

000 001 002 003 004 005 006 007 008 009 010 011 012 REFINE, DON’T REWRITE: THE LEARNFROM FRAME- WORK FOR CONSISTENCY-AWARE LLM DECOMPILA- TION

013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
Anonymous authors
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053

Paper under double-blind review

ABSTRACT

Decompilation aims to translate binary executables into high-level source code; yet, the task remains demanding. Traditional tools, such as Ghidra, yield output that is structurally faithful but rarely recompilable or linkable. Recent methods built upon large language models improve readability and executability. However, unconstrained LLMs exhibit inherent stochasticity, they provide little guarantees of semantic consistency with the originating binary, which forces extensive cross-checking against the disassembly. We introduce the LearnFrom to address this challenge by leveraging traditional decompiler outputs and treating a code block corresponding to each control-flow graph node as a minimal editable unit to constrain modification scope. This patch generation design limits potential semantic deviations while reducing verification overhead. To achieve better model performance on the patch generation task in the context of decompilation, we further construct an open-source dataset of four million functions with explicit control-flow graph annotations, then use it to fine-tune the DeepSeek-Coder model series for specialized adaptation to the patch generation task. Within the LearnFrom framework, highly preservation of CFG structural consistency is enforced throughout the editing process, ensuring reliable control-flow alignment. Under this constraint, identical base models achieve a 5% improvement in HumanEval re-execution rates over baseline systems, while the edit similarity is increased by 6%, with further gains of 8% when substituting DeepSeek-Coder V2 followed by re-fine-tuning. These results confirm that CFG-aligned constraints offer the structural reliability essential for large-scale reverse engineering, while still permitting further optimization as foundational models advance.

1 INTRODUCTION

Decompilation recovers source-level abstractions from binaries by rebuilding control flow, data flow, and types into a readable, high-level code. Decompilation supports a broad range of reverse-engineering tasks, from vulnerability discovery and malware analysis to program comprehension, making it an essential technique in the field.

Traditional decompilers such as Ghidra National Security Agency (2024) and RetDec Avast Software rely on control-flow structuring and data-flow recovery to emit pseudo-C that is primarily designed for human consumption. However, numerous studies Liu & Wang (2020); Wang et al. (2017) report that these outputs are rarely readily recompilable and typically require substantial manual repair before execution, limiting programmatic use. Motivated by these limitations, recent work employs large language models—via end-to-end generation, refinement of decompiler output, or recompilation-oriented feedback loops—to improve readability and, increasingly, re-executability.

Despite impressive performance metrics, current large language model (LLM) methods leave a practical concern largely unaddressed: semantic consistency between the recovered code and the originating binary, a gap exacerbated by the intrinsic randomness of LLM. Reverse engineering, particularly for algorithm reversal and security analysis, has zero tolerance for error. For instance, a seemingly intelligent optimization can corrupt the entire logic by failing to handle negative inputs correctly, such as a model replacing signed multiplication with a bitwise shift. In a security con-

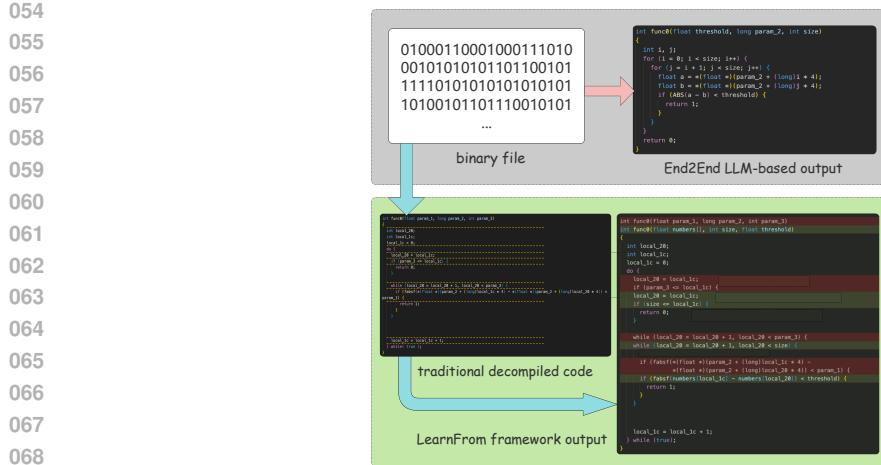


Figure 1: The upper panel illustrates the End2End LLM-based decompiler, the lower panel presents our LearnFrom approach. By integrating patch generation into the decompilation pipeline, the LearnFrom greatly reduces assembly-source verification effort.

text, an equivalent subtle error could neutralize a complex array bounds check, leading an analyst to approve vulnerable code and thereby invalidating the entire audit. Consequently, engineers must still audit the disassembly or decompiled code from traditional decompilers line-by-line, negating the promised efficiency gains.

A closer look at contemporary LLM-based pipelines helps explain the gap. Three dominant paradigms have emerged. First, exemplified by LLM4Decompile-End Tan et al. (2024), feeds assembly directly into a model that generates full C code in one pass. The second, adopted by DeGPT Hu et al. (2024) and the LLM4Decompile-Ref series, refines Ghidra’s pseudo-C with an LLM, yielding marked gains in recompilability while retaining the static analyzer’s structural precision. The third, represented by DecLLM Wong et al. (2025), loops between code generation and compilation or fuzzing; compiler diagnostics guide iterative fixes until all compiler tests pass. Although these diverse paths demonstrate an increasing awareness of the critical role of semantic consistency, their primary means of validation remain largely confined to techniques such as unit testing, fuzzing, and symbolic execution. While valuable for uncovering certain classes of errors, these methodologies are fundamentally limited by state explosion and are time-consuming, making it impractical to provide a sufficient security guarantee on their own. Therefore, rather than generating a flat sequence of executable code that requires extensive manual verification against low-level assembly, we aim to produce structurally reliable output that can be efficiently validated through limited comparison at the high-level language layer.

