# HELIOS: Hierarchical Graph Abstraction for Structure-Aware LLM Decompilation

Yonatan Gizachew Achamyeleh
University of California, Irvine
yachamye@uci.edu

Harsh Thomare
University of California, Irvine
hthomare@uci.edu

Mohammad Abdullah Al Faruque
University of California, Irvine
alfaruqu@uci.edu

***Abstract*—Large language models (LLMs) have recently been applied to binary decompilation, yet they still treat code as plain text and ignore the graphs that govern program control flow. This limitation often yields syntactically fragile and logically inconsistent output, especially for optimized binaries. This paper presents HELIOS, a framework that reframes LLM based decompilation as a structured reasoning task. HELIOS summarizes a binary's control flow and function calls into a hierarchical text representation that spells out basic blocks, their successors, and high level patterns such as loops and conditionals. This representation is supplied to a general purpose LLM together with raw decompiler output, optionally combined with a compiler in the loop that returns error messages when the generated code fails to build.**

**On HumanEval-Decompile for `x86_64`, HELIOS raises average object file compilability from 45.0% to 85.2% for Gemini 2.0 and from 71.4% to 89.6% for GPT-4.1 Mini. With compiler feedback, compilability exceeds 94% and functional correctness improves by up to 5.6 percentage points over text only prompting. Across six architectures drawn from x86, ARM, and MIPS, HELIOS reduces the spread in functional correctness while keeping syntactic correctness consistently high, all without fine tuning. These properties make HELIOS a practical building block for reverse engineering workflows in security settings where analysts need recompilable, semantically faithful code across diverse hardware targets.**

## I. Introduction

Reverse engineering of binary code is a foundational, yet demanding, activity in software security. It underpins tasks such as malware analysis, vulnerability triage, interoperability, and maintenance of legacy systems [1], [2]. This work remains a persistent bottleneck, since it requires specialists to reconstruct high level logic from low level, often obfuscated machine instructions. As software complexity and hardware diversity grow [3], the demand for skilled reverse engineers exceeds the available expertise [4], [5], [6], [7].

The community has explored two main directions for automation. Classical decompilers, such as Ghidra [8] and IDA Pro [9], translate assembly into C like pseudo code and are now standard tools in reverse engineering practice. Their output, however, is often syntactically fragile, poorly typed, and preserves convoluted control flow that still requires substantial manual effort to clean up [10], [11], [12]. More recently, large language models (LLMs) have been used to push past these limitations. Current work either fine tunes LLMs directly on pairs of binaries and source code [13], [14] or prompts general purpose models to rewrite decompiler output [15], [16], [17]. These approaches differ in training strategy, but they share a common assumption: the binary and its decompiled form can be treated as a one dimensional sequence of tokens. We refer to this family as *structurally blind* (or structurally agnostic) reasoning, since the model receives little or no explicit information about the control flow graph that actually governs program behavior.

Studies of how expert reverse engineers work highlight what this structurally blind view is missing. Observational work [1], [18] shows that analysts do not simply read assembly in a linear fashion. Instead, they build and refine a mental model of the program by reasoning over graphs, primarily the control flow graph (CFG), supported by the function call graph (FCG) and data flow information [19]. This matches a long standing observation in program analysis: graph based representations are a natural abstraction for software, and they are widely used for security tasks such as automated bug finding, function prediction and fuzzing [20], [21], [22], [23], [7].

At the same time, current LLMs are text native. They are trained to predict token sequences rather than to traverse graphs. This tension raises a natural question: can we bring the structural view that human analysts use into the token space where LLMs operate, without retraining the model itself. There has been progress on general graph to text encodings [24], [25], [26], but these methods focus on abstract graphs and do not address the specific structure of binaries, such as compiler level intermediate forms, low level control flow, and architecture specific idioms.

This paper introduces HELIOS, a framework that adapts these ideas to the domain of binary decompilation. We reframe LLM assisted decompilation as context aware structural reasoning (Path A in Figure 1). HELIOS uses a static analysis backend to derive control flow and call graphs, then encodes this information into a hierarchical textual representation. At a high level, the prompt contains (i) a summary of the function and its role, (ii) a compact description of the CFG and its main paths, and (iii) block level details that link individual basic blocks back to the raw decompiler output. A small set

Workshop on LLM Assisted Security and Trust Exploration (LAST-X) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-05-3
https://dx.doi.org/10.14722/last-x.2026.23079
www.ndss-symposium.org

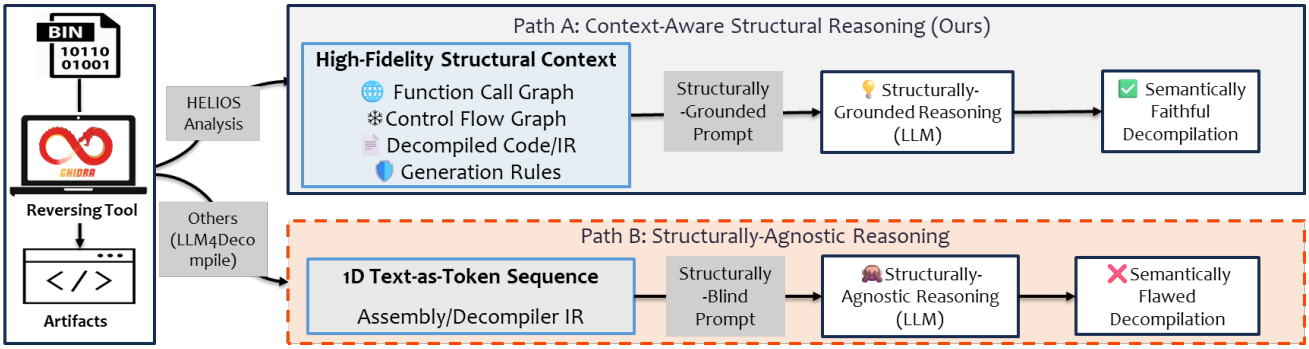


Fig. 1. An illustration comparing two approaches for LLM based decompilation. **Path B (Structurally-Agnostic)**, used by existing methods, treats binary artifacts as flat text. **Path A (Context-Aware)**, our proposed approach, creates a structural context that allows the LLM to reason over control flow and produce semantically faithful decompilation.

of natural language rules guides the model on how to use this structure, and an optional compiler in the loop provides error messages when the generated code fails to compile. The result is a fine tuning free, architecture agnostic pipeline that encourages the model to use control flow information in a way that is closer to how human analysts work.

In this work, we make the following contributions:

- We propose a method for encoding the control flow and call graphs of a binary, together with high level structural patterns such as loops and conditionals, into a compact textual format that can be consumed directly by general purpose LLMs.
- We instantiate this method in HELIOS and evaluate it on HumanEval-Decompile and MBPP, showing large gains in compilability and consistent improvements in functional correctness over text only prompting and over state of the art fine tuned decompilers, while using only a small fixed number of model calls and one optional feedback iteration.
- We conduct, to our knowledge, the first cross architecture study of structure aware LLM decompilation across six instruction sets, demonstrating that a single prompt design can generalize across x86, ARM, and MIPS without retraining, which is directly relevant for settings such as firmware and IoT analysis.

## II. Background and Related Work

Our work connects three areas: the representation of binary programs for analysis, the evolution of automated decompilation techniques, and the emerging use of LLMs for reasoning over graph-structured data.

