Enhancing Evolving Exact Decompilation with LLM-Mixed Seeding

Ryota Saito
Shizuoka University
Hamamatsu, Shizuoka, Japan
saito@sec.inf.shizuoka.ac.jp

Masakatsu Nishigaki
Shizuoka University
Hamamatsu, Shizuoka, Japan
nisigaki@inf.shizuoka.ac.jp

Tetsushi Ohki
Shizuoka University and RIKEN AIP
Hamamatsu, Shizuoka, Japan
ohki@inf.shizuoka.ac.jp

Abstract

Byte-equivalent decompilation seeks source code whose recompilation reproduces the instruction sequence of a target binary. Evolving Exact Decompilation (EED)-style methods cast this task as evolutionary search, but their cost depends on the quality of the initial population. We study whether large language models can improve EED-style search when used as initializers rather than final decompilers. We implement an independent EED-style system and introduce LLM-mixed seeding, which replaces half of the initial population with LLM-generated C candidates. On small C programs reconstructed from the original EED tasks, LLM-mixed seeding improves final normalized instruction-token similarity by 17.90 percentage points over a database-seeded baseline. After controlling for generation-0 similarity and experimental factors, LLM-mixed trials also close more of the remaining gap to 100% similarity. These results suggest that LLM-generated code can provide useful starting points for recompilation-guided evolutionary search.

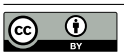
CCS Concepts

• **Security and privacy** → **Software reverse engineering**; • **Software and its engineering** → *Search-based software engineering*; • **Computing methodologies** → *Machine learning*.

ACM Reference Format:
Ryota Saito, Masakatsu Nishigaki, and Tetsushi Ohki. 2026. Enhancing Evolving Exact Decompilation with LLM-Mixed Seeding. In *2nd Workshop on Software Understanding and Reverse Engineering (SURE '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3847158.3847386

1 Introduction

Decompilation is a fundamental technique for translating binary executables into high-level representations. It is widely used in cybersecurity contexts, including malware analysis, vulnerability research, and binary patching. Most decompilers aim to produce readable output rather than recompilable source code; however, this design leaves two practical limitations relevant to reverse engineering. First, their outputs are typically pseudocode and therefore cannot be recompiled. Second, the recovered programs often contain inaccuracies in control-flow logic, data types, and data structures. These limitations make both manual and automated reverse engineering more difficult.



This work is licensed under a Creative Commons Attribution 4.0 International License.
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/10.1145/3847158.3847386

Byte-Equivalent Decompilation (BED) instead targets equivalent recoverability. Following Evolving Exact Decompilation (EED), we use byte-equivalent to mean that recompiling the generated source code yields a machine-instruction sequence equivalent to that of the input binary. EED[14] is an implementation of this idea. It evolves candidate source programs using a large database of source-code and assembly-instruction pairs together with a genetic algorithm, and terminates when bytecode equivalence with the input binary is achieved. Byte-equivalent decompilation could support automated reverse engineering, high-level fuzzing for vulnerability discovery from binaries, binary patching and re-hosting, and the application of formal verification and program analysis techniques to binary-only software.

To the best of our knowledge, there has been little subsequent research or independent validation related to BED and EED. One possible reason is that the original implementation of EED has not been made publicly available, and reproducing the system requires substantial engineering effort. A key bottleneck of EED is computational cost. Prior work reports that even small functions may require tens of thousands of iterations before a byte-equivalent match is found[14]. Since EED initializes its genetic search with candidates retrieved from a database, the quality of the initial population affects both the number of iterations required for exact recovery and the quality reachable under a finite search budget. Prior work reports that database-retrieved seeds reach only up to 55% similarity to the target instruction sequence[14], suggesting that limited seed quality is one source of EED's search cost.

Recent studies have explored decompilation methods based on large language models (LLMs) and machine learning (ML), including LLM4Decompile[16]. Without explicit equivalence checking or search, however, these methods do not guarantee bytecode-exact outputs. This motivates the hypothesis: LLM-generated code may provide initial candidates closer to the target under the fitness metric for EED, while EED's equivalence-guided search may repair those candidates toward bytecode equivalence.

In this paper, we build an independent EED-style implementation and use it to test this hypothesis under a fixed search budget. As a first step toward evaluating LLM-assisted EED-style search, we restrict our study to small single-function C programs reconstructed from the original EED tasks, with fixed compiler configurations and evaluation based on normalized instruction-token similarity. This controlled setting does not attempt to solve full-scale byte-equivalent decompilation, but is designed to evaluate whether LLM-generated initial candidates improve recompilation-guided evolutionary search under comparable conditions. We compare a database-seeded baseline with an LLM-mixed configuration in which half of the initial population is replaced by LLM-generated

candidates. Both configurations use the same evolutionary operators, compiler settings, normalization procedure, and evaluation budget. This design allows us to ask two separate questions: whether LLM-mixed seeding improves the final similarity reached by the search, and whether the search still makes greater residual progress after accounting for the quality of the initial population.

Under this controlled setting, our experiments show that LLM-mixed seeding improves final normalized instruction-token similarity under the fixed budget. They also show that the gain is not explained only by better generation-0 candidates: after controlling for initial similarity and experimental factors, LLM-mixed trials close more of the remaining gap to 100% similarity. These results suggest that LLM-generated code is useful not only as a decompiler output, but also as a source of structured starting points for recompilation-guided search. This paper makes the following contributions:

- We build an independent EED-style implementation for controlled experiments on EED-style byte-equivalent decompilation, without claiming numerical replication of the original EED results.
- We introduce LLM-mixed seeding, which initializes part of the evolutionary population with LLM-generated candidates while keeping the search budget and evolutionary procedure fixed.
- We empirically show that LLM-mixed seeding improves final instruction-token similarity over database-retrieved seeding under controlled experiments, and that this improvement is not explained only by a better starting point: after controlling for initial similarity, LLM-mixed trials close more of the remaining gap under the same evolutionary procedure.