Therefore, we introduce the LearnFrom framework, which starts from the static output of a conventional decompiler and restricts the large language model to edit only the code basic block mapped to each control-flow graph (CFG) node, as shown in Fig.1. Each edit is strictly scoped to blocks, ensuring that the surrounding control structure and the original semantics remain intact. Experiments show that a domain-adapted model outperforms a generic counterpart on decompilation tasks, and that agent-based interactive iteration, as in approaches such as DeGPT, is likewise effective. We therefore combine both ideas, creating a 4-million-function code dataset in a fill-in-the-middle (FIM) task and fine-tune the DeepSeek-Coder series Guo et al. (2024) with LoRA Hu et al. (2022), endowing the resulting models with decompilation capabilities while preserving their natural-language conversational fluency.

Our approach ensures the integrity of the CFG structure while also improving the recompilation success rate, and because the fine-tuned model runs locally, it prevents proprietary code from leaving the premises.

This work makes three contributions.

- 108 • We present the LearnFrom framework, which couples CFG-indexed editing with a version
 109 control module and supports locally hosted models, turning one-off code regeneration into
 110 controlled, reversible code editing. LearnFrom preserves structural integrity and constrains
 111 semantic drift during iterative edits, greatly reducing verification effort.
- 112 • For achieving better model performance on the patch generation task in the context of de-
 113 compilation, we built a four-million-function dataset enriched with CFG annotations and
 114 designed for a two-stage FIM task that first uses single-mask and then multi-mask objec-
 115 tives.
- 116 • We deliver a model that integrates decompilation and interactive optimisation in a single
 117 workflow, enabling practical, stepwise improvements while maintaining structural guaran-
 118 tees. When evaluated under these high semantic-consistency guarantees, our model delivers
 119 state-of-the-art recompilation performance.

121 2 RELATED WORK

123 Large language models Devlin et al. (2019); Shazeer et al. (2017), are now widely investigated as
 124 stand-ins or complements for traditional decompilers. LLM4Decompile offers an influential exam-
 125 ple: trained on paired C source and assembly, its End-to-End variant lifts binaries directly to C,
 126 whereas its Refined variant edits Ghidra output to improve identifier quality and high-level idioms.
 127 The two modes highlight a familiar tension: unconstrained generation yields fluent code but risks
 128 control-flow drift, whereas refinement-based methods, though starting from an existing skeleton,
 129 still provide no guarantee of structural consistency. Nova Jiang et al. (2023) advances direct lifting
 130 with hierarchical attention and contrastive objectives, improving Pass@1 and Pass@10 accuracy on
 131 several lifting tasks, though it still has deviations in loop and branch semantics that surface under
 132 symbolic execution.

133 To make decompiled programs runnable, subsequent work tightened the interface between static
 134 analysers and language models. DeGPT preserves Ghidra’s skeleton but prompts a model to simplify
 135 types, merge redundant branches, and rename variables; human evaluators find the result markedly
 136 easier to read, although executability is not measured. DecLLM closes that gap by looping decom-
 137 pile edit compile until the patched file builds, feeding compiler diagnostics back into the model. Us-
 138 ing GPT-4 Achiam et al. (2023), without any additional fine-tuning, DeGPT and DecLLM achieved
 139 performance close to the state of the art. However, repeated iterations on the code can lead to se-
 140 mantic drift, and the search cost grows with function size and can stall when error messages are
 141 misleading. Slade Armengol-Estabé et al. (2024) introduces intermediate-assisted decompilation: it
 142 first predicts a pseudo-high-level intermediate representation that captures control-flow edges, then
 143 realises the final C code in a constrained decoding step, reducing semantic drift without iterative
 144 search.

145 Insights from patch generation further inform decompilation. AlphaRepair Xia & Zhang (2022)
 146 tackles bugs through zero-shot infilling—given context, the model predicts the missing correct
 147 line—outperforming classic APR systems on Defects4J with no task-specific fine-tuning, though
 148 it relies on accurate localisation of the buggy region. ReCoder Zhu et al. (2021) generates syntax-
 149 guided edit scripts that operate on the abstract syntax tree (AST); every edit preserves AST well-
 150 formedness, so patches compile by construction and surpass statement-level baselines. RepairL-
 151 LaMA Li et al. (2024) shows that minute LoRA adapters, trained on 4 k bug-fix pairs, can match far
 152 larger models, underscoring the power of task-specific adapters. SRepair Xiang et al. (2024) pushes
 153 efficiency by scoping edits to a single function: driven by failing tests, it fixes 300 of 522 Defects4J
 154 Just et al. (2014) bugs at an average cloud cost of three cents apiece and even repairs multi-function
 155 defects without explicit fault localisation.

