovery* of source, which covers Re-compilability (RC), Re-executability (RE), regression, and three new recovery metrics that we define in Section 4 to evaluate the effectiveness of an iterative decompiler; *soundness* of the feedback oracle, which covers the oracle's False-acceptance (FA) and False-rejection (FR) rates judged against the ground-truth test suite; and *overhead*, which includes the average number of iterations.

(3) A systematic evaluation on 120 ExeBench functions [3] across four optimization levels (`O0`–`O3`), with both stripped and unstripped x86-64 binaries, using an off-the-shelf, open-weight Gemma4:31b model in all experiments, that reports 96.1% RC, 79.8% RE, 9.4% FA, and 26.2% recovery after first-generation failure. We also report better performance than LLM4Decompile [18] and Agent4Decompile [20].

2 Related Work

Table 1 compares related works over methodologies and design. The first family of LLM-assisted decompilers produces candidate source code without behavioral feedback to improve generation. LLM4Decompile [18] and Nova [13] fine-tune LLMs to translate assembly or Ghidra pseudo-C into compilable and re-executable C, and Idioms [8] jointly recovers code and its user-defined types from decompiler output. These methods do not utilize feedback derived from the generated code or differential analysis to improve the quality of generation, relying only on the LLMs inherent capability to recover good-quality source code. DeGPT [12] improves readability with a multi-role pipeline and a static symbolic check that refuses edits that change the function.

Iterative decompilation methods overcome the limitations of non-iterative approaches through feedback loops. DecLLM [19] repairs compiler errors, identifies semantic divergence using coverage-guided AFL++ [9] seeded by a supplied ground-truth test suite, and employs a sanitizer on the generated candidate to fix memory misuse. PCodeTrans [7] validates hot-swapped function using the program's official test suite, then localizes failures with breakpoint-matched differential tracing that compares control flow and variable values against the original. Both works use compilers and sanitizers, while DecLLM also uses a fuzzer for divergence feedback. However, the use of a sanitizer to asymetrically fix the generated code (while not the origin) is unfaithful decompilation. Agent4Decompile [20] proposes a three-tier hierarchical agentic approach covering syntax, compilation, and execution checks that assume the availability of supplied test cases, showcasing the first agentic approach to pseudocode enhancement.

The main limitation of the above methods is the reliance on a test suite for in-loop feedback, which may not be possible to access when reversing binaries whose source is unknown. Additionally, with the exception of PCodeTrans [7], prior works typically have limited coverage of observable behaviors for differential analysis, and show limited capability. In contrast, Chisel is an iterative enhancement framework that improves the output of traditional decompilers without supplied test suites. It uses in-loop signals derived solely from the original binary, performing static analysis using a compiler for compilation failures and differential analysis through type-aware coverage-guided fuzzing over a rich observable space that includes return values, output buffer contents, stdout, exit codes, timeouts, and crash signals. Chisel also integrates cross-iteration divergence memory and best candidate retention if the feedback oracle doesn't accept any candidate after the last iteration.

3 Proposed Framework: Chisel

This section presents the design components of Chisel, including the iterative enhancement algorithm, constituent oracles, features, prompts and parameter selection.

3.1 Iterative Enhancement Algorithm

Chisel's novelty lies in its test-suite-free iterative refinement through compiler and type-aware differential fuzzer feedback, augmented by features for broader observability, divergence memory, and best candidate retention. Fig. 1 provides an overview of Chisel and its iterative procedure is depicted by Algorithm 1. The algorithm takes Ghidra pseudo-C (P) of a single function, the original reference object (o) that needs to be decompiled, iteration budget (K), feedback oracles (O), feedback budgets (d_c, d_f), and features ($\mathcal{F}$). The algorithm returns one compilable C function `func0`. The subsequent text explains the algorithm assuming all components are active.