2 Related Work

Traditional decompilation. Traditional decompilation has studied how to recover high-level, human-readable representations from binary programs. Early work by Cifuentes and Gough formulated decompilation as a program-analysis problem involving control-flow recovery, data-flow analysis, and reconstruction of high-level program structures [5]. Later work improved control-flow structuring and semantic fidelity for native binaries [3, 19], while complementary studies addressed type recovery, decompiler fidelity issues, and human reverse-engineering behavior [4, 6, 11]. These methods improve the readability, structure, and analysis utility of decompiler output, but they are not primarily designed to make recompilation reproduce the target binary. In contrast, our work focuses on byte-equivalent decompilation, where recovered source is evaluated by a recompilation-based equivalence metric.

Byte-equivalent decompilation and EED. Evolving Exact Decompilation (EED) introduced byte-equivalent decompilation as a search-based alternative to conventional decompilation [14]. Rather than directly reconstructing a readable program through static analysis, EED searches over candidate source programs, recompiles them, and evaluates the resulting binaries against the input binary. The search is initialized using candidate programs drawn from a source-code database and may also incorporate the outputs of existing decompilers. Candidate programs are then refined through an evolutionary search process guided by byte-level similarity. This design is related in spirit to search-based program synthesis and stochastic superoptimization, where programs are improved by exploring a space of semantically constrained candidates [13]. However, EED applies this search principle to decompilation: the target is not merely a faster or shorter program, but a source program whose recompilation reproduces the binary under analysis. To the best of our knowledge, the original EED system has not been available as a readily reusable reference implementation for independent experimentation. Our work therefore builds an independent EED-style implementation and studies whether LLM-generated seeds can improve the search process under a fixed computational budget.

ML- and LLM-based decompilation. Machine-learning-based decompilation has been explored as an alternative or complement to analysis-based methods. Katz et al. applied recurrent neural networks to decompilation [10], and Coda framed decompilation as an end-to-end neural translation problem [7]. More recent work has used language models for decompilation and decompiler-output refinement. SLaDe proposed a portable small language model decompiler for optimized assembly [2]. LLM4Decompile trained large language models for binary-code decompilation and evaluated them on benchmarks such as HumanEval-Decompile and ExeBench [1, 16].

Recent work has also begun to decompose LLM-based decompilation into more specific subtasks. SK2Decompile separates structure recovery from identifier recovery, distinguishing the reconstruction of source-level program structure from the restoration of meaningful identifiers [15]. DeGPT instead uses LLMs to refine decompiler output, with the goal of improving the readability and simplicity of existing decompilation results [8]. These methods show that learned models can produce useful source-like outputs from low-level code or improve existing decompiler results. However, their primary objectives are typically readability, compilability, re-executability, or semantic similarity, rather than explicit byte-equivalent recompilation of the recovered source. Our work uses LLMs in a different role: rather than treating the LLM output as the final decompilation result, we use it as part of the initial population for EED-style evolutionary search. This design uses LLM outputs as candidate starting points while retaining an explicit search procedure guided by recompilation-based equivalence checking.

3 Proposed Method

3.1 Overview

Figure 1 shows the overview of our proposed method. Given a target binary, the system extracts the target instruction sequence used as the reference for search. The goal is to synthesize a C source program whose recompilation produces an instruction sequence equivalent, or sufficiently close, to the target sequence. Unlike conventional decompilers, which primarily aim to recover human-readable source-like code, our method treats decompilation as a search problem over candidate source programs.

The proposed method follows an EED-style evolutionary search procedure. The system maintains a fixed-size population of candidate source programs. Each candidate is compiled, disassembled, normalized, and evaluated against the target. Candidates with better fitness are selected as parents, and the next generation is produced by applying crossover and mutation. This process is repeated until

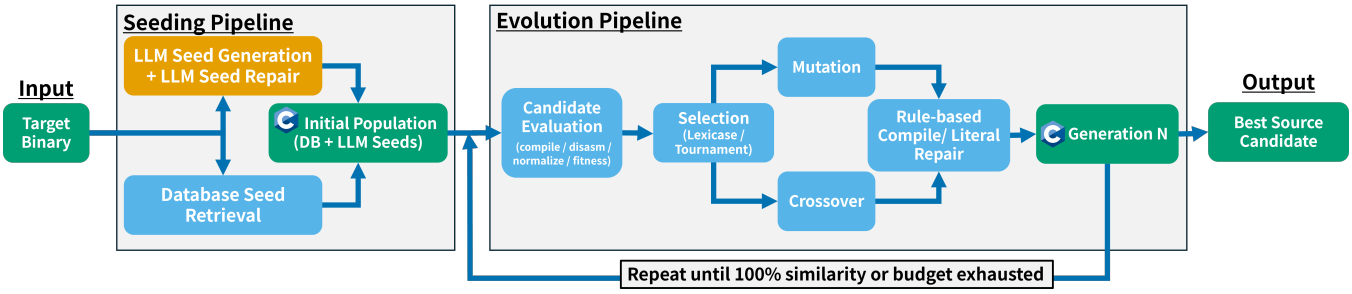


Figure 1: Overview of the proposed method.

a byte-equivalent candidate is found or a predefined computational budget is exhausted.

Our main extension is LLM-mixed seeding. Instead of initializing the population only with database-retrieved candidates, we replace part of the initial population with LLM-generated candidates. These LLM outputs are used only as initial seeds for evolutionary search.

3.2 Search Objective and Fitness Function

The search objective is to find a source program whose recompilation produces an instruction sequence close to that of the target binary. In this work, comparison is performed over normalized instruction tokens rather than raw bytes. Each token represents one disassembled instruction after operand normalization, combining the mnemonic with normalized operands.