156 Across these diverse systems, two recurring motifs emerge. First, constraining the model’s action
 157 space tends to maintain syntactic and semantic validity – whether by generating structured edits or
 158 by scoping the task to a single function or line. This reduces the chance of the model drifting into
 159 irrelevant or invalid code. Second, using external feedback loops improves outcomes – be it compi-
 160 lation success signals, runtime tests, or other diagnostics that steer successive attempts. Techniques
 161 like DeGPT’s error-informed loop and SRepair’s use of test failures show that incorporating envi-
 162 ronment feedback helps the model converge to a correct solution instead of blindly trusting its first
 163 guess.

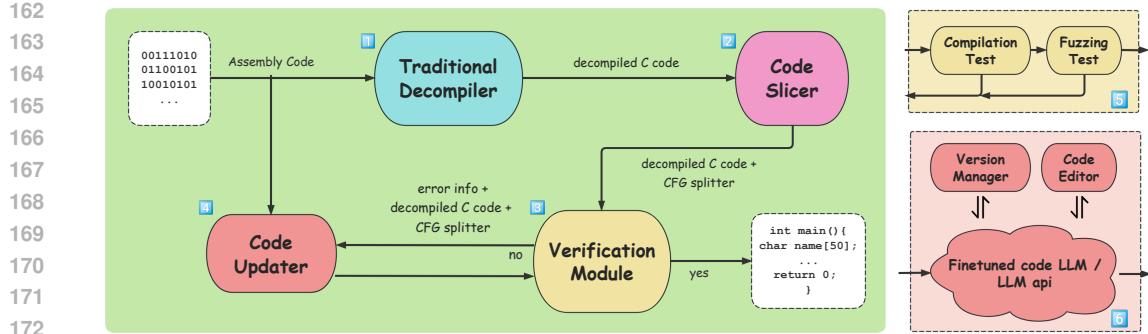


Figure 2: Overview of the LearnFrom framework.

3 METHOD

3.1 OVERVIEW

The overall workflow of LearnFrom is illustrated in Fig. 2. Given a binary executable, we first feed it to a conventional decompiler to obtain an initial pseudo-C. The code is passed to the Code Slicer module, which performs static analysis, reconstructs the control-flow graph, and inserts a comment token `cfg_splitter` at the first line of each code block corresponding to a CFG node—hereafter referred to as a basic block —while simultaneously assigning every basic block a unique identifier. Practical considerations are detailed in CFG Extraction section, such as how to treat comments, blank lines and other non-semantic elements. The annotated code is then subjected to ordinary compilation test and to libFuzzer fuzzing test; if both checks succeed the pipeline terminates, otherwise it proceeds to the next stage. At this point the LearnFrom model receives a compound prompt that includes the `cfg-splitter`-tagged decompiler output, the original assembly for context and the current compilation error messages. The model is first asked to identify the blocks that require modification and then, through an external script (Code Editor), to rewrite only those blocks. Whenever the LLM produces a new basic block, we rigorously verify that the occurrence and placement of branch keywords—such as `if` and `while`—match corresponding original block. The revised programme is compiled and fuzzed once more; should it still fail, control passes to Version Manager that enforces code version control designed to prevent the model from drifting through repeated erroneous edits and to limit resource consumption. Once the code passes both compilation tests and fuzzing tests, it is emitted as the final output; during testing, the code is fed into the test-case module.

3.2 CODE SLICER

Currently, no off-the-shelf tool can comprehensively map CFG nodes to their corresponding code blocks in C and C++. We therefore developed Code Slicer, which reports the first-line number and an index for each block, allowing LLMs to locate code problems quickly, telling LLMs the minimal editable unit. It parses the listing with an incremental syntax engine rooted in the Tree-Sitter grammar for C and C++, walks the resulting concrete syntax tree, and recognises every construct that delimits control flow—conditional branches, loops, switch cases, early returns, labelled statements, and nested compound blocks. For each such boundary it records the first source line belonging to the block and inserts a lightweight comment token that later serves as an anchor.

3.3 FINE-TUNING

An agent composed of Code Slicer, Version Manager and a lightweight external editing script, paired with a general LLM, can reproduce the full LearnFrom workflow. However, most general-purpose and code-oriented large LLM lack native decompilation capabilities, and their re-execution success rate within LearnFrom still lags behind that of highly specialised systems such as LLM4Decompile. Paradoxically, LLM4Decompile’s extensive task-specific training leaves it unable to interpret compiler diagnostics, so it cannot drive the iterative repair loop required by LearnFrom.

216 We assembled a parallel corpus of source code and its corresponding assembly output and then fine-
 217 tuned several code LLMs with a parameter-efficient LoRA scheme, which lowers training overhead
 218 while preserving the model’s capacity for interactive refinement. Each source-assembly pair is refor-
 219 mulated as a fill-in-the-middle (FIM) reconstruction task: Code Slicer annotates every control-flow
 220 graph (CFG) node with a `cfg-splitter` tag; a subset of those blocks is masked, while the compiled
 221 assembly is supplied as additional context. The model must therefore infer and restore the missing
 222 fragments with instruction-level fidelity.

223 To sharpen the model’s grasp of program structure, we retain the CFG annotations during fine-
 224 tuning. The annotation allows the LLM to reason over the relationships between execution paths and
 225 their semantic implications rather than merely imitating surface syntax, helping LLM to outperform
 226 in this specific task. The curriculum progresses in two stages of increasing difficulty. During the
 227 initial phase the model sees single-mask samples, mirroring the pre-training paradigm of DeepSeek-
 228 Coder and ensuring a smooth hand-over to our task domain. The second phase raises the bar by
 229 masking multiple, interdependent blocks within the same function, thereby forcing the network
 230 to co-ordinate several local reconstructions under a global consistency constraint and significantly
 231 strengthening its reasoning capability.