### A. Program Representation for Binary Analysis

Graph-based structures are a standard way to represent the logic of a binary program. The Control Flow Graph (CFG), which models possible execution paths within a function, and the Function Call Graph (FCG), which captures relationships between functions, are widely used in program analysis [27], [28], [23], [7]. In security, these graph structures are the main data structures for tasks such as automated vulnerability discovery and fuzzing [29], [30], [22]. A large body of work has shown that reasoning over these graphs is central to deep program understanding and to discovering subtle bugs [20], [21].

### B. Automated Decompilation

The goal of decompilation is to translate low-level machine code back into high-level source code. This is an ill-posed problem, since compilation is inherently lossy [31]. Traditional, rule-based decompilers such as Ghidra apply complex heuristics but often produce brittle and semantically incomplete code, for example, by emitting poorly typed variables or misrepresenting control flow [11], [32]. As a result, human analysts still spend substantial effort correcting and simplifying decompiler output.

*LLM-based decompilation.* The emergence of large language models has led to two main lines of work on automated decompilation. The first, which we refer to as *end-to-end decompilation*, uses large-scale fine-tuning. Systems such as LLM4Decompile [13], SLaDe [33], and Nova [14] train models on paired source and assembly corpora to translate directly from binaries to high-level code. These approaches can produce high-quality results on the architectures and optimization settings they are trained on, but they require substantial resources and are typically tied to specific instruction sets such as x86_64. Adapting them to new architectures or to new base models usually requires another round of training.

The second line of work focuses on *decompiler output refinement*. Frameworks such as DeGPT [15] and LMPA [2], as well as follow-up work [10], prompt general-purpose LLMs to clean up or explain pseudo-code produced by existing decompilers. These tools are primarily designed to improve readability and analyst productivity rather than to generate fully recompilable code in a fully automated loop. They mostly operate on the raw decompiler text, with limited access to the underlying control flow or intermediate representations.

Both families of approaches treat the binary and its decompiled form as flattened token sequences. In this sense they are structurally blind, or structurally agnostic, because they do not expose the graph structure that program analysis tools use

internally. In contrast, our work keeps the decompiler in the loop. It feeds a general-purpose LLM an explicit, multi-level description of the CFG and related context, aiming to improve recompilability and functional correctness across architectures without additional fine-tuning.

### C. Graph Reasoning with Large Language Models

The mismatch between text native LLMs and graph-structured data has led to a growing literature on methods that linearize graphs into text that an LLM can process [24], [25], [26]. These techniques show that it is possible to encode nodes, edges, and paths into token sequences in a way that preserves useful structural information and allows the model to perform tasks such as classification, question answering, or link prediction over graphs.

Our work builds on these ideas in a domain-specific way. Rather than targeting generic graphs, HELIOS focuses on the control flow and call graphs that arise in binary analysis, together with the `P-Code` level operations attached to each basic block. The framework constructs a hierarchical textual representation with three main components: a function-level summary that captures inter-function structure and coarse semantic patterns, a logical flow section that presents intra-function control flow and explicit successor relationships, and a block-level view that provides the low-level evidence. On top of this structure, HELIOS adds a small set of natural language rules that constrain how the model should use the graph information, and an optional compiler feedback loop.

In summary, prior work has established graph-based representations as central for binary analysis and has applied LLMs to decompilation and graph reasoning, but these lines of work are largely separate. Fine-tuned decompilers optimize for direct translation at the cost of architectural specialization, and prompt-based refinement tools mainly target readability while remaining structurally blind. HELIOS combines program analysis, graph-aware prompting, and compiler feedback to encourage a general-purpose LLM to reason about control flow and correctness rather than only about textual similarity, and we evaluate this design across multiple architectures without any task-specific training.

## III. THE HELIOS FRAMEWORK

To address the challenge of "structural blindness" in current LLM-based decompilation, we designed and implemented HELIOS, a modular, end-to-end framework for augmenting LLMs with the rich, structural context of a binary. Instead of treating decompilation as a text-to-text translation task, HELIOS reframes it as a context-aware reasoning problem. The core principle of our framework is to provide the LLM not only with the decompiled pseudo-code but also with a textual representation that models the program's underlying graph structures at multiple levels of abstraction. This approach is designed to mimic the analytical process of a human expert, who synthesizes information from various representations to form a holistic understanding of the program.

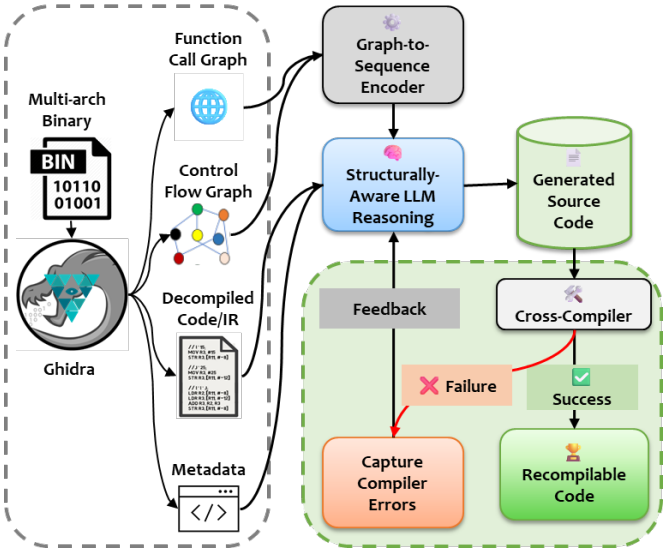


Fig. 2. The high-level architecture of the HELIOS framework. A binary is first analyzed to extract the decompiled code and its CFG. HELIOS then abstracts the CFG and combines it with other metadata to form a context-rich prompt for the LLM, enabling structurally-aware decompilation and optional iterative refinement via compiler feedback.

### A. System Overview

The HELIOS pipeline, shown in Figure 2, operates per function and follows a simple pattern. Given a binary, we run a static analysis backend to recover pseudo-C code, control-flow graphs, and related metadata. For each function, we then build a hierarchical textual summary of this structure, attach the raw decompiler output, and pass the result to a general-purpose LLM. An optional compiler-in-the-loop checks the generated code and, if necessary, triggers a single round of repairs.

For clarity, we refer to the main stages as:

1) **Static analysis and feature extraction**, which collects per-function artifacts such as pseudo-code, CFG, and call information.
2) **Hierarchical prompt generation**, which encodes these artifacts into the HELIOS prompt format.
3) **Structurally aware prompting**, which uses a fixed instruction template plus the prompt to guide the LLM.
4) **Iterative refinement with compiler feedback**, which is an optional second pass based on compiler diagnostics.

The following subsections describe the analysis, prompt format, and prompting strategy in more detail.

### B. Static Analysis and Feature Extraction

The first stage relies on the rich artifacts produced by modern reverse engineering platforms. Our implementation uses the Ghidra Software Reverse Engineering Framework [8] and its headless analyzer, driven by a script that processes whole binaries in a reproducible way. The design is tool-agnostic and could be implemented on top of other analysis frameworks with similar capabilities.

For each function, we extract:

- the decompiler's C or C-like pseudo code, including the function signature and local variables,
- the complete control flow graph (CFG) derived from Ghidra's `P-Code` intermediate representation [34], including a list of basic blocks and directed edges between them [35],
- the interprocedural function call graph (FCG), restricted to calls that originate in the current function, and
- auxiliary metadata such as loop headers, string references, imported functions, and constant values.