Before comparison, the target binary and each compiled candidate are disassembled and normalized. The normalization removes padding instructions and rewrites address-dependent operands. Bare direct branch and call targets are represented as offsets from the instruction address, while known PLT targets are represented by their symbol names. For other operands, hexadecimal immediates are rewritten as section-relative offsets only when their values fall within the ELF `.text` or `.rodata` ranges; otherwise, they are preserved. RIP-relative memory operands are normalized from their effective addresses rather than from their encoded displacements. This normalization reduces differences caused by code placement while preserving the instruction-level structure used for search. Let $t = \langle t_1, \ldots, t_m \rangle$ be the normalized instruction-token sequence extracted from the target binary, and let $c = \langle c_1, \ldots, c_n \rangle$ be the corresponding sequence obtained by compiling and disassembling a candidate source program. We report the normalized LCS-based similarity as

$$\mathrm{sim}(t, c) = \frac{2 \cdot \mathrm{LCS}(t, c)}{|t| + |c|} \times 100,$$

where $\mathrm{LCS}(t, c)$ denotes the length of the longest common subsequence between the two normalized instruction-token sequences. This score ranges from 0 to 100, with 100 indicating that the two sequences are identical. The LCS-based similarity is used as the primary reporting metric. For evolutionary selection, the implementation uses the Levenshtein distance between the same normalized instruction-token sequences as the scalar fitness value, and additionally derives a region-wise fitness vector for lexicase selection.

3.3 Seeding Pipeline

The seeding pipeline constructs the initial population before evolutionary updates begin. Each seed is produced by inserting a candidate function body into a target-derived C template. The baseline configuration fills all seed slots using database-retrieved fragments, whereas the LLM-mixed configuration fills a configured fraction of the seed slots with LLM-generated function bodies and uses database-based seeding for the remaining slots.

Target-derived templates. For each target function, the system constructs a C template with a fixed function signature and a stub body. Candidate bodies obtained from the database or the LLM are spliced into this template, so the search operates on function bodies while keeping the surrounding compilation context fixed. In this prototype, target functions and signatures are taken from the reconstructed benchmark tasks; function discovery, signature inference, and type recovery are treated as separate front-end problems, rather than as contributions of the EED-style search evaluated in this paper.

Database-based seeding. For database-based seeding, the system retrieves candidate function definitions on a per-function basis using the target's normalized disassembly tokens and function names as retrieval signals. One retrieved function is then sampled according to the configured similarity-weighted sampling policy and spliced into the target-derived template. Since the original Project Euler corpus and the Pliny database infrastructure used by EED [9, 14] are not publicly available, we use Project CodeNet [12] as the replacement corpus and implement retrieval with SQLite.

LLM-mixed seeding. LLM-mixed seeding adds LLM-generated candidates to the initial population. As in the database-based path, LLM-based seeding is performed at the function level. For each function in the target-derived template, the corresponding disassembly is formatted and given to the LLM. The generated function body is then extracted and spliced into the template; functions without usable LLM output remain as template stubs. The resulting initial population consists of database-retrieved candidates and LLM-generated candidates. The fraction of LLM-generated candidates is controlled by a configuration parameter, while the remaining seed slots are filled by the database-based initialization procedure.

Seed-stage repair. Seed candidates may fail to compile after being inserted into the target-derived template. Such candidates are first repaired using the same bounded rule-based compile-repair

procedure used during candidate evaluation. If an LLM-generated seed remains uncompilable after rule-based repair, it may be passed to `gpt-4.1-mini` for compile-error repair. This LLM-based repair is applied only to uncompilable LLM-generated seed candidates during initialization; it is not applied to database-retrieved seeds and is not invoked in the evolution pipeline. Appendix E provides additional details on the LLM input format, output extraction, and seed-stage repair procedure.

3.4 Evolution Pipeline

After initialization, the system searches by iteratively editing a fixed-size population of C source programs. At each generation, every candidate is compiled, disassembled, normalized, and evaluated as described in Section 3.2. The evaluated candidates are selected as parents according to the configured selection strategy, and the next generation is produced by applying crossover and mutation. The search terminates when a byte-equivalent candidate is found or when the predefined evaluation budget is exhausted.

Candidate evaluation and repair. During evolution, candidates may fail to compile. Such candidates are repaired using a bounded sequence of rule-based fixes. If a candidate remains uncompilable after the configured number of repair attempts, it receives the worst fitness score. These repair steps are part of candidate evaluation and are separate from the genetic mutation operators.

The system also applies lightweight literal repair by replacing string and numeric literals with values mined from the target binary when the replacement improves fitness.

Selection and variation. Following EED, we implement tournament selection and lexicase selection as parent selection strategies, and use homologous crossover as the crossover operator. Tournament selection uses the Levenshtein distance between normalized instruction-token sequences as the scalar fitness, where lower values are better. This distance is used as the internal selection criterion, whereas the LCS-based similarity in Section 3.2 is used for reporting and for comparing candidate quality across experiments. Lexicase selection uses a region-wise fitness vector derived from local edit-distance scores. Mutation is performed at the level of AST statement ranges rather than raw text. We adopt the four basic EED mutation operators: CUT, INSERT, SWAP, and REPLACE. The source of inserted or replaced statements is controlled by mutation modifiers, which restrict whether candidate statements are taken from the current program or from the database.

4 Experiments and Evaluation

We evaluate LLM-mixed seeding against a database-seeded EED-style baseline under an otherwise shared evolutionary search procedure to answer two research questions.

- **RQ1.** Under a fixed evaluation budget, does LLM-mixed seeding improve the final similarity reached by the search, relative to the database-seeded baseline?
- **RQ2.** After accounting for the quality of the starting point, does the search close more of the residual gap to 100% similarity when initialized with LLM-mixed seeding than with the database-seeded baseline?

4.1 Experimental Setup

Configurations. We compare two initialization strategies. The *baseline* is a database-seeded EED-style configuration that constructs the initial population from database-retrieved source fragments. The LLM-mixed configuration replaces half of the initial population with candidates generated by `enhanced-llm4decompile-end-6.7b`, a decompilation-oriented code model. Uncompilable LLM-generated seed candidates first undergo rule-based compile repair and may then be passed to `gpt-4.1-mini` for compile-error repair. This seed-stage LLM repair is not applied to database-retrieved seeds and is not invoked during the evolution pipeline. The remaining half of the population is database-retrieved. Sections 3.3 and 3.4 describe these constructions in detail. The evolutionary loop is otherwise shared between the two arms. Therefore, the comparison evaluates the initialization strategy as a whole, including LLM-based seed generation and seed-stage LLM repair, rather than the isolated effect of raw LLM outputs.