232 When deciding how to expose the assembly context, we compared two alternatives: (i) indexing
 233 the assembly into an external, queryable memory and (ii) injecting the raw assembly directly into
 234 the prompt. Because typical functions produce a manageable amount of machine code—and the
 235 second option reduces system complexity—we adopt the direct-injection strategy. Empirically, this
 236 assembly-aware reconstruction mechanism enables the model to recover developer intent more faith-
 237 fully than naive text-only variants.

238

239 4 EXPERIMENTS

240

241 4.1 CFG EXTRACTION

242

243 Our pipeline begins with a structural analysis pass that converts each compilation unit into a control-
 244 flow graph view whose vertices correspond exactly to source-level basic blocks. To recover the
 245 blocks, we parse the original `.c` or `.cpp` file with the *tree-sitter* incremental parsing library,
 246 whose C API and Python bindings expose a full concrete syntax tree with constant-time random
 247 access to node boundaries. We configure the parser with the official `tree-sitter-cpp` gram-
 248 mar. The parse tree is then traversed in preorder; whenever the visitor encounters a node whose
 249 `type` matches an entry in the hand-curated set, including `if_statement`, `while_statement`,
 250 `for_statement`, `switch_statement`, and other control-flow statements. The opening line
 251 number of that node is recorded as a split point. Edge cases are handled explicitly to ensure that
 252 every recorded split line corresponds to a semantically meaningful basic block. First, non-semantic
 253 tokens—braces, null statements, comments, and all pre-processor directives—are skipped so that
 254 purely syntactic elements never define a block boundary. Second, consecutive declaration or ex-
 255 pression statement nodes are merged into a single fall-through block, preventing an explosion of
 256 one-line blocks in straight-line code. Third, labeled statements are treated specially: the label itself
 257 is ignored, but the statement that follows is promoted to its own block so that indirect jumps remain
 258 well-formed in the reconstructed CFG. Pragma-style annotations such as `#type` are currently ig-
 259 nored, and no block tag is emitted for them. Nested compound statements are traversed recursively,
 260 and any duplicate line numbers collected from overlapping walks are removed at the end, guaran-
 261 teeing a strictly increasing, unique sequence of split points. Finally, if the parser encounters malformed
 262 input, the analyser returns an empty split list together with an error flag, allowing downstream com-
 263 ponents to fall back to the original decompiler output without interrupting the pipeline.

264

265 4.2 DATASETS

266

267 The corpus is built from decompile bench, a parallel collection of source files and their compiled
 268 assembly, stored as arrow records that contain the source string, the original file path, an optional
 269 symbol name and the corresponding assembly. Preprocessing proceeds in a uniform pipeline for C
 and C++, with a minor difference in the front-end used to recover control-flow boundaries. Each
 record is first normalised by extracting the `code`, `file-suffix` and `asm` fields. The suffix determines

```

270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323

```

```

input:
Context: A C/C++ source code with
several lines obscured, control-
flow split points are tagged with
"//<cfg splitter>". The
complete assembly generated from
the unobscured version of that
source. Task: Use the assembly
and the CFG split tags to infer
the original C/C++ code for each
masked block. Output the
recovered blocks in numerical
order.

MASKED_CODE:
{masked code}

ASSEMBLY:
{assembly code}

output:
{recovered code}

```

```

(assembly code)
pushq %rbp
movq %rsp, %rbp
subq $0x570, %rsp
# imm = 0x570
movq %rdi, -0x8(%rbp)
movq %rsi, -0x10(%rbp)
movq %rax, -0x548(%rbp)
leaq -0x1b0(%rbp), %rdi
movq %rdi, -0x540(%rbp)
callq 0xf97c0
movq -0x548(%rbp), %rdi
movq -0x540(%rbp), %rsi
callq 0x6703a0
...
```

```

(recovered code)
<| fim_holestart |>    ExceptionOr<DepT> depResult;
                           getDepResult(depResult);<| fim_holeend |>

(masked code)
<| fim_begin |>void getImpl(ExceptionOrValue& output) override {
    typedef ::FixVoid<::ReturnTypes<Func, _DepT>> T;
    typedef ::FixVoid<_DepT> DepT;
    <| fim_hole |>
    KJ_IF_SOME(depException, depResult.exception) //<cfg splitter>
        output.as<T>() = ExceptionOr<T>{false, kj::mv(depException)}; //<cfg splitter>
    } else KJ_IF_SOME(depValue, depResult.value) //<cfg splitter>
        output.as<T>() = handle(MaybeVoidCaller<DepT, T>::apply(func, kj::mv(depValue))); //<cfg splitter>
    }
}<| fim_end |>

```

Figure 3: It presents an example from the training set, illustrating the dataset’s composition.

which parser is applied, while headers or unsupported suffixes are skipped. The extractor returns an ordered list of source line numbers, each marking the first line of a block.