During this pass, we also record a mapping between each basic block and the corresponding region of the decompiled pseudo-code. This mapping allows the prompt generator to refer to blocks in a stable way and to cross-check CFG edges against the textual output. All subsequent stages operate on these per-function summaries rather than on raw binaries.

### C. Hierarchical Abstraction and Prompt Format

The second stage converts the analysis artifacts into a structured prompt. The goal is not only to linearize the CFG, but to present the program at several levels of abstraction that mirror how a human analyst would approach it. We adopt a simple segment based format, illustrated in Figure 3, with four main sections:

1) **`[FUNCTION_CONTEXT]`** A top-level summary that includes the function name and signature, the target architecture, and coarse structural statistics such as the number of basic blocks and loops. This gives the model a quick mental picture of the function before it sees any code.
2) **`[CFG_OVERVIEW]`** A compact description of the CFG that lists, for each basic block, its successors and any notable role such as loop header or branch target. This section provides a readable map of the possible execution paths without exposing low level instructions.
3) **`[BLOCK_DETAILS]`** A block by block summary that contains distilled `P-Code` instructions for each basic block, with stable block identifiers that match those used in the cfg overview. This serves as the main source of semantic evidence for the model.
4) **`[RAW_DECOMPILED_CODE]`** The original pseudo C output from the decompiler, presented without modification. This gives the model a baseline implementation to refine.

In a typical case, `[FUNCTION_CONTEXT]` is only a few lines, `[CFG_OVERVIEW]` is a list of tens of basic blocks, and `[BLOCK_DETAILS]` contributes a few hundred tokens of `P-Code`. The complete prompt for a single function remains well within the context limits of current models.

This hierarchical layout is intended to address structural blindness directly. The `[FUNCTION_CONTEXT]` section supplies the high-level structure, such as the presence of loops or early returns. The `[CFG_OVERVIEW]` section materializes the CFG as a simple adjacency list that the model can reason over. The `[BLOCK_DETAILS]` section

```
[System Prompt]
[Critical Rules]
[FUNCTION_CONTEXT]
 ; This section summarizes function, its signature,
architecture, statistics, external calls, and high-level
patterns.
  Name: func0
  Signature: int func0()
  Architecture: x86:LE:64:default
  Summary: 11 blocks, 12 edges. Contains 1 loop
 ...

[CFG_OVERVIEW]
  ; High-level logical structure of the function, derived from
CFG and pattern analysis.
  ; FORMAT: BLOCK_LABEL -> [SUCCESSORS]
  BLOCK_0 -> [BLOCK_1]
  BLOCK_1 -> [BLOCK_2]
  BLOCK_3 -> [BLOCK_10, BLOCK_4]
...
  BLOCK_10 ; Exit Block

[BLOCK_DETAILS]
  ; This section provides the low-level P-code for each block,
along with explicit predecessor and successor relationships.

  [BLOCK id=BLOCK_0 type=entry]
    [SUCCESSORS: BLOCK_1]
    [PCODE]
      (unique, 0x10000088, 8) PIECE (register, 0x3c, 4)
      ---  CALL (ram, 0x10012c, 8)
      (stack, 0xfffffffffffffe4, 4) COPY (const, 0x0, 4)
    [END_BLOCK]
...
  [BLOCK id=BLOCK_10 type=exit]
    [PREDECESSORS: BLOCK_3]
    [PCODE]
    [END_BLOCK]

[RAW_DECOMPILED_CODE]
```

Fig. 3. The Multi-Part Prompt Format of HELIOS. Our prompt provides a hierarchical view of the binary, starting with a high-level **`[FUNCTION_CONTEXT]`**, followed by the **`[CFG_OVERVIEW]`** (a topological map of the CFG), and finally, the low-level evidence in **`[BLOCK_DETAILS]`** before the **`[RAW_DECOMPILED_CODE]`**.

grounds that reasoning in concrete low-level operations. Finally, `[RAW_DECOMPILED_CODE]` anchors the output to the existing decompiler, so the model can focus on correcting and simplifying rather than inventing a completely new implementation.

### D. Structurally Aware Prompting and Iterative Refinement

In the final stage, HELIOS turns the structural summary into a concrete instruction for the LLM. The prompt follows a universal template that has three parts: a short description of the task, a list of critical rules that the model should follow, and the four structural sections described above.

The critical rules are based on a manual study of common failure modes we observed when prompting LLMs with decompiler output. We grouped the failures into a small number of recurring patterns, including:

- reimplementing standard library functions instead of calling them,
- producing control flow that does not match the CFG, for example, by inventing extra branches or omitting error paths, and

- introducing type mismatches, especially in code compiled with high optimization levels, where type information is partially erased.

Each pattern is expressed as a short natural language rule in the system prompt, for example, that all branches in the generated code must correspond to branches in `[CFG_OVERVIEW]`, and that new global variables must not be introduced unless they are present in `[BLOCK_DETAILS]` or `[RAW_DECOMPILED_CODE]`. The ablation study in Table V shows that adding these rules on top of the structural sections yields a large jump in compilability, which suggests that guidance on how to use the structure is as important as the structure itself.

To further improve robustness, HELIOS can optionally run a compiler feedback loop. After the first model call, the generated C code is compiled with the same toolchain used to create the original binary. If compilation succeeds and the resulting object file links into a test harness, the process stops. If compilation fails, the compiler diagnostics are appended to the prompt in a dedicated section, and the model is asked to correct the code while maintaining control-flow consistency with `[CFG_OVERVIEW]` and `[BLOCK_DETAILS]`. This produces a second candidate, which we compile again and then evaluate with unit tests. In all reported experiments, we use at most one such feedback iteration per function, which keeps the overall cost predictable while still yielding large gains in compilability and functional correctness.

## IV. EXPERIMENTAL SETUP

To evaluate the efficacy of HELIOS, we designed a series of experiments to answer the following research questions, framed to assess both the practical security application and the underlying AI advancements:

**RQ1:** Does providing explicit structural graph context to an LLM lead to a significant improvement in the quality and correctness of decompilation compared to approaches that reason over unstructured text alone?

**RQ2:** How effectively does HELIOS generalize to different hardware architectures compared to structurally-blind methods?

**RQ3:** To what extent does an optional, iterative compiler feedback loop improve the rate of syntactically correct and re-executable code generation?

**RQ4:** What is the individual contribution of each component in the HELIOS hierarchical prompt to the overall improvement in decompilation quality?

### A. Datasets

Our experiments use two widely adopted code generation benchmarks, **HumanEval** [36] and **MBPP** [37]. Both provide natural language specifications and high-quality unit test suites, which we use as an objective measure of functional correctness. Following prior work on LLM-based decompilation, such as LLM4Decompile [13], we use the publicly available *C-converted* versions of these benchmarks, which supply C implementations aligned with the original Python tasks and compatible test suites.

From these C programs, we construct our primary dataset, the **Cross-Architecture Decompilation Database (Cross-Arch-DB)**. For each task, we compile a self-contained C implementation across **six** target architectures: `x86_32` (i686), `x86_64`, `arm_32`, `aarch64`, `mips_32`, and `mips_64`. These targets cover the three most common instruction set families and span both CISC (x86) and RISC (ARM, MIPS) designs. Each binary is compiled with GCC 11.4 using four optimization levels (`-O0` through `-O3`). The resulting binaries and their associated test suites form **Cross-Arch-DB**, which we use to study both architecture-specific behavior (RQ1) and cross-architecture generalization (RQ2). When discussing results on decompiled outputs, we refer to the HumanEval-derived test cases as **HumanEval-Decompile** and the MBPP-derived test cases as **MBPP-Decompile**.