[2] https://github.com/VarunKohli18/chisel

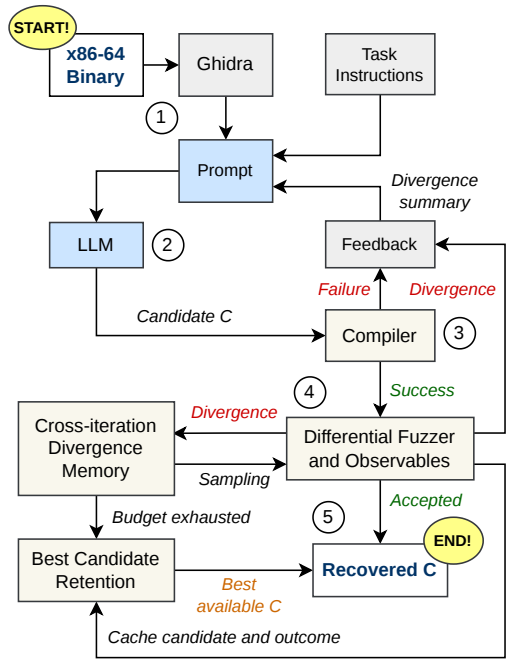

Figure 1: Overview of CHISEL. The overall sequence of steps is marked with numbers.

Each round samples a single candidate c from the LLM at zero temperature, using a self-contained prompt assembled from P, the previous candidate (*prev*), and the current feedback (f_b) derived from oracles (line 2-22): O·COMPILER is a static feedback oracle that compiles the generated C, parses observed failures, and shares them as $f_b \leftarrow$ DIAGNOSTICS(c) (lines 8-9), nudging the LLM towards generating compilable C in the next iteration. If c compiles, O.DIFFERENTIAL searches for inputs that expose divergence of c relative to o through coverage-guided entropic libFuzzer mining of B samples from a type-aware seed corpus (TYPESEEDS) derived from the parsed signature (σ) for a better starting point than random (lines 11-16). Discriminating inputs are sampled from the accumulated set (E) and passed as feedback $f_b \leftarrow$ SAMPLE(E) (lines 17-21). Only a maximum of d_c compilation errors and d_f fuzzer divergences are passed as feedback and the remaining are appended as a count instead to stay within token budgets. If neither oracle flags an issue, c is returned as the recovered source (lines 23-24).

Beyond the oracles, CHISEL is equipped with three key *features* to improve recovery of failed generations and oracle soundness. (1) $\mathcal{F}$·OBSERVE (lines 13-17) makes the differential analysis type-aware and extends observation to return values, output buffer contents, stdout, timeouts, exit codes, and crash signals, and compares pointer results by content rather than raw address, providing a richer differential feedback to the LLM for improvement. (2) $\mathcal{F}$.MEMORY (lines 18-19) grows M with the deduplicated divergence-inducing inputs across iterations and replays them under the same sampling budget to ensure that the generation maintains performance on prior divergences and converges. Finally, to handle no accepted generation or the regression of code quality

Algorithm 1: CHISEL's iterative enhancement loop.

Input: pseudo-C P, reference object o, round budget K, feedback budgets d_c, d_f, oracles O, features $\mathcal{F}$
Output: a C function `func0`