We traverse the block identifiers to obtain the index and line range of every region, yielding a deterministic mapping from each identifier to a half-open interval that defines what can be masked. Masking policies differ only in the number of blocks removed: the single-mask phase randomly drops one block, whereas the multi-mask phase omits 15 % of the blocks, rounding up when fewer than one would otherwise be selected. If no split points are detected, the entire file is replaced by a single mask. When constructing the masked source, the covered lines are deleted and a placeholder is left on each block’s first line; blank lines from the original file are preserved. Because real-world code carries irregular line endings and erratic indentation, additional normalisation steps are applied to ensure stable parsing. Each data instance, illustrated in Fig.3, is divided into an input part and an output part. The input part begins with a brief natural-language preamble that states the task, followed by two components: the masked source fragment delimited by FIM markers and a verbatim copy of the assembly emitted from the unmasked code. The output part lists the masked basic blocks in their original order. All placeholders are standardised to a single hole token to reduce lexical fragmentation, and the ground-truth blocks are concatenated in ascending block-index order so their original sequence is implicit. Both training stages are produced by the same pipeline. The first stage, limited to single masks, stabilises learning and echoes the model’s pre-training regime, offering short, well-anchored snippets whose recovery depends mainly on local context and explicit control-flow cues. The second stage increases difficulty by masking multiple, often non-contiguous blocks in the same function, forcing the network to use CFG anchors and assembly guidance for globally consistent reconstruction. For both phases the corpus is split at file level in a 15 : 1 : 1 ratio into the respective training sets and a shared validation set, with ExeBench Armengol-Estabé et al. (2022) and HumanEval-Decompile held out for testing. Strict checks ensure that any newly generated block preserves the presence and placement of branching keywords such as if and while, guaranteeing high CFG-level consistency.

HumanEval-Decompile is a compact benchmark distilled from the 164 tasks of OpenAI’s HumanEval suite: each Python reference solution was mechanically translated into a single, self-contained C function together with its original assertions. Every task depends only on the standard C library, so the code compiles out-of-the-box with GCC and isolates pure algorithmic reasoning without project-specific baggage. ExeBench, by contrast, draws 5,000 real-world C functions mined from diverse GitHub projects and paired with I/O examples. The corpus retains user-defined structures, type aliases and helper routines, offering a much richer syntactic and semantic landscape than HumanEval-Decompile and mirroring production code complexity across nine established metrics.

324 These characteristics make ExeBench a tougher test of decompilation: successful recovery requires
 325 not only control-flow fidelity but also accurate reconstitution of custom data types and external in-
 326 teractions.
 327

328 4.3 CONFIGURATION

330 We fine-tune the models in two curriculum stages that mirror the construction of the corpus. In both
 331 stages the backbone weights remain frozen and only LoRA adapters are updated. Unless otherwise
 332 stated, all runs use bf16 arithmetic, AdamW with a fixed learning rate of 1×10^{-4} , gradient clipping
 333 at 1.0, a warm-up ratio of 0.1, per-device batch size of 8 sequences, gradient accumulation of 8
 334 steps, and a maximum sequence length of 1,024 tokens. We employ gradient checkpointing and
 335 DeepSpeed ZERO-2 sharding to fit the memory footprint while keeping optimiser states offloaded
 336 to host memory when beneficial. The LoRA configuration follows the script: rank $r = 8$, $\alpha = 32$,
 337 dropout 0.1.
 338

339 Experiments are executed on two GPU pools: a node equipped with eight NVIDIA A100 40,GB
 340 cards and a second node with four NVIDIA A800 80,GB cards. The A100 node is used for most
 341 ablations and all StageI runs for both backbones. The A800 node provides the extra headroom
 342 needed for the largest configuration on DeepSeek-Coder-v2 Zhu et al. (2024); with the above hyper-
 343 parameters, this job requires nine days of wall-clock time from the start of StageI to the end of
 344 Stage II, including validation and checkpointing. Smaller variants, such as the v1 backbone or re-
 345duced curriculum lengths, complete proportionally faster on the A100 pool. To avoid I/O contention
 346 on the shared filesystem we redirect all HuggingFace, Torch and CUDA caches to a dedicated NVMe
 347 path.
 348

349 Convergence is tracked on the held-out validation split from each stage. StageI reduces validation
 350 perplexity rapidly over the first epoch and then saturates, at which point we transition to StageII
 351 using the best StageI adapter as initialisation. StageII exhibits slower but steady improvements,
 352 consistent with the increased difficulty of multi-mask reconstruction. The curriculum consistently
 353 yields the best recompilation success when both stages are completed; omitting StageI or training
 354 StageII from scratch degrades performance.
 355

Model/Benchmark	HumanEval-Decompile					ExeBench				
	O0	O1	O2	O3	Avg	O0	O1	O2	O3	Avg
LLM4Decompile-End-6.7B	68.05	39.51	36.71	37.20	45.37	22.89	16.60	16.18	16.25	17.98
Ghidra+LLM4Decompile-Ref-1.3B	68.90	37.20	40.85	37.20	46.04	-	-	-	-	-
Ghidra+LLM4Decompile-Ref-6.7B	74.39	46.95	47.56	42.07	52.74	-	-	-	-	-
DecLLM	80.49	65.24	65.5	58.5	67.43	-	-	-	-	-
Ghidra+LFMv1-1.3B+LFW	65.54	45.90	46.61	37.70	48.94	24.44	17.97	17.83	16.60	19.21
Ghidra+LFMv1-6.7B+LFW	87.39	49.53	48.16	44.88	57.49	2.87	22.53	22.24	22.10	24.19
Ghidra+LFMv2+LFW	88.60	55.57	52.45	48.06	61.17	33.55	22.72	22.31	22.26	25.21
IDA Pro+LFMv2+LFW	88.21	58.76	53.80	42.77	60.89	34.10	22.20	22.97	22.99	25.57
IDA Pro+Claude 4.0+LFW	50.57	48.04	49.05	33.18	45.21	20.09	16.19	15.48	15.66	16.86