Benchmarks and factors. We start from the 19 small C programs listed in Table II of the original EED paper [14]: 13 Learn C the Hard Way exercises and 6 Project Euler problems. Because the exact target sources used by EED were not released, we reconstruct each program from the corresponding exercise or problem statement. The Project Euler targets are therefore independent re-implementations, and the suite reproduces EED's targets as tasks rather than as byte-identical sources. We exclude one multi-function program because our current implementation targets a single function, leaving 18 targets. Each target is crossed with the compiler optimization level $opt \in \{\texttt{O0}, \texttt{O2}\}$ and the parent selection method $sel \in \{\texttt{lexicase}, \texttt{tournament}\}$. Each combination is run for three independent trials, yielding 72 conditions and 216 trials per arm. The compiled binaries serve as the ground-truth targets for byte-equivalence.

Budget. Both arms use a population of 256 candidates and a budget of 32,768 candidate evaluations per trial. Each candidate is given up to three rule-based compile-repair attempts. A trial terminates when a byte-equivalent candidate is found or when the evaluation budget is exhausted.

Compiler configuration. All target binaries and candidate programs are compiled with Clang 14.0.6 for x86-64. The common flags are `-m64 -g -fno-pie -no-pie`, and the optimization level is varied between `-O0` and `-O2`. Thus, each condition uses one of the two commands `clang -m64 -g -O0 -fno-pie -no-pie` or `clang -m64 -g -O2 -fno-pie -no-pie`.

Metrics. For each trial, we record its *final* similarity, defined as the best similarity value at termination under the LCS-based metric of Section 3.2. We also record its *generation-0* similarity, defined as the best similarity in the initial population. This value measures the quality of the starting point before evolutionary updates.

To separate the contribution of the search from the quality of the initial population, we define the *residual closure rate* of a trial as

$$\frac{final - gen0}{100 - gen0}.$$

This quantity measures the fraction of the remaining gap to 100% similarity that the search closes after generation 0. It is defined

Table 1: RQ1: per-condition difference in mean final similarity (Δ = LLM − baseline, in pp), with the Wilcoxon signed-rank test over paired condition means.

Stratum	n	Mean Δ	Median	Wilcoxon p
All	72	+17.90	+12.81	2.8×10^{-10}
`O0`	36	+11.81	+4.99	1.5×10^{-4}
`O2`	36	+23.98	+18.64	5.9×10^{-7}
lexicase	36	+19.34	+15.74	3.1×10^{-8}
tournament	36	+16.45	+6.14	5.7×10^{-5}

Table 2: Threshold attainment by optimization level: the fraction of each arm's trials whose final similarity reaches each threshold under the fixed evaluation budget.

	`O0` (n=108)		`O2` (n=108)	
Threshold	Base	LLM	Base	LLM
≥ 50%	59.3%	66.7%	18.5%	63.9%
≥ 60%	31.5%	51.9%	5.6%	47.2%
≥ 70%	14.8%	47.2%	5.6%	34.3%
≥ 80%	5.6%	38.9%	3.7%	17.6%
≥ 90%	0.0%	21.3%	0.0%	11.1%
100%	0.0%	11.1%	0.0%	6.5%

only for incomplete initial populations, i.e., trials with $gen0 < 100$. Trials whose generation-0 similarity is already 100% are therefore excluded from residual closure analyses.

4.2 RQ1: Final Similarity under a Fixed Budget

We first compare the two arms by the per-condition difference $\Delta = final_{\text{LLM}} - final_{\text{base}}$, where each term is the mean final similarity over the three trials in the same condition. We treat differences within ±1 percentage point (pp) as practical ties. Statistical significance is assessed using a sign test over the non-tied conditions and the Wilcoxon signed-rank test over paired condition means. Because the tests are applied to paired condition means, they do not treat the three repeated trials within a condition as independent observations. Table 1 reports the result. Across all 72 conditions, LLM-mixed seeding improves final similarity by a mean of +17.90 pp, winning in 54 conditions, tying in 12, and losing in 6. The sign-test result is $p = 9.7 \times 10^{-11}$, and the Wilcoxon signed-rank test gives $p = 2.8 \times 10^{-10}$. The advantage appears under both selection methods and both optimization levels. It is larger at `O2` (+23.98 pp) than at `O0` (+11.81 pp), suggesting that optimized targets are the setting where database-retrieved fragments provide a weaker starting point. The same pattern appears in threshold attainment. At both optimization levels and every similarity threshold, LLM-mixed seeding reaches the threshold in more trials than the baseline (Table 2). At `O2`, for example, LLM-mixed reaches at least 60% similarity in 47.2% of trials, compared with 5.6% for the baseline. LLM-mixed seeding also reaches 100% similarity in some trials, unlike the baseline.

Table 3: RQ2: residual closure by generation-0 similarity band. "closed" is the median residual closure rate among trials with $gen0 < 100$. "exact" counts trials in which search improves an incomplete seed to 100% similarity.

	Baseline			LLM-mixed		
gen0 band	n	closed	exact	n	closed	exact
$[0, 40)$	148	13.3%	0	104	15.1%	0
$[40, 50)$	34	18.7%	0	17	50.9%	0
$[50, 60)$	28	7.2%	0	39	48.1%	1
$[60, 95)$	6	0.0%	0	41	12.4%	3

4.3 RQ2: Complementarity of Seeds and Search

The RQ1 advantage could arise entirely from better starting points, with the evolutionary search adding little beyond the initial LLM seeds. A direct comparison of absolute post-generation-0 gains is therefore misleading: LLM-mixed trials often start closer to 100% similarity and have less remaining headroom for improvement. We therefore compare residual closure rates, which measure the fraction of the remaining gap to 100% similarity closed after generation 0. For example, improving from 40% to 70% and improving from 80% to 90% both close half of the remaining gap, even though the absolute gains are 30 pp and 10 pp, respectively. Residual closure is undefined when $gen0 = 100$, so the 15 LLM-mixed trials that are already exact at generation 0 are excluded from this analysis.