1 $prev, f_b, M, C \leftarrow \varnothing$;
2 **for** $k \leftarrow 1$ **to** K **do**
3 $c \leftarrow \text{LLM}(P, prev, f_b)$;
4 **if** $O = \varnothing$ **then**
5 **return** c ; `// Single-shot generation`
6 $prev \leftarrow c$; $f_b \leftarrow \varnothing$;
7 $accept \leftarrow$ `true`;
8 **if** O.COMPILER **and** $\neg$ COMPILE(c) **then**
9 $f_b \leftarrow$ DIAGNOSTICS(c);
10 **continue**;
11 **if** O.DIFFERENTIAL **then**
12 $\sigma \leftarrow$ PARSESIGNATURE(c);
13 **if** $\mathcal{F}$.OBSERVE **then**
14 $X \leftarrow$ LIBFUZZER($c, o, \sigma, B,$ TYPESEEDS(σ));
15 **else**
16 $X \leftarrow$ LIBFUZZER($c, o, \sigma, B,$ RANDOMSEEDS(σ));
17 $E \leftarrow$ DIFF($c, o, X \cup M$);
18 **if** $\mathcal{F}$.MEMORY **then**
19 $M \leftarrow M \cup E$; `// Memory accumulation`
20 **if** $E \neq \varnothing$ **then**
21 $f_b \leftarrow$ SAMPLE(E);
22 $accept \leftarrow$ `false` ;
23 **if** $accept$ **then**
24 **return** c ; `// Accepted candidate`
25 $C \leftarrow C \cup \{c\}$;
26 **if** $\mathcal{F}$.RETAIN_BEST **then**
27 **return** arg min DIVERGENCE(C) ; `// Best candidate`
28 **return** c

through iterations, (3) $\mathcal{F}$.RETAIN_BEST (lines 23-24) returns the best-performing candidate from the loop as the recovered source code. Overall, CHISEL provides a novel iterative harness that nudges generation towards compilable and functionally similar code.

3.2 Prompts and Parameters

Every round is driven by one self-contained prompt which is depicted in Table 2. It consists of a fixed task instruction that guides the LLM to rewrite the pseudo C as compilable C named `func0` that reproduces the original's behavior exactly, preserving the control flow and the order of operations while dropping decompiler scaffolding that reflects the machine and the ABI rather than the program. The first round shows only this instruction and the pseudo C. Later rounds append *prev* and f_b as per availability.

We set $K = 5$, the LLM is set at temperature 0 for reproducibility, and reasoning is turned off. Candidates are compiled at `O0` optimization by default for static and differential analysis. Each round tests c against o on a 200-sample corpus of type-aware seeds and

Table 2: Prompt components and descriptive samples.

Component	Content
Task instruction	"Below is Ghidra pseudo C for one function. Rewrite it as compilable C named func0 that reproduces the original's behavior exactly, the same return value and the same writes through its pointer arguments. Keep the computation and control flow as written: preserve the order of operations and do not restructure or simplify loops, reorder side effects, or add or remove returns. But omit the decompiler scaffolding that reflects the machine and the ABI rather than the program, such as stack-protector canary checks, the placeholders the decompiler invents for values it could not recover, and register or stack spill temporaries, and rewrite synthetic types and raw memory addressing as ordinary C. The guiding rule: keep every statement that affects the return value or bytes written through the pointer arguments, and drop every statement that does not. Output only the C source of func0, no prose and no comments, and include every header and type it needs."
Ghidra pseudo-C	"Ghidra pseudo C: {PSEUDO_C}"
Previous generation	E.g."Previous attempt: {PREVIOUS}"
Compiler feedback	E.g. 1."Compilation failed: {ERRORS}. Fix it." E.g. 2."Compilation timed out. Simplify the control flow and try again."
Fuzzer feedback	E.g. 1."On func0(...): outputs agree on the first {COUNT} value(s), then differ at index {INDEX}: original {ORIGINAL_VALUES}." E.g. 2."On func0(...): original printed {OUTPUT}, candidate crashed with signal {S}." E.g. 3."On func0(...): original printed {OUTPUT}, candidate timed out." E.g. 4."On func0(...): original printed {OUTPUT}, candidate printed {OUTPUT'}."

$B = 2000$ libFuzzer-mined inputs, with a per-input timeout of 2 CPU seconds to catch runaway loops. We set $d_c = 5$ and $d_f = 10$.

4 Experimental Setup

This sections provides details of the experimental setup including hardware, software, dataset, experiments, evaluation metrics, and baselines compared.