### B. Evaluation Metrics

To evaluate the quality of the generated code, we adopt four metrics established in prior work on decompilation correctness and code generation [11], [36], [13], [33].

- **Object File Compilability (Re-compilability):** The percentage of all functions that successfully compile into an object file (`.o`). This serves as our check for syntactic & type correctness.
- **Executable Linkability:** The percentage of all functions that can be successfully linked into a complete executable binary, a stricter test that also verifies the resolution of all external symbols.
- **Functional Correctness (Re-executability):** The percentage of all functions that produce a valid executable *and* pass their original ground-truth test suite. This is our strictest measure of semantic preservation.
- **Edit Similarity:** To measure the textual closeness to the original source, we use a metric based on the normalized Levenshtein edit distance. A higher score indicates a more textually accurate reconstruction [13].

### C. Baselines

To rigorously evaluate the contributions of HELIOS, we compare it against two representative baseline categories: (1) an **LLM-Text-Only** baseline that provides the LLM with only the raw Ghidra decompiled text, and (2) **LLM-Finetuned** models, using the publicly available *Nova* [14] and *LLM4Decompile* [13] models (6.7B and 1.3B variants) as representatives of the state-of-the-art fine-tuning approach. In line with [13], we do not compare against SLaDe [33], as its methodology requires intermediate compiler artifacts that are unavailable in the realistic black-box scenario our work addresses.

### D. Implementation Details

Our HELIOS prototype is built using Ghidra 11.0 for static analysis and Python 3.10 for the prompt generation modules. For our experiments, we use two general-purpose

LLMs, `gpt-4.1-mini` and `gemini-2.0-flash`, chosen for their balance of reasoning capability, cost, and inference speed. Experiments were orchestrated from a server with an NVIDIA A100 GPU, while all model inferences are issued via hosted APIs. We will release our implementation, prompts, and evaluation scripts, together with instructions for reconstructing Cross-Arch-DB, upon acceptance.

We report our primary experimental results, including performance across architectures, optimization levels, and evaluation metrics, in the main body of the paper. The Appendix provides additional evaluations and details, including the standardized NOVA evaluation protocol, and ablation tables.

## V. Results and Analysis

In this section, we present the results of our experiments. Overall, our findings show that providing hierarchical structural context allows general-purpose LLMs to produce substantially more reliable decompilations than text-only prompts or specialized fine-tuned models.

### A. RQ1: Efficacy of Structural Context on `x86_64`

We first study RQ1 on the **`x86_64` architecture** using HumanEval-Decompile (Table I).

*a) Text-only prompting is brittle under optimization:* The structurally blind, text-only baselines in Group A exhibit poor and inconsistent behavior, especially under higher optimization levels. GPT-4.1 Mini reaches a functional correctness of 74.3% on unoptimized (`-O0`) binaries, but drops to 47.2% at `-O3`. Gemini-2.0-Flash shows the same pattern, falling from 53.0% functional correctness at `-O0` to 26.2% at `-O3`. These results indicate that, without explicit guidance on control flow, models latch onto superficial patterns in the decompiler output that are easily disrupted by compiler transformations.

The average functional correctness of GPT-4.1 Mini on `x86_64` (58.0%) also appears unusually high. We hypothesize that data contamination in the pre-training corpus is a plausible explanation. Since `x86_64` is the most common architecture and HumanEval-style benchmarks are widely used, it is likely that some of these programs or close variants appear in the model's training data. The cross-architecture results in Table III are consistent with this hypothesis. The same GPT-4.1 Mini baseline averages around 46% to 52% functional correctness on non-`x86_64` architectures, with some settings dropping to about 39%. This spread suggests that the high numbers on `x86_64` are not due to robust reasoning alone.

*b) Structural context dramatically improves syntactic correctness:* Adding HELIOS structural context (Group C) substantially improves syntactic correctness. For GPT-4.1 Mini, average object-file compilability increases from 71.4% to 89.6% (+18.2 points). For Gemini-2.0-Flash, the effect is even larger, from 45.0% to 85.2% (+40.2 points). These gains are stable across optimization levels; for example, GPT-4.1 Mini with HELIOS maintains 88.6% compilability at `-O3`. When we enable the compiler feedback loop, average compilability reaches 94.9% for Gemini-2.0-Flash and 96.5% for GPT-4.1 Mini, bringing syntactic correctness close to saturation.

*c) Structural context helps general-purpose LLMs surpass specialized models:* Group B compares HELIOS against fine-tuned decompilers. Nova performs well on edit similarity but almost fails on functional correctness, with averages of 2.2% and 3.2% for its 1.3B and 6.7B variants respectively. LLM4Decompile fares better, with its 6.7B model reaching 63.2% average compilability and 36.3% functional correctness. In contrast, Gemini-2.0-Flash plus HELIOS and feedback achieves 94.9% average compilability and 53.2% functional correctness, while GPT-4.1 Mini with HELIOS and feedback reaches 96.5% compilability and 55.9% functional correctness.

Fine-tuned models achieve the highest edit similarity scores, such as LLM4Decompile (6.7B) with 45.8% average similarity. This pattern suggests that training on text pairs encourages imitation of the original source code's surface form. HELIOS, by contrast, prioritizes control flow and semantics, which yields lower textual similarity but higher functional correctness.

*d) Validation on MBPP-Decompile:* To validate these findings on a different benchmark, we evaluate on MBPP-Decompile (Austin et al. 2021) compiled to `x86_64` (Table II). The trends mirror HumanEval-Decompile. For Gemini-2.0-Flash, enabling HELIOS and feedback increases average compilability from 45.87% to 95.96% and functional correctness from 41.36% to 58.42%. The fine-tuned LLM4Decompile models again lag behind, with average functional correctness between 26.18% and 34.52%, even though their edit similarity is competitive or higher. This agreement across two independent benchmarks strengthens the case that structural context is key for reliable decompilation.

In summary, structural prompting on `x86_64` turns general-purpose LLMs into strong decompilers. It significantly improves compilability, matches functional correctness, and enables general models to outperform specialized, fine-tuned systems.

### B. RQ2: Cross-Architecture Generalization

RQ2 asks whether HELIOS generalizes across architectures without any fine-tuning. We study this on HumanEval-Decompile in Table III and on MBPP-Decompile in Table IV.

*a) Structurally blind prompts do not transfer well:* The text-only baselines show large variation across architectures. For example, Gemini-2.0-Flash on HumanEval-Decompile has functional correctness values around 35% on `x86_32`, but averages only about 22% on `mips_64`. GPT-4.1 Mini exhibits a range of about 20 percentage points in functional correctness across `x86_32`, `arm_32`, and `mips_32`, with some optimization settings dropping below 40%. This volatility indicates that models prompted only with decompiler output tend to learn architecture-specific idioms rather than architecture-independent program logic.