Table 3 summarizes the comparison by generation-0 similarity band. In the bands where both arms have non-trivial support, LLM-mixed seeding closes a larger fraction of the remaining gap: 50.9% versus 18.7% in the $[40, 50)$ band, and 48.1% versus 7.2% in the $[50, 60)$ band. At higher generation-0 similarity, the overlap becomes limited: only six baseline trials fall in the $[60, 95)$ band, compared with 41 LLM-mixed trials. We therefore treat this high-generation-0 band primarily as suggesting that LLM-mixed seeding reaches starting regions that the database-seeded baseline rarely reaches, rather than as a balanced common-support comparison.

As a robustness check, we also fit regressions of residual closure that adjust for generation-0 similarity, optimization level, selection method, and benchmark identity. The LLM-mixed indicator remains positive across these specifications, with an adjusted effect of about 16 percentage points in residual closure.

The residual-closure advantage could still be misleading if LLM seeding reduced the initial population to a small set of near-duplicate survivors, leaving the search little to recombine. We therefore measure initial-population diversity using the number of distinct fitness values among compilable candidates as a lower bound. This measure does not fully characterize semantic diversity, but it does not support the simplest collapse explanation: LLM-mixed populations compile more often (57.5% and 58.1% at `O0` and `O2`, versus 41.1% and 33.3% for the baseline) and contain roughly three to seven times as many distinct fitness values. Under this lower-bound measure, the residual-closure advantage is not consistent with reduced initial-population diversity under LLM-mixed seeding.

LLM-mixed seeding reaches 100% similarity in 19 trials, whereas the database-seeded baseline reaches none. These exact recoveries arise through two paths. In 15 trials, the initial population already

contains a candidate with 100% similarity, indicating that LLM-generated seeds can sometimes provide exact candidates under our normalized instruction-token metric without further evolutionary updates. In the remaining four trials, the initial population is incomplete, and the EED-style search closes the remaining gap to 100% similarity under the fixed budget. Thus, exact recovery under LLM-mixed seeding reflects both direct improvements in seed quality and cases where LLM-generated seeds place the search close enough for evolutionary refinement.

5 Discussion

5.1 Final Similarity under a Fixed Search Budget

The results for RQ1 show that LLM-mixed seeding improves the final similarity reached under a fixed evolutionary-search budget. This result indicates that the initialization strategy affects how much progress the same search procedure can make within a fixed number of candidate evaluations. In particular, LLM-generated seeds appear to provide starting candidates that are closer to the target under the normalized instruction-token similarity metric used in this work.

The larger gain at `O2` suggests that LLM-mixed seeding is especially useful when database-retrieved fragments provide weak initial candidates. In optimized binaries, source-level structure is more aggressively transformed by the compiler, and fragments retrieved from a code database may be less likely to align with the target's normalized instruction-token sequence. LLM-generated candidates do not solve this problem by themselves, but they can provide syntactically plausible programs that start closer to the target under the metric. These results suggest that LLM-mixed seeding can bias the evolutionary search toward different starting regions, while leaving the objective used to evaluate candidates unchanged. Despite the aggregate improvement, LLM-mixed seeding underperforms the baseline by more than 1 pp in six of the 72 conditions. Appendix D shows no common pattern in generation-0 compilability or fitness-value diversity across these cases: one condition shows strong concentration in a single fitness value, whereas the others are less concentrated or remain similar under this measure. These observations do not identify a single failure mechanism.

5.2 Residual Search Progress as Evidence of Complementarity

The results for RQ2 suggest that the benefit of LLM-mixed seeding is not limited to producing better generation-0 candidates. After accounting for generation-0 similarity, LLM-mixed trials close a larger fraction of the remaining gap to 100% similarity than database-seeded trials under the same evolutionary procedure. This is consistent with the interpretation that LLM-generated seeds can provide starting points from which the search process makes more effective progress. The search-attributable exact recoveries provide additional support for this interpretation. Some LLM-generated seeds are incomplete but close enough for the subsequent recompilation-guided search to close the remaining gap. This result should be interpreted as evidence of greater residual search progress under the tested benchmark and budget, not as a full causal decomposition of seed quality and search productivity. In this role, an LLM-generated candidate need not be trusted as correct; it only needs to provide a starting point from which recompilation-guided search can make progress under the target metric.

6 Limitations and Future Work

Scope and scalability. Our evaluation is limited to small single-function C programs reconstructed from the benchmark tasks used in the original EED paper. The results therefore do not establish that the same effect will hold for larger functions, multi-function programs, or substantially different source-code domains. Because the original EED implementation and exact benchmark sources are not publicly available, our results are not a numerical replication of EED. Extending the approach to realistic multi-function binaries would additionally require function discovery, signature and type recovery, and consistency across call sites and ABI-level argument passing. Moreover, our fixed evaluation budget does not directly characterize wall-clock scalability; larger-scale evaluations should therefore report both time-to-recovery and computational cost.

Evaluation objective and experimental assumptions. A similarity of 100% means that the target and candidate have identical normalized instruction-token sequences under our normalization procedure; it does not imply raw binary identity, semantic equivalence, or recovery of the original high-level program structure. Our results therefore evaluate the EED-style recompilation-guided search objective rather than source-level decompilation quality. Experiments also assume a known and fixed compiler and compilation configuration. Applying the same objective to an unknown stripped binary would require these conditions to be inferred or searched separately.