4.1 Hardware, Software, and Dataset

We run all experiments on a x86-64 Linux host with 4 NVIDIA H200 GPUs. We use the Ollama-provisioned `Gemma4:31b`[3] at int-4 quantization for our experiments. The input input and output context limits set to 24,576 and 8,192 tokens respectively. The harness uses Ghidra for lifting assembly to pseudo-C, `gcc -c` for compiler feedback, and libFuzzer for divergence feedback. We draw the 150 longest real-world functions from ExeBench's real and val-real splits [3], covering applications including and not limited to, digital signal processing, cryptography, geometry, linear algebra, and data structure. The functions are given a ground truth test suite of 1000 input-output pairs using a generator and further filtered to 120 functions based on the availability of at least 200 functional test cases. The selected functions have 46-265 lines-of-code, and are compiled on x86-64 at `O0-O3` optimizations in stripped and unstripped variants. The cached dataset is available on our GitHub.

4.2 Experiments and Evaluation Metrics

We evaluate Chisel through a systematic, six-arm incremental ablation of its harness starting from a one-shot LLM, and adding oracles (compiler, fuzzer) and features (observables, cross-iteration memory, best candidate retention), where each arm retains the prior features and the final arm corresponds to the complete Chisel framework. The ablation enables us to evaluate the value of each component along three key axes, i.e., *recovery* which measures how much correct source C the loop produces, *soundness* measures whether the feedback oracle's accept and reject decisions agree with the test suite, and *overhead* which measures the cost of the added complexity. All percentage metrics are computed on the single candidate the loop finally returns, scored against the held-out suite, while overhead is measured in average number of iterations ($\bar{k}$). A candidate *compiles* if it builds, and *re-executes* if it matches the original on every suite input. The metrics, including our newly defined $R_{comp}, R_{exec}, R_{tot}$, are as follows:

- *RC* and *RE*: the fractions of returned candidates that compile and that re-execute (pass all test cases), respectively.
- *Pass*: the average fraction of test cases passed across samples.
- R_{comp}: of the samples whose first generation did not compile, the fraction whose returned candidate compiles.
- R_{exec}: of the samples whose first generation compiled but did not re-execute, the fraction whose returned candidate re-executes.
- R_{tot}: of all samples whose first generation did not re-execute, the fraction whose returned candidate re-executes, i.e. the loop's overall recovery from first-generation failure.
- *FA*: of the candidates the oracle accepted, the fraction that stopped on an incorrect candidate.
- *FR*: of the candidates the oracle did not accept, the fraction that were labeled as correct by the evaluation test suite.
- *Reg*: of all samples whose first generation already re-executed, the fraction whose returned candidates did not re-execute (regressed).
- $\bar{k}$: the mean number of iterations the decompiler runs across all samples in the dataset.

Beyond *RC*, *RE*, and *Pass*, a good iterative decompiler is also defined by its $R_{comp}, R_{exec}, R_{tot}$, and *Reg*. The feedback oracle should (ideally) only accept good-quality candidates and reject incorrect ones. Thus, the design of an oracle should report low *FA* and *FR*. Over-rejection (high *FR*) leads to longer iterative loops and higher $\bar{k}$. However, unless the iteration budget and overhead constraints are tight, this means the oracle is strict, making it a less serious concern since the LLM is then triggered to generate "better" code. However,

[3]https://ollama.com/library/gemma4:31b

Table 3: Metrics derived from one run of a six-arm ablation, averaged over all variants.

Harness Arm	*RC*	*RE*	*Pass*	R_{comp}	R_{exec}	R_{tot}	*FA*	*FR*	*Reg*	$\#\bar{k}$	Remarks
LLM (one-shot)	90.0	73.0	75.1	–	–	–	–	–	–	–	LLM-only, no feedback
+ `compiler`	96.6	77.1	79.2	65.6	0.0	15.1	20.2	0.0	0.0	1.18	gcc oracle
+ `fuzzer`	96.6	79.3	81.7	67.3	7.5	23.5	17.6	45.0	0.0	1.43	Differential libFuzzer oracle
+ `observe`	96.7	79.9	82.0	66.7	13.0	26.0	9.4	45.4	0.3	2.05	Type aware and rich observables
+ `memory`	96.1	79.6	81.8	64.1	14.4	25.9	9.4	45.1	0.1	2.06	Divergence memory and replay
+ `best` (full CHISEL)	96.1	79.8	82.1	64.1	15.0	26.2	9.4	45.9	0.0	2.06	Best-candidate retention

high *FA* implies the system produces low-quality candidates, which is undesirable for any decompiler.