*b) HELIOS narrows cross-architecture gaps:* With HELIOS, the same models become much more stable across architectures. On HumanEval-Decompile, GPT-4.1 Mini with HELIOS and feedback achieves functional correctness that is clustered in a relatively narrow band across `x86_32`,

TABLE I

PERFORMANCE COMPARISON ON **x86_64 ARCHITECTURE** ACROSS OPTIMIZATION LEVELS O0, O1, AND O3 ON THE HUMANEVAL-DECOMPILE DATASET. OUR HELIOS FRAMEWORK IS BENCHMARKED AGAINST BASELINES ACROSS FOUR KEY METRICS.

| | Obj. Compilability (%) | | | | Exec. Linkability (%) | | | | Func. Correctness (%) | | | | Edit Similarity | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Model** | **O0** | **O1** | **O3** | **AVG** | **O0** | **O1** | **O3** | **AVG** | **O0** | **O1** | **O3** | **AVG** | **O0** | **O1** | **O3** | **AVG** |
| **A. Text-Only Baselines (Structurally-Blind)** | | | | | | | | | | | | | | | | |
| Gemini-2.0-Flash | 68.6 | 40.1 | 26.2 | 45.0 | 60.1 | 41.4 | 30.7 | 44.1 | 53.0 | 35.2 | 26.2 | 38.1 | 30.5 | 23.8 | 20.3 | 24.9 |
| GPT-4.1 Mini | 85.8 | 71.0 | 57.3 | 71.4 | 84.8 | 64.5 | 52.1 | 67.1 | 74.3 | 52.4 | 47.2 | 58.0 | 31.6 | 24.1 | 18.0 | 24.6 |
| **B. Specialized Baselines (Fine-Tuned)** | | | | | | | | | | | | | | | | |
| LLM4Decompile (1.3B) | 47.6 | 48.8 | 46.3 | 47.6 | 47.6 | 46.7 | 43.3 | 45.9 | 34.5 | 25.6 | 19.5 | 26.5 | 46.1 | 39.8 | 38.4 | 41.4 |
| LLM4Decompile (6.7B) | 73.2 | 59.8 | 56.7 | 63.2 | 69.2 | 55.2 | 50.6 | 58.3 | 54.0 | 30.2 | 24.7 | 36.3 | 53.2 | 43.7 | 40.5 | 45.8 |
| Nova (1.3B) | 30.8 | 39.0 | 38.7 | 36.2 | 8.8 | 20.4 | 16.5 | 15.2 | 1.2 | 3.1 | 2.4 | 2.2 | 22.4 | 25.3 | 24.5 | 24.1 |
| Nova (6.7B) | 66.2 | 69.5 | 65.9 | 67.2 | 41.5 | 36.6 | 34.5 | 37.5 | 4.0 | 3.1 | 2.4 | 3.2 | 31.2 | 31.4 | 29.4 | 30.7 |
| **C. Our Method (HELIOS w/ Structural Context)** | | | | | | | | | | | | | | | | |
| Gemini-2.0-Flash + HELIOS | 97.6 | 79.8 | 78.3 | 85.2 | 71.3 | 54.4 | 52.8 | 59.5 | 58.1 | 46.9 | 42.7 | 49.2 | 36.0 | 30.1 | 23.0 | 29.7 |
| GPT-4.1 Mini + HELIOS | 96.6 | 83.6 | 88.6 | 89.6 | 76.4 | 65.8 | 59.6 | 67.3 | 64.2 | 45.6 | 41.1 | 50.3 | 36.8 | 27.8 | 23.3 | 29.3 |
| Gemini-2.0-Flash + w/ Fback | 100.0 | 92.5 | 92.2 | 94.9 | 84.5 | 68.4 | 62.1 | 71.7 | 66.6 | 47.9 | 45.0 | 53.2 | 31.6 | 24.1 | 18.0 | 24.6 |
| GPT-4.1 Mini + w/ Feedback | 99.3 | 93.1 | 97.1 | 96.5 | 95.3 | 82.4 | 69.3 | 82.3 | 69.6 | 52.1 | 46.0 | 55.9 | 36.8 | 27.8 | 23.3 | 29.3 |

TABLE II

DETAILED PERFORMANCE ON THE `MBPP` DATASET FOR THE X86_64 ARCHITECTURE ACROSS COMPILER OPTIMIZATION LEVELS (`-O0`, `-O1`, `-O3`). HELIOS CONSISTENTLY OUTPERFORMS STRUCTURALLY-BLIND BASELINES AND LLM4DECOMPILE VARIANTS ACROSS ALL METRICS. FEEDBACK-BASED REFINEMENT NOTABLY BOOSTS CROSS-STAGE CONSISTENCY.

| | Obj. Compilability (%) | | | | Exec. Linkability (%) | | | | Func. Correctness (%) | | | | Edit Similarity | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Model** | **O0** | **O1** | **O3** | **AVG** | **O0** | **O1** | **O3** | **AVG** | **O0** | **O1** | **O3** | **AVG** | **O0** | **O1** | **O3** | **AVG** |
| *Gemini-2.0-Flash* | | | | | | | | | | | | | | | | |
| Baseline | 62.65 | 46.28 | 35.46 | 45.87 | 62.65 | 46.28 | 35.46 | 45.87 | 57.16 | 41.11 | 31.65 | 41.36 | 34.29 | 28.19 | 21.84 | 27.46 |
| HELIOS | 81.25 | 62.76 | 57.14 | 64.90 | 81.25 | 62.76 | 57.14 | 64.89 | 59.73 | 44.97 | 38.39 | 46.35 | 37.49 | 27.35 | 19.86 | 27.14 |
| HELIOS w/ Feedback | 99.00 | 95.48 | 93.99 | 95.96 | 84.53 | 72.67 | 63.08 | 71.39 | 72.42 | 58.82 | 49.37 | 58.42 | 38.12 | 29.86 | 20.86 | 28.56 |
| LLM4Decompile (1.3B) | 46.46 | 45.53 | 42.51 | 44.60 | 32.19 | 26.13 | 22.48 | 26.18 | 32.19 | 26.13 | 22.48 | 26.18 | 47.11 | 41.81 | 36.04 | 41.11 |
| LLM4Decompile (6.7B) | 53.70 | 50.77 | 46.46 | 50.02 | 42.71 | 34.55 | 29.11 | 34.52 | 42.71 | 34.55 | 29.11 | 34.52 | 52.40 | 42.88 | 36.49 | 43.03 |

`arm_32`, `aarch64`, and `mips_32`, while maintaining high compilability everywhere. For example, object-file compilability remains above 94% on all six architectures, compared to baselines that frequently fall into the 40% to 70% range. This pattern is consistent with the intended effect of HELIOS: the model reasons over control-flow graphs and critical instruction rules rather than over the quirks of a specific instruction set.

The MBPP-Decompile cross-architecture results in Table IV tell the same story. For `-O0`, Gemini-2.0-Flash with HELIOS already improves compilability on all six architectures, and the feedback loop raises both compilability and linkage to above 93% and 78% respectively, while functional correctness improves across the board. This behavior is achieved without any architecture specific fine-tuning.

Taken together, these findings provide strong evidence that structural prompting enables a single general-purpose model to decompile binaries across multiple architectures with high, relatively uniform quality. Unlike fine-tuned models that must be retrained for each hardware target, HELIOS offers a practical, fine-tuning-free path toward cross-architecture decompilation.

### C. RQ3: Impact of Compiler Feedback

RQ3 investigates the extent to which the iterative, tool-augmented feedback loop contributes beyond static structural prompting.