Future directions. Future work should evaluate the method on larger and more diverse benchmarks, including multi-function programs, additional compilers, and optimization settings. Another direction is to relax instruction-token equivalence toward structural or semantic objectives, such as control-flow or basic-block equivalence, although such objectives introduce different verification costs and ambiguities. On the model side, this paper evaluates LLM-mixed seeding as an initialization strategy, including seed-stage repair, rather than isolating the effect of raw LLM outputs. Evaluating newer code models and introducing LLMs as bounded repair or mutation operators are natural extensions, but such comparisons should explicitly account for LLM-query cost, token budget, and wall-clock time.

7 Conclusion

This paper evaluated LLM-mixed seeding for EED-style byte-equivalent decompilation under a fixed search budget. On small C programs reconstructed from the original EED tasks, LLM-mixed seeding improved final normalized instruction-token similarity by 17.90 percentage points over a database-seeded baseline. After accounting for generation-0 similarity and experimental factors, LLM-mixed trials also closed more of the remaining gap to 100%, indicating benefits beyond improved initial candidates.

References

[1] Jordi Armengol-Estap'e, Jackson Woodruff, Alexander Brauckmann, Jos'e Wesley de Souza Magalh aes, and Michael F. P. O'Boyle. 2022. ExeBench: An ML-Scale Dataset of Executable C Functions. In *Proceedings of the 6th ACM SIGPLAN*

International Symposium on Machine Programming (MAPS 2022). Association for Computing Machinery, New York, NY, USA, 50–59. doi:10.1145/3520312.3534867

[2] Jordi Armengol-Estap'e, Jackson Woodruff, Chris Cummins, and Michael F. P. 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. doi:10.1109/CGO57630.2024.10444788

[3] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 353–368. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz

[4] Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. 2022. Decomperson: How Humans Decompile and What We Can Learn From It. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2765–2782. https://www.usenix.org/conference/usenixsecurity22/presentation/burk

[5] Cristina Cifuentes and K. John Gough. 1995. Decompilation of Binary Programs. *Software: Practice and Experience* 25, 7 (1995), 811–829. doi:10.1002/spe.4380250706

[6] Luke Dramko, Jeremy Lacomis, Edward J. Schwartz, Bogdan Vasilescu, and Claire Le Goues. 2024. A Taxonomy of C Decompiler Fidelity Issues. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 379–396. https://www.usenix.org/conference/usenixsecurity24/presentation/dramko

[7] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An End-to-End Neural Program Decompiler. In *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*. 3703–3714. https://papers.neurips.cc/paper/2019/hash/093b60fd0557804c8ba0cbf1453da22f-Abstract.html

[8] Peiwei Hu, Ruigang Liang, and Kai Chen. 2024. DeGPT: Optimizing Decompiler Output with LLM. In *31st Annual Network and Distributed System Security Symposium (NDSS 2024)*. Internet Society, San Diego, CA, USA. doi:10.14722/ndss.2024.24401

[9] Chris Jermaine. 2016. The Pliny Database. http://cmj4.web.rice.edu/PlinyDBSlides.pdf. DARPA MUSE program, Rice University.

[10] Deborah S. Katz, Jason Ruchti, and Eric Schulte. 2018. Using Recurrent Neural Networks for Decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Campobasso, Italy, 346–356. doi:10.1109/SANER.2018.8330222

[11] Matthew Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*. Association for Computing Machinery, Santa Barbara, CA, USA, 27–41. doi:10.1145/2908080.2908119

[12] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, Joaquin Vanschoren and Serena Yeung (Eds.), Vol. 1. https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/a5bfc9e07964f8dddeb95fc584cd965d-Paper-round2.pdf

[13] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*. Association for Computing Machinery, Houston, TX, USA, 305–316. doi:10.1145/2451116.2451150

[14] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. 2018. Evolving Exact Decompilation. In *Workshop on Binary Analysis Research (BAR)*. San Diego, CA, USA. https://www.ndss-symposium.org/wp-content/uploads/2018/07/bar2018_8_Schulte_paper.pdf

[15] Hanzhuo Tan, Weihao Li, Xiaolong Tian, Siyi Wang, Jiaming Liu, Jing Li, and Yuqun Zhang. 2026. SK2Decompile: LLM-based Two-Phase Binary Decompilation from Skeleton to Skin. In *International Conference on Learning Representations (ICLR)*. Poster. https://openreview.net/forum?id=jSQPqdoidy

[16] 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*. Association for Computational Linguistics, Miami, Florida, USA, 3473–3487. doi:10.18653/v1/2024.emnlp-main.203

[17] Tree-sitter contributors. 2026. Tree-sitter. https://github.com/tree-sitter/tree-sitter. Parser generator tool and incremental parsing library. Accessed: 2026-06-28.

[18] Tree-sitter contributors. 2026. tree-sitter-c. https://github.com/tree-sitter/tree-sitter-c. C grammar for Tree-sitter. Accessed: 2026-06-28.

[19] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *22nd Annual Network and Distributed System Security Symposium (NDSS 2015)*. Internet Society, San Diego, CA, USA. doi:10.14722/ndss.2015.23185

Acknowledgments

This study was supported in part by JST Moonshot JPMJMS2215.

Generative AI Usage

The authors used ChatGPT[1] by OpenAI for language editing, phrasing suggestions, and limited revision of explanatory text. The authors used Claude Code[2] by Anthropic for assistance in developing code used in the experiments. All technical content, experimental results, statistics, citations and AI-generated code were manually verified by the authors.

A Experimental Configuration

The target suite consists of 18 single-function C programs: the six Project Euler problems `euler1`–`euler6` and the twelve Learn C the Hard Way exercises `hw1`, `hw3`, and `hw6`–`hw15` listed in Table 9. We start from the 19 programs of Table II of the original EED paper [14] and exclude `hw18`, which contains multiple functions, because the current implementation targets a single function. Because the exact sources used by EED were not released, each target is an independent reconstruction from the corresponding problem or exercise statement; the suite reproduces EED's targets as tasks rather than as byte-identical sources. Table 4 lists the search and seed-generation parameters shared by all conditions.