4.3 Baselines

We evaluate two prior works on our dataset and evaluation harness: i) LLM4Decompile-9B-v2 from [18] and available on HuggingFace [15], and ii) Agent4Decompile [20] (code available at [2]). We replace the LLM in Agent4Decompile with Gemma4:31 for a fair comparison of the harness. Further, to align with our test-suite-free setting , no method has access to the ground truth test suite during iterations. Both methods use Ghidra for binary to pseudo-C lifting, and are run using their specified prompts and input formats. We do not compare with other iterative methods such as DecLLM [19] and PCodeTrans [7] since their code is not public.

5 Results

We evaluate CHISEL's performance using metrics that capture recovery, soundness, and overhead, shown first as a six-arm ablation of its harness, then performance across optimizations for unstripped and stripped cases, and finally compare against recent work.

As Table 3 and Fig 2 show, the raw one-shot model compiles 90.0% and re-executes 73.0% of functions. Adding the compiler oracle lifts re-compilation to 96.6% with $R_{comp} = 65.6\%$, but is unable to repair semantic behavior after first generation ($R_{exec} = 0$), since it accepts the first candidate that builds without discriminating over behavioral correctness. The differential fuzzer adds the required behavioral signal to improve *RE* and R_{exec}, recovers 7.5% of compiled-but-incorrect first generations, raises *RE* to 79.3%, and lowers *FA* from 27% for the LLM arm and 20.2% for the compiler arm, to 17.6%. The decisive soundness gain comes at the observe arm, which *halves FA* from 17.6% to 9.4% at equal *RE* and *FR*, highlighting the value of rich observation that in turn leads to rich feedback and recovery. The memory and best-candidate retention arms show an improvement in R_{exec} and reduction in regression. Samples derived from best-candidate retention report mean suite pass-rate increase from 50.9% to 52% compared to the previous arm. Fig. 3 (left) further highlights the importance of cross-iteration divergence memory. *Pass* is higher for the `+memory` arm compared to `+observe`. Fig. 3 (right) shows the distribution of feedback oracle acceptance per iteration.

Based on our inspection of the dataset, Ghidra pseudo C, and the LLM's generation through CHISEL, the residual 9.4% *FA* is dominated by signature recovery errors predominantly in the stripped samples. The differential oracle tests each candidate against its *own*

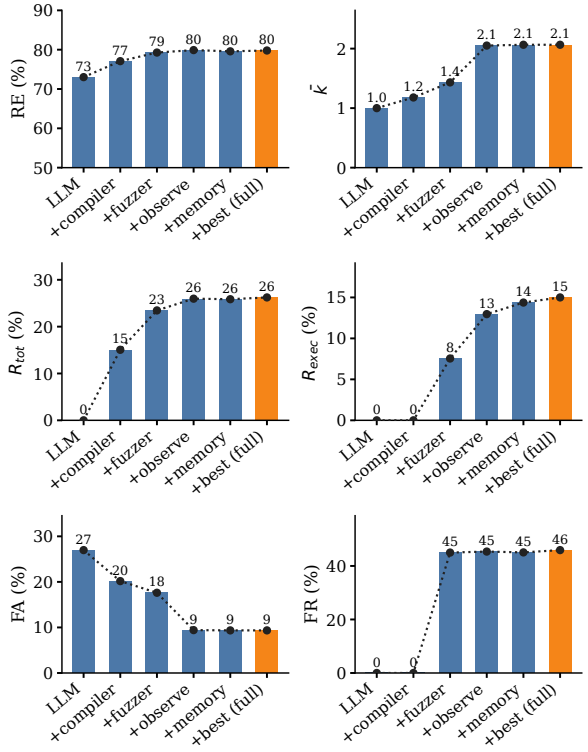

Figure 2: Key metric trends across the six-arm ablation highlighting their value in driving recovery and oracle soundness.