*a) Feedback pushes compilability toward perfection:* On `x86_64` HumanEval-Decompile (Table I), compiler feedback raises average compilability for Gemini-2.0-Flash from 85.2% with HELIOS to 94.9%, and for GPT-4.1 Mini from 89.6% to 96.5%. A similar effect appears on MBPP-Decompile (Table II), where Gemini-2.0-Flash jumps from 64.90% compilability with HELIOS to 95.96% with feedback. Across architectures in Table III, feedback-enabled configurations consistently reach near-perfect compilability and linkage, often above 95% compilability and above 70% linkage.

*b) Feedback also improves functional correctness:* While the feedback loop is primarily designed to repair syntactic issues, it also yields meaningful gains in functional correctness. On `x86_64` HumanEval-Decompile, Gemini-2.0-Flash plus HELIOS improves functional correctness from 38.1% (text-only) to 49.2%, and the feedback loop further raises it to 53.2%. For GPT-4.1 Mini, HELIOS alone yields 50.3% functional correctness, and feedback increases this to 55.9%. On MBPP-Decompile, the same pattern appears: Gemini-2.0-Flash with HELIOS increases functional correctness from 41.36% to 46.35%, and feedback raises it to 58.42%.

These improvements suggest a strong coupling between syntactic and semantic errors. Fixing compiler errors, such

TABLE III

COMPREHENSIVE PERFORMANCE RESULTS ACROSS MULTIPLE ARCHITECTURES FOR OBJ. COMPILABILITY (%), EXEC. LINKABILITY (%), FUNC. CORRECTNESS (%), AND EDIT SIMILARITY ACROSS ALL OPTIMIZATION LEVELS (-O0 TO -O3) ON HUMANEVAL-DECOMPILE.

| | Obj. Compilability (%) | | | | | Exec. Linkability (%) | | | | | Func. Correctness (%) | | | | | Edit Similarity | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Model** | **O0** | **O1** | **O2** | **O3** | **AVG** | **O0** | **O1** | **O2** | **O3** | **AVG** | **O0** | **O1** | **O2** | **O3** | **AVG** | **O0** | **O1** | **O2** | **O3** | **AVG** |
| **ARM32** | | | | | | | | | | | | | | | | | | | | |
| *Gemini-2.0-Flash* | | | | | | | | | | | | | | | | | | | | |
| Baseline | 60.88 | 46.71 | 37.25 | 37.91 | 45.69 | 61.22 | 43.09 | 37.25 | 36.93 | 44.62 | 50.00 | 35.20 | 28.43 | 24.18 | 34.45 | 28.40 | 23.29 | 21.99 | 19.63 | 23.33 |
| HELIOS | 98.64 | 85.86 | 84.64 | 83.01 | 88.04 | 76.53 | 55.59 | 54.90 | 51.96 | 59.75 | 58.50 | 36.51 | 37.58 | 37.58 | 40.91 | 29.63 | 21.30 | 19.92 | 18.11 | 22.24 |
| HELIOS w/ Feedback | 99.66 | 96.38 | 93.46 | 92.48 | 95.50 | 88.78 | 69.08 | 68.30 | 66.01 | 73.04 | 64.97 | 37.50 | 39.22 | 31.37 | 43.26 | 30.30 | 23.64 | 21.66 | 19.59 | 23.80 |
| *GPT-4.1 Mini* | | | | | | | | | | | | | | | | | | | | |
| Baseline | 80.95 | 68.75 | 67.65 | 66.67 | 71.01 | 81.97 | 63.49 | 60.13 | 58.82 | 66.10 | 70.41 | 52.30 | 44.12 | 42.81 | 52.41 | 33.05 | 26.96 | 25.28 | 23.19 | 27.12 |
| HELIOS | 96.94 | 80.59 | 80.39 | 74.51 | 83.11 | 78.57 | 56.91 | 52.94 | 51.63 | 60.01 | 63.27 | 41.45 | 34.97 | 30.72 | 42.60 | 32.00 | 23.30 | 22.02 | 19.38 | 24.18 |
| HELIOS w/ Feedback | 97.62 | 88.49 | 85.95 | 82.68 | 88.69 | 85.71 | 66.12 | 61.76 | 59.48 | 68.27 | 68.71 | 48.03 | 39.87 | 36.28 | 48.22 | 33.12 | 24.81 | 23.20 | 20.18 | 25.33 |
| **mips_32** | | | | | | | | | | | | | | | | | | | | |
| *Gemini-2.0-Flash* | | | | | | | | | | | | | | | | | | | | |
| Baseline | 55.78 | 38.76 | 34.95 | 34.63 | 41.03 | 54.76 | 37.46 | 34.30 | 34.63 | 40.29 | 46.26 | 31.92 | 27.51 | 27.18 | 33.22 | 29.19 | 22.79 | 22.09 | 20.74 | 23.70 |
| HELIOS | 95.24 | 85.34 | 82.85 | 82.20 | 86.41 | 71.43 | 53.75 | 53.40 | 53.72 | 58.08 | 58.84 | 43.00 | 40.13 | 39.81 | 45.44 | 29.90 | 22.06 | 20.47 | 19.72 | 23.04 |
| HELIOS w/ Feedback | 99.32 | 94.14 | 93.53 | 92.56 | 94.89 | 83.33 | 65.80 | 66.02 | 66.34 | 70.37 | 63.61 | 44.95 | 41.10 | 41.42 | 47.77 | 30.98 | 23.96 | 21.85 | 21.11 | 24.48 |
| *GPT-4.1 Mini* | | | | | | | | | | | | | | | | | | | | |
| Baseline | 78.23 | 65.80 | 66.02 | 63.75 | 68.45 | 75.51 | 60.59 | 58.58 | 53.72 | 62.10 | 65.31 | 48.21 | 47.25 | 41.10 | 50.47 | 33.50 | 26.81 | 25.59 | 24.50 | 27.60 |