363 Table 1: Comparison of each method’s re-execution rate across different test sets and compiler-
 364 optimization levels. The + sign indicates that the preceding decompiler is used together with the
 365 LearnFrom framework. LFM stands for LearnFrom Model, i.e., the model we obtained by fine-
 366 tuning with our method, whereas v1 and v2 refer to different versions of the DeepSeek Coder base
 367 model.
 368

369 4.4 RESULTS

370 While LearnFrom’s chief contribution is to guarantee trustworthy CFG alignment in LLM-based
 371 decompilation, we can still compare its effectiveness with prior work by evaluating the re-execution
 372 rate—the proportion of recovered functions that both compile and pass the original I/O test harness—
 373 on HumanEval-Decompile and ExeBench across optimisation levels O0 to O3. Because
 374 the framework enforces strict basic-block boundaries, the recovered code already preserves CFG
 375 fidelity; in our view, meeting this level of semantic consistency is a prerequisite rather than a per-
 376 formance target, so traditional edit-similarity metrics, which is used in earlier work, are no longer
 377 informative and are therefore omitted.
 378

Model/Optimization	O0	O1	O2	O3	Avg
Ghidra+LLM4-Decompile-Ref-6.7B	15.59	13.53	13.42	12.73	13.82
CIM 6.7B+CIW	49.16	40.25	39.54	39.02	41.99
Ghidra+LFMv1+LFW	48.77	42.53	36.96	37.15	41.35

Table 2: Main comparison of Edit Similarity with different methods on the HumanEval-Decompile benchmark, where L2R denotes a fine-tuning task implemented in the conventional left-to-right next-token prediction format.

Model/Optimization	O0	O1	O2	O3	Avg
Ghidra+LLM4-Decompile-Ref-6.7B	74.39	46.95	47.56	42.07	52.74
Ghidra+DeepSeek-Coder-6.7B+LFW	58.52	33.97	31.56	31.95	39.00
Ghidra+DeepSeek-Coder-6.7B-L2R+LFW	76.02	47.50	47.97	42.43	53.48
Ghidra+LFMv1+LFW	89.39	47.53	48.16	44.88	57.49

Table 3: Main comparison for the ablation study, where L2R denotes a fine-tuning task implemented in the conventional left-to-right next-token prediction format.

We first reproduce LLM4Decompile-End, the current open-source state of the art for End-to-End binary decompilation. On HumanEval-Decompile, 6.7 B model attains an average re-execution rate of 45.37 %, and only 17.98 % on ExeBench. Adding Ghidra’s structured output as a front-end raises the average to 52.74 %, confirming earlier observations that injecting coarse syntactic hints helps large sequence models align with compiler idioms .

We fine-tune DeepSeek-Coder checkpoints with Low-Rank-Adaptation (LoRA) layers using our four-million-function, CFG-annotated FIM corpus. With LoRA, the 6.7 B model already surpasses the reference by +4.9 % absolute on HumanEval, confirming that parameter-efficient tuning can endow a code LLM with decompilation priors at modest cost . Our 6.7 B-parameter system surpasses the strongest available 22 B end-to-end model in re-execution performance. A lightweight 1.3 B variant remains competitive, trailing the 6.7 B LLM4Decompile reference by just 3.8 %.

To maximise performance, we fine-tuned the 16 B parameter DeepSeek-Coder-V2-Lite-Instruct model, paired it with the state-of-the-art IDA Pro decompiler, and integrated it into the LearnFrom framework, securing an improvement of nearly 10 % while maintaining high CFG structural consistency. On HumanEval-Decompile the fine-tuned DeepSeek-Coder-V2 with LearnFrom reaches 61.17 % average re-execution, a +15.8 pp improvement over the best end-to-end model. ExeBench rises to 25.57 %. When we replace our tuned model with Claude 4 Opus and run it through LearnFrom, performance climbs from 21.22 % to 45.21 % on HumanEval, closely matching the 46.04 % achieved by the much larger 1.3 B LLM4Decompile-Ref. This confirms that LearnFrom’s block-scoped repair loop is itself responsible for roughly half of the observed accuracy gains.

We also conducted experiments on code edit similarity. As shown in Tab. 2, which presents the edit similarity between decompiled and source code, our method significantly outperforms the baseline. It is worth noting that CIM and CIW Liu et al. (2025), a paper that has not undergone peer review, report a slightly higher similarity score than ours. However, their work does not address CFG-level consistency.

4.5 ABLATION STUDY

We conducted a series of ablations to isolate the contribution of individual components. First, to evaluate the efficacy of our fill-in-the-middle formulation we replicated the data construction strategy of LLM4Decompile but substituted LoRA fine-tuning so that the base model retained interactive editing capability . Under identical hyper-parameters, the LoRA variant improves re-execution by two percentage points over the original LLM4Decompile reference, whereas our block-aligned FIM

432
433
434
435
436

Model/Benchmark	O0	O1	O2	O3	AVG
LFMv1 w/o Fuzzing	83.83	48.77	43.98	42.52	54.78
LFMv1 w Fuzzing	87.39	49.53	48.16	44.88	57.49

437
438
439

Table 4: Ablation study on fuzzing test module.