Table 4: Search and seed-generation parameters used in the experiments.

Parameter	Value
Population size	256
Evaluation budget	32,768 evaluations per trial
Compile-repair attempts	up to 3 per candidate
Selection (factor)	lexicase, tournament
Optimization level (factor)	`O0`, `O2`
Compiler	Clang 14.0.6, x86-64
Compiler flags	`-m64 -g -fno-pie -no-pie`
Trials per condition	3
Conditions / trials per arm	72 / 216
LLM-mixed seed share	0.5 of the initial population
Seed LLM	enhanced-llm4decompile-end (6.7B)
Seed LLM decoding	temperature 0.2, $\leq$ 512 new tokens
Snippet database (baseline)	Project CodeNet [12]
Snippet retrieval	BM25 over opcode sequences, edit-distance re-rank

B Full Per-Condition Results

Table 9 reports the mean generation-0 and final similarity of both arms for all 72 conditions. These rows are the source for the stratified aggregates in Section 4, including the per-stratum means, the win/tie/loss counts, and the per-benchmark differences. The six conditions with $\Delta < -1$ pp are the only negative entries in the Δ columns; they are discussed in Section 5 and Appendix D.

[1] chatgpt.com
[2] claude.com

Table 6: Regression analysis of residual closure after adjusting for starting similarity and experimental factors.

	Spec. 1 gen0 + opt	Spec. 2 + benchmark FE
LLM-mixed indicator β_{LLM}		
estimate	+0.120	+0.164
p, OLS	4.6×10^{-7}	1.3×10^{-12}
p, cluster by condition	4.1×10^{-4}	2.0×10^{-6}
p, cluster by benchmark	1.5×10^{-2}	2.4×10^{-3}
p, condition-mean	3.3×10^{-3}	4.3×10^{-5}
generation-0 coeff. (per pp)	−0.001	−0.004
opt = O2 coeff.	−0.118	−0.148
trials / conditions	417 / 140	

C Supporting Analyses for RQ2

Table 5: Naive decomposition of the final-similarity gain into generation-0 and post-generation-0 components.

Stratum	Δ	Seed	Search	$GAlift_{\text{base}}$	$GAlift_{\text{LLM}}$
All	+17.90	+17.29	+0.60	+12.53	+13.13
O0	+11.81	+17.26	−5.44	+17.93	+12.49
O2	+23.98	+17.33	+6.65	+7.13	+13.78

Table 5 gives a naive decomposition of the final-similarity gain into the generation-0 difference and the post-generation-0 search gain. The decomposition attributes most of Δ to the seed term and only a small part to search; at O0, the search term is negative. This interpretation is confounded by the starting point, because LLM-mixed trials start closer to 100% similarity and therefore have less absolute headroom. Table 6 addresses this confound by modeling residual closure while controlling for generation-0 similarity and experimental factors. The LLM-mixed indicator remains positive across specifications and remains significant under benchmark-clustered standard errors. The negative generation-0 coefficient captures the ceiling effect behind the naive decomposition.

D Generation-0 Diversity

Table 7: Generation-0 compilability and diversity by arm and optimization level.

Arm	*opt*	Compile	Unique fit.	Unique sim.
Baseline	O0	41.1%	3.3	4.9
LLM-mixed	O0	57.5%	10.8	15.0
Baseline	O2	33.3%	1.9	2.7
LLM-mixed	O2	58.1%	12.8	18.0

A possible alternative explanation is that LLM-mixed seeding produces a less diverse initial population, leaving less material for recombination. We use compilability and the number of distinct fitness values among compilable candidates as coarse indicators of generation-0 diversity. These measures do not characterize source-level or semantic diversity.

Table 8: Generation-0 diversity in conditions where LLM-mixed seeding underperforms the baseline.

Condition	Compile	Unique fit.	Top share
`hw3/o2/tournament`	2.2%→51.3%	2.0→3.3	69.4%→97.5%
`hw6/o0/tournament`	0.7%→43.4%	1.7→43.7	66.7%→10.1%
`hw7/o0/lexicase`	9.5%→54.3%	2.3→16.0	75.8%→39.7%
`hw7/o0/tournament`	9.5%→54.3%	2.3→16.3	75.8%→47.0%
`hw10/o0/lexicase`	15.0%→15.8%	3.0→4.7	64.0%→60.5%
`hw11/o0/tournament`	15.0%→15.8%	3.0→4.7	63.4%→60.0%

Table 7 does not support a general reduction in diversity under LLM-mixed seeding: LLM-mixed populations compile more often and contain several times as many distinct fitness values as the baseline. Table 8 focuses on the six conditions in which LLM-mixed seeding underperforms the baseline by more than 1 pp. Here, *Top share* denotes the fraction of compilable candidates having the most common fitness value. Only one condition shows strong concentration in a single fitness value; the remaining cases have lower or similar concentration under this measure. We therefore do not identify a common diversity-based explanation for these losses.

E LLM-Mixed Seeding Details

This appendix summarizes the implementation details of the LLM-mixed seeding procedure. LLM seeding is performed per function. For each function in the target-derived template, the system formats the corresponding target disassembly using the LLM4Decompile-style prompt:

```
# This is the assembly code:
<func_name>:
<one instruction per line>
# What is the source code?
```

The prompt contains only this assembly block and the function label. It does not include the C template, the target function signature, ABI hints, or other natural-language instructions. Instructions are emitted in program order using the mnemonic and operand string produced by Capstone. The model receives the raw pre-normalization disassembly; the normalized instruction-token form used for fitness evaluation is not used as model input.

The LLM output is parsed as a C translation unit using Tree-sitter with its C grammar [17, 18]. The system then searches the syntax tree for a function definition whose name exactly matches the requested function. If a match is found, the system extracts only the compound-statement body and splices it into the target-derived template. The generated signature is discarded. Outputs without a matching function definition, such as prose-only replies, wrong-name functions, or bare compound statements, are treated as unusable and leave the corresponding template stub unchanged.