| HELIOS | 91.84 | 76.87 | 77.35 | 77.35 | 80.85 | 69.05 | 50.16 | 50.16 | 51.13 | 55.13 | 57.48 | 41.37 | 37.86 | 37.86 | 43.65 | 31.58 | 23.23 | 22.23 | 21.45 | 24.62 |
| HELIOS w/ Feedback | 95.24 | 85.02 | 82.52 | 84.14 | 86.73 | 76.53 | 57.98 | 58.25 | 58.58 | 62.84 | 64.29 | 46.58 | 43.04 | 43.37 | 49.32 | 32.74 | 24.25 | 23.23 | 22.45 | 25.67 |
| **X86_32** | | | | | | | | | | | | | | | | | | | | |
| *Gemini-2.0-Flash* | | | | | | | | | | | | | | | | | | | | |
| Baseline | 60.20 | 42.02 | 33.87 | 33.33 | 42.36 | 60.86 | 43.97 | 37.10 | 36.57 | 44.63 | 53.04 | 35.18 | 27.18 | 26.21 | 35.40 | 29.33 | 21.74 | 22.46 | 22.00 | 23.88 |
| HELIOS | 94.74 | 80.13 | 81.61 | 81.23 | 84.43 | 76.64 | 55.37 | 51.94 | 54.37 | 59.58 | 58.11 | 46.91 | 43.04 | 42.72 | 47.69 | 29.92 | 20.77 | 19.64 | 19.45 | 22.45 |
| HELIOS w/ Feedback | 94.41 | 85.34 | 89.68 | 90.61 | 90.01 | 85.53 | 66.78 | 62.58 | 64.08 | 69.74 | 66.55 | 47.88 | 43.69 | 44.98 | 50.78 | 33.15 | 24.00 | 24.36 | 23.81 | 26.33 |
| *GPT-4.1 Mini* | | | | | | | | | | | | | | | | | | | | |
| Baseline | 87.50 | 72.31 | 60.32 | 58.58 | 69.68 | 84.21 | 66.12 | 58.06 | 56.63 | 66.26 | 74.32 | 52.44 | 46.60 | 47.25 | 55.15 | 33.64 | 25.41 | 24.41 | 23.95 | 26.85 |
| HELIOS | 92.43 | 79.15 | 84.84 | 85.11 | 85.38 | 80.59 | 59.61 | 56.13 | 58.25 | 63.65 | 64.19 | 45.60 | 43.37 | 41.10 | 48.56 | 32.58 | 22.96 | 23.55 | 23.00 | 25.52 |
| HELIOS w/ Feedback | 99.67 | 88.27 | 93.23 | 93.20 | 93.59 | 86.51 | 64.50 | 60.00 | 60.52 | 67.88 | 69.60 | 52.12 | 46.93 | 45.96 | 53.65 | 30.71 | 22.16 | 20.03 | 19.83 | 23.18 |
| **AARCH64** | | | | | | | | | | | | | | | | | | | | |
| *Gemini-2.0-Flash* | | | | | | | | | | | | | | | | | | | | |
| Baseline | 69.05 | 49.02 | 42.67 | 38.44 | 49.80 | 53.74 | 36.60 | 36.48 | 33.22 | 40.01 | 50.00 | 34.97 | 33.88 | 29.97 | 37.20 | 29.40 | 24.17 | 22.40 | 19.38 | 23.84 |
| HELIOS | 96.94 | 90.52 | 85.67 | 82.74 | 88.97 | 69.39 | 57.52 | 54.07 | 57.00 | 59.50 | 60.54 | 45.75 | 43.32 | 41.37 | 47.75 | 31.73 | 22.86 | 20.92 | 18.12 | 23.41 |
| HELIOS w/ Feedback | 99.66 | 98.04 | 94.79 | 91.21 | 95.93 | 82.99 | 71.57 | 68.40 | 65.15 | 72.03 | 69.73 | 51.31 | 49.19 | 43.32 | 53.39 | 32.87 | 24.23 | 22.21 | 18.73 | 24.51 |
| *GPT-4.1 Mini* | | | | | | | | | | | | | | | | | | | | |
| Baseline | 85.03 | 77.78 | 70.68 | 71.66 | 76.29 | 72.11 | 60.13 | 55.70 | 53.42 | 60.34 | 65.31 | 51.31 | 48.53 | 42.35 | 51.87 | 35.00 | 27.09 | 25.16 | 22.27 | 27.38 |
| HELIOS | 96.26 | 86.60 | 83.06 | 78.83 | 86.19 | 75.17 | 59.80 | 59.93 | 56.35 | 62.81 | 64.97 | 47.06 | 43.97 | 36.16 | 48.04 | 34.82 | 23.96 | 22.67 | 19.41 | 25.22 |
| HELIOS w/ Feedback | 98.64 | 91.50 | 87.95 | 84.69 | 90.70 | 84.35 | 68.95 | 67.75 | 64.50 | 71.39 | 71.09 | 52.61 | 51.79 | 41.37 | 54.22 | 35.43 | 25.30 | 23.61 | 20.11 | 26.11 |
| **MIPS_64** | | | | | | | | | | | | | | | | | | | | |
| *Gemini-2.0-Flash* | | | | | | | | | | | | | | | | | | | | |
| Baseline | 56.12 | 28.99 | 26.21 | 25.89 | 34.30 | 51.70 | 28.01 | 27.51 | 27.18 | 33.60 | 31.97 | 18.89 | 19.42 | 18.45 | 22.18 | 29.42 | 23.28 | 21.82 | 20.51 | 23.76 |
| HELIOS | 96.60 | 87.95 | 78.96 | 77.67 | 85.30 | 72.11 | 57.65 | 55.02 | 52.75 | 59.38 | 47.96 | 40.39 | 36.25 | 36.25 | 40.21 | 31.14 | 22.27 | 20.48 | 19.69 | 23.40 |
| HELIOS w/ Feedback | 94.90 | 85.99 | 85.76 | 84.79 | 87.86 | 80.61 | 62.54 | 59.87 | 62.14 | 66.29 | 45.92 | 40.39 | 37.54 | 38.51 | 40.59 | 33.10 | 24.06 | 22.93 | 22.20 | 25.57 |
| *GPT-4.1 Mini* | | | | | | | | | | | | | | | | | | | | |
| Baseline | 80.95 | 69.38 | 65.05 | 65.37 | 70.18 | 75.85 | 61.56 | 57.28 | 58.25 | 63.235 | 54.42 | 44.63 | 45.31 | 39.16 | 45.88 | 34.53 | 26.54 | 25.14 | 24.74 | 27.74 |
| HELIOS | 92.18 | 79.8 | 79.61 | 78.64 | 82.56 | 72.11 | 56.03 | 52.1 | 54.05 | 58.57 | 43.54 | 37.13 | 36.25 | 37.22 | 38.53 | 32.16 | 23.39 | 22.08 | 21.58 | 24.80 |
| HELIOS w/ Feedback | 99.66 | 96.42 | 93.85 | 90.29 | 95.056 | 79.25 | 68.73 | 69.58 | 66.02 | 70.90 | 52.04 | 40.39 | 43.04 | 41.42 | 44.22 | 31.92 | 24.19 | 21.7 | 21.03 | 24.71 |