recovered signature, so a candidate that mis-recovers an argument is self-consistent under the oracle yet fails the ground-truth suite that is built on the original signatures. This is visible in the strip split shown in Tables 4 and 5, where *FA* = 16.4% on stripped binaries but only 1.7% on unstripped, as stripping removes the type information needed by the decompiler (Ghidra) to recover the signature, which subsequently passes the error to the signature extractor used in our work (see PARSESIGNATURE at line 12 of Algorithm 1). *FR* rises to nearly 45% once a behavioral oracle is integrated into the system, suggesting that the oracle is stricter than the LLM-only and compiler arms. *FR* falls monotonically with optimization regardless of stripped or unstripped symbols. We also observe that our reported *FR* is inflated partly by the limitation of the test suite. The differential oracle outpaces the test suite due to a wider input

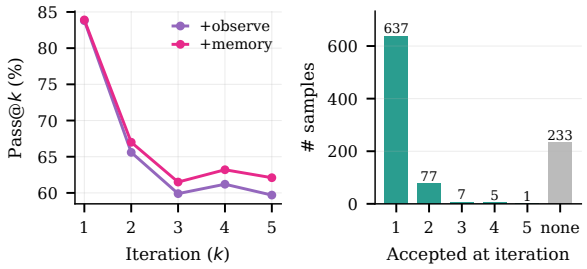

Figure 3: Per-iteration evaluation.

Table 4: Chisel's recovery and soundness on 480 unstripped x86-64 binaries (120 functions), by optimization level.

Optimization	*RC* (%)	*RE* (%)	R_{exec} (%)	R_{tot} (%)	*FA* (%)	*FR* (%)
O0	99.2	93.3	14.3	33.3	0.0	75.8
O1	97.5	90.0	10.0	42.9	2.1	58.3
O2	96.7	82.5	28.6	38.2	2.4	45.7
O3	95.8	80.8	11.1	32.4	2.4	44.7
Average	**97.3**	**86.7**	**17.9**	**36.6**	**1.7**	**55.4**

Table 5: Chisel's recovery and soundness on 480 stripped x86-64 binaries (120 functions), by optimization level.

Optimization	*RC* (%)	*RE* (%)	R_{exec} (%)	R_{tot} (%)	*FA* (%)	*FR* (%)
O0	97.5	78.3	8.3	18.8	16.3	54.5
O1	96.7	75.8	13.0	19.4	17.0	40.0
O2	95.0	69.2	13.3	21.3	17.6	27.6
O3	90.8	68.3	18.5	19.1	14.8	21.9
Average	**95.0**	**72.9**	**13.5**	**19.8**	**16.4**	**34.0**

Table 6: Comparison with prior work.

Method	*RC* (%)	*RE* (%)	R_{tot} (%)	*FA* (%)	Iterations
Ghidra pseudo-C	35.0	28.1	–	–	–
LLM4Decompile-9B-v2 [18]	63.2	17.0	–	–	–
`Gemma4:31b`	90.0	73.0	–	–	–
+ compiler	96.6	77.1	15.1	20.2	1.18
+ fuzzer	96.6	79.3	23.5	17.6	1.43
A4D-`Gemma4:31b` [20]	87.3	77.7	15.7	11.0	2.68
Chisel (full)	**96.1**	**79.8**	**26.2**	**9.4**	2.06

domain (2000 samples per-iteration not considering cross-iteration memory samples). Its coverage includes divergences that were not included in the test suite.