440

task yields the larger gains reported in Tab. 3, confirming that structural cues—not merely additional training—drive the improvement.

441

We removed the fuzz-testing stage from the validation loop and relied exclusively on compilation feedback. Many patches that passed the compiler subsequently crashed when re-executed, lowering the overall re-execution rate by three percentage points, as in Tab. 4, and confirming prior work that fuzzing is indispensable for revealing latent semantic defects that elude static checks .

442

We also disabled the Version Manager, allowing the agent to iterate without checkpoint control. Re-execution variance increased sharply and wall-time nearly doubled, indicating that disciplined state management is critical for stable convergence, in line with recent observations on iterative code-repair pipelines . Finally, we replaced the external code editor with a prompt that asked the language model to “use the fewest edits possible.” Although this shortcut occasionally sped up convergence, the model still altered CFG structure or semantics in a substantial fraction of cases, even when explicitly instructed to avoid such changes, echoing earlier findings on the limits of prompt-only control .

443

Taken together, these ablations show that the gains stem from the joint effect of CFG-constrained masking, fuzz-based validation, and version-controlled iteration. Semantic fidelity to the original program is treated as a prerequisite rather than a tunable metric; therefore we do not report edit-similarity scores that earlier studies used for looser, token-level comparisons.

446

447

5 CONCLUSION

448

In summary, LearnFrom demonstrates that constraining a fine-tuned code model with CFG granularity, staged FIM curricula, and a limit-rollback schedule can systematically raise recompilation success—delivering 5 pp and 8 pp absolute gains on HumanEval Chen et al. (2021) when applied to DeepSeek-Coder v1 and v2 respectively, while retaining the readable output that recent LLM-based decompilers have highlighted as their chief advantage. At the same time, LoRA adaptation keeps training economical and privacy-preserving on commodity clusters, confirming previous observations about its efficiency in large-model fine-tuning.

449

Although these results close much of the gap between classical tools—whose strength lies in structural fidelity—and end-to-end neural synthesis, they do not yet match the interactive conveniences reverse engineers rely on daily: mainstream decompilers such as Ghidra and Hex-Rays let analysts jump from a high-level variable to its exact machine-level definition, preserving a one-to-one mapping between symbolic names and assembly addresses for rapid cross-reference. Enabling the same clickable correspondence in a learned editing pipeline remains an open challenge; doing so will likely require augmenting the language model with a symbolic store of variable lifetimes and pointer provenance or with a post-processing layer that reconciles generated identifiers against the original program’s data-flow. We leave this integration of semantic links—and its attendant UI affordances—as future work, anticipating that a tight loop between learned refinement and source-binary cross-navigation will be the next decisive step toward a practical, engineer-grade neural decompiler.

480

481

6 STATEMENT

482

483

6.1 ETHICS STATEMENT

484

485

This work adheres to the ICLR Code of Ethics. The research is entirely computational and does not involve human subjects, sensitive personal information, or the collection of private data. The

486 datasets used are publicly available and have been widely employed in the research community,
 487 and we followed responsible practices in data usage and documentation. The methods proposed
 488 are intended for scientific research and education, and we are not aware of any immediate harmful
 489 applications. We have made efforts to avoid introducing or amplifying unfair bias and to ensure that
 490 the results respect principles of fairness, integrity, and transparency. The authors declare that there
 491 are no conflicts of interest or external sponsorships that could compromise the integrity of this work.
 492

493 6.2 REPRODUCIBILITY STATEMENT

494 We have taken active measures to ensure the reproducibility of our results. All key details of the pro-
 495 posed method, including model architectures, training procedures, hyperparameters, and evaluation
 496 metrics, are described in the main paper and appendix. To further facilitate reproducibility, we will
 497 provide an anonymous link to the source code and scripts used for data preprocessing, model train-
 498 ing, and evaluation in the supplementary material. We include a clear description of data processing
 499 steps for all datasets used in experiments. Together, these resources should enable independent
 500 researchers to reproduce and verify our findings.
 501

502 REFERENCES

- 503 Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Ale-
 504 man, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical
 505 report. *arXiv preprint arXiv:2303.08774*, 2023.
- 506 Emmanuel Ameisen, Jack Lindsey, Adam Pearce, Wes Gurnee, Nicholas L. Turner, Brian Chen,
 507 Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar,
 508 Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan,
 509 Adam Jermyn, Andy Jones, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman,
 510 Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. Circuit tracing: Revealing
 511 computational graphs in language models. *Transformer Circuits Thread*, 2025. URL <https://transformer-circuits.pub/2025/attribution-graphs/methods.html>.
- 512 Jordi Armengol-Estabé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Maga-
 513 lhaes, and Michael FP O’Boyle. Exebench: an ml-scale dataset of executable c functions. In
 514 *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp.
 515 50–59, 2022.
- 516 Jordi Armengol-Estabé, Jackson Woodruff, Chris Cummins, and Michael FP O’Boyle. Slade: A
 517 portable small language model decompiler for optimized assembly. In *2024 IEEE/ACM Interna-
 518 tional Symposium on Code Generation and Optimization (CGO)*, pp. 67–80. IEEE, 2024.
- 519 Avast Software. RetDec: A retargetable machine-code decompiler. <https://retdec.com/>.
- 520 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared
 521 Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large
 522 language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- 523 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep
 524 bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of
 525 the North American chapter of the association for computational linguistics: human language
 526 technologies, volume 1 (long and short papers)*, pp. 4171–4186, 2019.
- 527 Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao
 528 Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming—
 529 the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.
- 530 Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang,
 531 Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.
- 532 Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. In *Pro-
 533 ceedings 2024 Network and Distributed System Security Symposium*, volume 267622140, 2024.