Seed-stage LLM repair is applied only to generation-0 candidates produced by the LLM, before any evolutionary offspring are generated. Candidates first undergo the same bounded rule-based compile repair available to all arms. When LLM repair is triggered, the repair client constructs a chat-completion request containing

the full current C source and the latest `clang` stderr. The repair request uses the following fixed system prompt:

```
You are a C compiler-error repair assistant. The user
supplies broken C source plus clang's stderr.
Reply with ONLY a JSON object of the form
{"corrected_source": "<the entire repaired C
source>"}.
Do not include explanations, markdown fences, or any
text outside the JSON object.
```

The repaired source is recompiled, and the first repair attempt that compiles is adopted. If no repair attempt compiles within the configured repair budget, the candidate follows the same non-compiling penalty path as other candidates. This LLM repair is not applied to database-retrieved seeds and is not invoked after generation 0.

Table 9: Full per-condition results. Each cell is the mean over three independent trials. *gen0* is the best similarity in the initial population and *final* is the best similarity at termination, both as the LCS-based score in percent. $\Delta = final_{\mathbf{LLM}} - final_{\mathbf{base}}$ in percentage points. The two optimization levels `O0` and `O2` are shown side by side; rows are grouped by target and selection method.

		`O0`					`O2`				
		Baseline		LLM-mixed		Δ	Baseline		LLM-mixed		Δ
Target	sel	gen0	final	gen0	final	(pp)	gen0	final	gen0	final	(pp)
Project Euler											
`euler1`	lexicase	50.00	59.70	100.00	100.00	+40.30	6.56	6.56	36.07	65.99	+59.43
	tournament	50.00	60.00	100.00	100.00	+40.00	6.56	9.35	54.70	97.74	+88.39
`euler2`	lexicase	52.78	59.49	69.04	88.80	+29.32	38.50	37.88	100.00	100.00	+62.12
	tournament	52.78	52.08	72.98	91.18	+39.09	38.50	37.88	100.00	100.00	+62.12
`euler3`	lexicase	58.54	63.80	76.74	86.38	+22.58	24.30	42.45	24.30	84.26	+41.81
	tournament	58.54	61.54	76.74	81.88	+20.34	24.30	22.94	24.30	56.17	+33.23
`euler4`	lexicase	49.41	64.08	50.57	77.69	+13.61	14.88	18.42	38.53	70.38	+51.95
	tournament	49.41	54.68	50.57	57.03	+2.35	14.88	17.54	34.61	36.86	+19.32
`euler5`	lexicase	47.46	61.63	47.46	75.67	+14.04	18.75	19.56	25.35	31.57	+12.01
	tournament	47.46	54.09	53.86	59.83	+5.74	18.75	18.03	25.35	22.43	+4.41
`euler6`	lexicase	46.05	57.65	53.97	93.10	+35.46	53.33	53.33	69.17	69.17	+15.83
	tournament	46.05	49.50	53.97	87.36	+37.85	53.33	53.33	82.50	82.50	+29.17
Learn C the Hard Way											
`hw1`	lexicase	0.00	80.00	86.96	86.96	+6.96	58.97	58.97	55.56	76.92	+17.95
	tournament	0.00	80.00	84.64	88.53	+8.53	58.97	58.97	53.70	61.11	+2.14
`hw3`	lexicase	46.51	74.29	51.43	78.52	+4.23	58.33	58.33	48.41	58.14	−0.19
	tournament	46.51	67.13	51.43	67.40	+0.27	58.33	58.33	48.41	48.41	−9.92
`hw6`	lexicase	30.51	37.78	29.73	36.88	−0.89	14.78	32.15	15.84	36.31	+4.16
	tournament	30.51	37.29	30.05	31.83	−5.46	14.78	28.69	19.50	28.95	+0.26
`hw7`	lexicase	37.33	62.55	36.52	57.52	−5.03	36.84	47.96	32.28	63.61	+15.65
	tournament	37.33	50.96	36.52	44.61	−6.35	36.84	38.10	34.56	39.00	+0.91
`hw8`	lexicase	33.66	43.58	33.66	45.12	+1.54	10.39	27.36	60.00	60.00	+32.64
	tournament	33.66	34.62	33.66	34.62	0.00	10.39	26.18	57.17	57.63	+31.44
`hw9`	lexicase	25.28	40.24	26.60	39.52	−0.72	25.58	49.19	31.77	60.67	+11.48
	tournament	25.28	30.79	26.60	33.05	+2.25	25.58	40.19	29.54	39.92	−0.27
`hw10`	lexicase	32.60	55.53	32.60	52.05	−3.48	25.20	37.04	49.13	71.00	+33.96
	tournament	32.60	47.74	32.60	49.49	+1.75	25.20	33.10	54.75	67.12	+34.03
`hw11`	lexicase	34.12	48.43	34.12	55.68	+7.25	12.14	32.61	46.32	68.12	+35.51
	tournament	34.12	50.97	34.12	48.28	−2.69	12.14	31.88	40.70	69.75	+37.86
`hw12`	lexicase	43.55	56.71	67.03	85.51	+28.79	29.09	32.35	34.92	53.54	+21.18
	tournament	43.55	48.69	70.33	79.71	+31.03	29.09	33.33	34.34	44.20	+10.87
`hw13`	lexicase	31.93	38.35	31.93	39.88	+1.53	5.84	13.21	13.33	14.00	+0.79
	tournament	31.93	35.17	31.93	35.17	0.00	5.84	11.73	11.11	12.33	+0.60
`hw14`	lexicase	0.00	74.07	94.87	100.00	+25.93	30.43	32.32	65.54	70.67	+38.34
	tournament	0.00	74.07	97.44	100.00	+25.93	30.43	32.32	65.98	72.88	+40.56
`hw15`	lexicase	16.75	30.46	16.75	33.71	+3.26	8.88	26.54	24.94	43.50	+16.95
	tournament	16.75	20.69	16.75	20.69	0.00	8.88	24.13	26.88	30.67	+6.54