Chisel re-executes 86.7% of unstripped and 72.9% of stripped functions on average. Within each variant re-execution falls as optimization rises, as seen in both unstripped and stripped cases where *RE* falls from 93.3% at O0 to 80.8% at O3, and 78.3% at O0 to 68.3% at O3, respectively. Meanwhile recovery rates R_{exec} and R_{tot} confirm that a growing share of that gap is repaired by the loop.

Comparison with baselines. Table 6 presents the results of Chisel and two SOTA decompilers on the same 120 functions and test suite. The specialized one-shot decompiler, LLM4Decompile [19] (LLM4Decompile-9B-v2), achieves only 17% RE, which is below our LLM-only one-shot Gemma arm that achieves 73.0% and highlights the significance of LLM capabilities for decompilation. We also evaluate Agent4Dedcompile (A4D) [20] in a test-suite free setting. Chisel achieves performs better at a lower iteration cost.

6 Limitations, Risks and Ethics

Signature recovery from stripped binaries continues to be a limitation for decompilers. In addition, our experiments were conducted at 0 temperature for determinism and non-reasoning settings reduced overhead. Higher temperatures and LLM reasoning may help improve recovery through more creative and better generation, critical thinking, and response to feedback. Further, we use Ghidra pseudo-C of the original binary as the source of truth and further evaluation into decompiler resilience is required [23]. Additionally, I/O sampling, both as an in-loop oracle and evaluation instrument, is an approximation of functional similarity and does not indicate semantic equivalence. Lastly, while ExeBench is popularly used in decompilation research, future works may consider using newly released datasets, e.g., [14], to avoid the risk of testing LLMs on their training data.

On the ethical front, this research was conducted to push the frontier of reverse engineering. However, our work uses open-weight models, and our ideas could potentially be adopted by threat actors for unauthorized activities. We urge ethical use of this work.

7 Conclusion

We presented Chisel, a test suite-free iterative decompiler that integrates static and differential analysis using compilers and coverage-guided fuzzers for compilation and execution recovery, augmented by rich observables, cross-iteration divergence memory, and best-candidate recovery. On a 120 function subset of ExeBench functions compiled to four optimizations on x86-64 with stripped and unstripped variants, Chisel showed 96.1% RC and 79.8% RE, and repairs 26.2% of first-generation failures, while halving the in-loop oracle's FA from 17.6% to 9.4% relative to a return-only differential fuzzer. The decisive gain comes from richer observation rather than from more search. On the same subset Chisel far exceeds the specialized one-shot LLM4Decompile-9B-v2 model (79.8% vs 17.0% RE) and outperforms Agent4Decompile in terms of RE, FA, and overhead using the same LLM and test-suite assumptions. Our results highlight that, in addition to the LLM, the feedback harness is an important lever to improve decompilation performance.

Acknowledgments

This research/project is supported by the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore under the National Cybersecurity R&D Programme and the CyberSG R&D Programme Office (Award CRPO-GC2-ASTAR-001). Any opinions, findings, conclusions, or recommendations expressed in these materials are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, the Cyber Security Agency of Singapore, or the CyberSG R&D Programme Office.

References

[1] National Security Agency. 2026. *ghidra*. https://github.com/NationalSecurityAgency/ghidra

[2] Anonymous. 2026. *Agent4Decompile Artifacts*. https://anonymous.4open.science/r/agent4decompile-artifacts-0F69/

[3] Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhaes, and Michael FP O'Boyle. 2022. ExeBench: an ML-scale dataset of executable C functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 50–59.

[4] Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael FP O'Boyle. 2024. Slade: A portable small language model decompiler for optimized assembly. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 67–80.

[5] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. 2024. 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)*. 361–378.

[6] Zion Leonahenahe Basque, Samuele Doria, Ananta Soneji, Wil Gibbs, Adam Doupé, Yan Shoshitaishvili, Eleonora Losiouk, Ruoyu Wang, Simone Aonzo, et al. 2026. Decompiling the Synergy: An Empirical Study of Human-LLM Teaming in Software Reverse Engineering. In *Proc. NDSS*.