- 540 Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Nova: Gen-
 541 erative language models for assembly code with hierarchical attention and contrastive learning.
 542 *arXiv preprint arXiv:2311.13721*, 2023.
- 543
- 544 René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable
 545 controlled testing studies for java programs. In *Proceedings of the 2014 international symposium*
 546 *on software testing and analysis*, pp. 437–440, 2014.
- 547
- 548 Yuan Li, Yue Wang, Xiang Chen, Zhiqiang Liu, Shuai Wang, Yuqun Zhang, and Lingxiao
 549 Chen. Repairllama: Efficient program repair with large language models. *arXiv preprint*
 550 *arXiv:2407.12345*, 2024.
- 551
- 552 Peipei Liu, Jian Sun, Rongkang Sun, Li Chen, Zhaoteng Yan, Peizheng Zhang, Dapeng Sun, Dawei
 553 Wang, Xiaoling Zhang, and Dan Li. The codeinverter suite: Control-flow and data-mapping
 554 augmented binary decompilation with llms. *arXiv preprint arXiv:2503.07215*, 2025.
- 555
- 556 Zhibo Liu and Shuai Wang. How far we have come: Testing decompilation correctness of c decom-
 557 pilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing*
 558 *and Analysis*, pp. 475–487, 2020.
- 559
- 560 National Security Agency. *Ghidra Software Reverse Engineering Framework*. NSA Research Di-
 561 rectorate, 2024. URL <https://github.com/NationalSecurityAgency/ghidra>.
- 562
- 563 Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton,
 564 and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer.
 565 *arXiv preprint arXiv:1701.06538*, 2017.
- 566
- 567 Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with
 568 large language models. *arXiv preprint arXiv:2403.05286*, 2024.
- 569
- 570 Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen,
 571 Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*,
 572 2017.
- 573
- 574 Wai Kin Wong, Daoyuan Wu, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen
 575 Nie, and Shi Wu. Decllm: Llm-augmented recompilable decompilation for enabling program-
 576 matic use of decompiled code. *Proceedings of the ACM on Software Engineering*, 2(IST):
 577 1841–1864, 2025.
- 578
- 579 Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated
 580 program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software*
 581 *Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 959–
 582 971, 2022.
- 583
- 584 Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Zizheng Zhang, Haotian Zhang, and
 585 Yuqun Zhang. How far can we go with practical function-level program repair? *arXiv preprint*
 586 *arXiv:2404.12833*, 2024.
- 587
- 588 Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. A
 589 syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM joint meet-
 590 ing on European software engineering conference and symposium on the foundations of software*
 591 *engineering*, pp. 341–353, 2021.
- 592
- 593 Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li,
 594 Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models
 595 in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.

594 **A APPENDIX**
595596 **A.1 LLM USAGE STATEMENT**
597598 Large language models were employed in this work only in a limited and carefully managed capac-
599 ity. Their use was confined to linguistic refinement, including grammar checking, stylistic polishing,
600 and the improvement of textual fluency. In addition, LLMs were cautiously utilized to generate a
601 few minor auxiliary code modules and simple test scripts, but they played no role in shaping the
602 research questions, designing or implementing the central methods, executing experiments, or an-
603 alyzing results. The intellectual conception of the study, the methodological framework, the core
604 system development, and the interpretation of findings were conducted solely by the authors. All
605 content produced with the assistance of LLMs was rigorously reviewed and revised to ensure both
606 accuracy and originality. The authors accept full responsibility for the manuscript in its entirety,
607 and, consistent with ICLR policy, LLMs are not recognized as contributors or authors of this work.
608609 **A.2 VERSION MANAGER**610 During iterative repair we consider two principal revision paradigms. The first relies exclusively
611 on compiler diagnostics: the model emits large numbers of candidate patches, each is compiled
612 and, when possible. The second paradigm spawns a branch for every edit—much like a Git work-
613 flow—and delegating to a supervisory agent the choice of which branch to pursue or abandon. Our
614 preliminary experiments showed that LLM appears unable to decide which branch is closest to the
615 correct solution, choosing branches randomly; they may even propagate errors by iterating on in-
616 correct branches, and excessively long contexts lead to performance degradation. It touches an ever
617 larger set of blocks and consumes disproportionate time. And because our setting constrains code
618 changes to pre-defined basic blocks, the search space is modest, the former mode’s time cost is
619 acceptable.620 Therefore, we integrate the two modes and design a Version Manager: the LLM can decide whether
621 it would edit on the last version or rollback to the pristine baseline, and it is allowed to stay on its
622 editing branch for a maximum of three consecutive edits, and if compilation still fails, the system
623 rolls back to the pristine baseline; overall, no more than four such rollbacks are permitted during
624 evaluation. In practice this cap yields a favourable balance between search depth and reliability,
625 delivering the desired accuracy without excessive resource expenditure.626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647