[7] Yuxin Cui, Zeyu Gao, Shuxian He, Siliang Qin, and Chao Zhang. 2026. PCode-Trans: Translate Decompiled Pseudocode to Compilable and Executable Equivalent. *arXiv preprint arXiv:2603.14855* (2026).

[8] Luke Dramko, Claire Le Goues, and Edward J Schwartz. 2026. Idioms: A Simple and Effective Framework for Turbo-Charging Local Neural Decompilation with Well-Defined Types.. In *NDSS*.

[9] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[10] Zeyu Gao, Yuxin Cui, Hao Wang, Siliang Qin, Yuanda Wang, Zhang Bolun, and Chao Zhang. 2025. DecompileBench: A comprehensive benchmark for evaluating decompilers in real-world scenarios. In *Findings of the Association for Computational Linguistics: ACL 2025*. 23250–23267.

[11] Hex-Rays. 2026. *IDA Pro: Power Disassembler, Decompiler & Debugger*. https://hex-rays.com/ida-pro

[12] Peiwei Hu, Ruigang Liang, and Kai Chen. 2024. DeGPT: Optimizing Decompiler Output with LLM.. In *NDSS*.

[13] Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, Xiangyu Zhang, and Petr Babkin. 2025. Nova: Generative language models for assembly code with hierarchical attention and contrastive learning. In *International Conference on Learning Representations*, Vol. 2025. 95905–95926.

[14] Chang Liu, Noah Fleischmann, Nicolò Altamura, Edward Raff, James Holt, and Kristopher Micinski. 2026. ASSEMBLAGE-DEEPHISTORY: A Cross-Build Binary Dataset with Temporal Coverage. arXiv:2605.21615 [cs.CR] https://arxiv.org/abs/2605.21615

[15] LLM4Decompile. [n. d.]. *LLM4Decompile-9B-v2*. https://huggingface.co/LLM4Binary/llm4decompile-9b-v2

[16] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, Adam Doupé, Chitta Baral, and Ruoyu Wang. 2024. "Len or index or count, anything but v1": Predicting Variable Names in Decompilation Output with Transfer Learning. In *2024 IEEE Symposium on Security and Privacy (SP)*. 4069–4087.

[17] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*.

[18] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. Llm4decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 3473–3487.

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

[20] Yifan Zhang, Xiaohan Wang, Yueke Zhang, Yu Huang, and Kevin Leach. 2026. Constraint-Guided Multi-Agent Decompilation for Executable Binary Recovery. *arXiv preprint arXiv:2604.23940* (2026).

[21] Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu, Min Yang, Zhe Wang, and Chenggang Wu. 2022. Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths. In *31st USENIX Security Symposium (USENIX Security 22)*. 4365–4382.

[22] Muqi Zou, Hongyu Cai, Hongwei Wu, Zion Leonahenahe Basque, Arslan Khan, Berkay Celik, Antonio Bianchi, Dongyan Xu, et al. 2025. D-lift: Improving llm-based decompiler backend via code quality-driven fine-tuning. *arXiv preprint arXiv:2506.10125* (2025).

[23] Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave (Jing) Tian. 2024. D-Helix: A Generic Decompiler Testing Framework Using Symbolic Differentiation. In *33rd USENIX Security Symposium (USENIX Security 24)*. 397–414.

A Generative AI Usage

We used generative AI tools, include Claude and Gemini, to (i) improve the language of the manuscript, (ii) generate the plotting code (verified) that renders figures from our experimental outputs, and (iii) writing the code for our prototype. All experiments, datasets, and numerical results were produced by our own pipeline, every figure and table was regenerated directly from the raw experimental outputs and checked against the underlying data. The authors reviewed all AI-assisted code, and take full responsibility for the accuracy, originality, and integrity of this work.