as missing declarations or mismatched types, often forces the model to reconsider the surrounding logic, thereby correcting latent semantic bugs. Although the feedback loop introduces additional latency and tool calls, the data shows that it significantly improves both syntactic and functional quality.

### D. RQ4: Ablation Study of Prompt Components

RQ4 isolates the contribution of each component in the HELIOS hierarchy. Table V reports an ablation on HumanEval-Decompile for `x86_64`, varying which parts of the structured prompt are present: the control-flow graph (CFG), critical instruction rules, high-level function information, and compiler feedback (CF).

*a) Raw structure alone is not enough:* Starting from the text-only base prompt, Gemini-2.0-Flash achieves an average compilability of 40.43%. Adding only CFG information increases this to 51.91%, a modest gain that shows structure helps but is not sufficient by itself. GPT-4.1 Mini exhibits a similar pattern, and in some cases raw CFG alone can even undermine stability under higher optimization levels.

*b) Reasoning rules drive the largest gains:* The largest single jump occurs when we introduce the critical instruction

TABLE IV

CROSS-ARCHITECTURE GENERALIZATION PERFORMANCE ACROSS SIX DISTINCT ARCHITECTURES AT THE −O0 OPTIMIZATION LEVEL ON THE MBPP-DECOMPILE DATASET. THE TABLE SHOWS SYNTACTIC CORRECTNESS (COMP., LINK.) AND FUNCTIONAL CORRECTNESS (TEST.)

| | x86_32 | | | x86_64 | | | arm_32 | | | aarch64 | | | mips_32 | | | mips_64 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Model** | **Comp.** | **Link.** | **Test.** | **Comp.** | **Link.** | **Test.** | **Comp.** | **Link.** | **Test.** | **Comp.** | **Link.** | **Test.** | **Comp.** | **Link.** | **Test.** | **Comp.** | **Link.** | **Test.** |
| *Gemini-2.0-Flash* | | | | | | | | | | | | | | | | | | |
| Baseline | 67.2 | 60.9 | 52.7 | 69.0 | 60.2 | 57.2 | 73.6 | 67.2 | 52.0 | 70.8 | 58.2 | 55.9 | 66.3 | 56.9 | 54.2 | 65.1 | 55.3 | 40.9 |
| HELIOS | 95.5 | 77.4 | 53.4 | 94.9 | 78.6 | 59.7 | 96.1 | 76.9 | 54.5 | 95.3 | 72.3 | 57.3 | 93.6 | 69.6 | 52.1 | 94.4 | 69.8 | 43.7 |
| HELIOS W/ FEEDBACK | 97.5 | 83.6 | 63.2 | 99.0 | 84.5 | 72.4 | 98.3 | 85.1 | 69.8 | 97.4 | 81.5 | 72.4 | 95.3 | 78.3 | 67.6 | 96.2 | 77.8 | 53.1 |

TABLE V

ABLATION STUDY ON HUMANEVAL−DECOMPILE FOR X86_64. "CF" IS COMPILER FEEDBACK, "FUNC." IS HIGH-LEVEL FUNCTION CALL INFO, "RULES" ARE CRITICAL INSTRUCTION RULES, AND "CFG" IS CONTROL-FLOW GRAPH. WE REPORT RE-COMPILABILITY, RE-EXECUTABILITY, AND EDIT SIMILARITY OVER OPTIMIZATION LEVELS −O0 TO −O3.

| **Prompt Configuration** | **Obj. Compilability (%)** | | | | | **Exec. Linkability (%)** | | | | | **Edit Similarity** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **O0** | **O1** | **O2** | **O3** | **AVG** | **O0** | **O1** | **O2** | **O3** | **AVG** | **O0** | **O1** | **O2** | **O3** | **AVG** |
| *Gemini-2.0-Flash* | | | | | | | | | | | | | | | |
| Base | 68.58 | 40.07 | 26.86 | 26.21 | 40.43 | 60.14 | 41.37 | 31.07 | 30.74 | 40.83 | 30.53 | 23.75 | 22.85 | 20.32 | 24.36 |
| +CFG | 73.99 | 48.86 | 43.69 | 41.10 | 51.91 | 60.14 | 43.00 | 39.16 | 36.57 | 44.72 | 28.12 | 21.23 | 21.39 | 19.10 | 22.46 |
| +Rules | 94.93 | 81.76 | 86.41 | 80.26 | 85.84 | 75.68 | 54.72 | 53.72 | 51.46 | 58.90 | 31.91 | 24.29 | 22.89 | 19.65 | 24.69 |
| +Func. | 97.64 | 79.80 | 84.14 | 78.32 | 84.98 | 71.28 | 54.40 | 54.37 | 52.75 | 58.20 | 31.63 | 24.13 | 21.18 | 18.04 | 23.75 |
| +CF | 100.00 | 92.51 | 93.20 | 92.23 | 94.49 | 84.46 | 68.40 | 61.81 | 62.14 | 69.20 | 31.80 | 25.10 | 23.50 | 20.60 | 25.30 |
| *GPT-4.1 Mini* | | | | | | | | | | | | | | | |
| Base | 85.81 | 71.01 | 57.61 | 57.28 | 67.93 | 84.80 | 64.50 | 55.02 | 52.10 | 64.11 | 36.02 | 30.06 | 25.97 | 23.02 | 28.77 |
| +CFG | 66.78 | 44.77 | 38.89 | 39.81 | 47.56 | 59.32 | 45.10 | 41.83 | 43.04 | 47.32 | 30.07 | 23.08 | 23.32 | 20.59 | 24.27 |
| +Rules | 95.61 | 77.20 | 88.96 | 84.47 | 86.56 | 74.66 | 63.52 | 61.69 | 58.90 | 64.69 | 35.29 | 26.17 | 25.81 | 22.79 | 27.52 |
| +Func. | 96.58 | 83.55 | 92.51 | 88.60 | 90.31 | 76.37 | 65.79 | 61.24 | 59.61 | 65.75 | 36.83 | 27.83 | 26.27 | 23.26 | 28.55 |
| +CF | 99.32 | 93.14 | 97.40 | 97.09 | 96.74 | 95.25 | 82.35 | 72.08 | 69.26 | 79.74 | 39.20 | 29.30 | 27.50 | 24.50 | 30.10 |

rules that explain how to use the CFG. For Gemini-2.0-Flash, average compilability rises from 51.91% to 85.84%, and for GPT-4.1 Mini from 47.56% to 86.56%. This step also improves executability and edit similarity. These results highlight that exposing structure alone is insufficient; the prompt must also teach the model how to reason over that structure.

*c) Function context and feedback close the remaining gap:* Adding high-level function context provides a smaller but consistent improvement. For GPT-4.1 Mini, the combination of CFG, rules, and function information reaches 90.31% average compilability and improves both linkage and edit similarity. Finally, enabling compiler feedback yields the strongest configuration, with average compilability of 94.49% for Gemini-2.0-Flash and 96.74% for GPT-4.1 Mini, and the highest executability across all prompt variants.

Overall, the ablation confirms that each layer of HELIOS plays a distinct role. Explicit, well-formatted structure is helpful, explicit reasoning rules are essential, and the feedback loop further refines the output to achieve near-perfect recompilation while also improving functional behavior.

## VI. DISCUSSION

### A. Implications

The main lesson from our study is that large language models behave differently when they are treated as structure-aware reasoning engines rather than as text-to-text translators. Feeding an LLM only decompiler output leaves it exposed to small syntactic shifts caused by compiler optimizations, which our experiments show clearly. In contrast, when the model is given a compact, hierarchical view of the control flow and some guidance on how to use it, it produces code that is much more stable across optimization levels and architectures.

This has two practical consequences. First, for binary analysis tools, this suggests wrapping existing decompilers with LLMs without retraining, while retaining much of the robustness associated with specialized models. Second, it shows that a relatively small amount of structural context can substitute for large, architecture-specific training corpora. General-purpose models such as GPT-4.1 Mini and Gemini-2.0-Flash, when paired with HELIOS, match or outperform fine-tuned decompilers on our functional metrics, even though they have never been trained directly for this task.

The results also highlight a tension between surface similarity and actual behavior. Fine-tuned baselines obtain higher edit similarity scores, but their functional correctness is often much lower. HELIOS, by design, optimizes for control flow and semantic fidelity rather than textual imitation. For reverse engineering workflows, this trade-off is usually acceptable. Analysts care more about what the recovered program does than about reproducing the original formatting or identifier choices. A stronger decompiler also reduces the effort needed to understand opaque binaries, which can simplify both benign reverse engineering and the analysis of vulnerable or malicious code.

A further implication concerns the cost profile. Our exper-

iments use at most one round of compiler feedback per function, so each decompilation requires no more than two model calls and two compiler invocations. This keeps the approach practical for batch use and compares favorably to methods that rely on fine-tuning large models for each architecture. In settings where human analyst time is the dominant cost, the additional model and compiler calls are likely to be an acceptable trade-off for higher quality output.

More broadly, the same pattern should apply beyond decompilation. Any task where the underlying object has rich structure, such as a control flow graph, a protocol state machine, or a complex data schema, can likely benefit from a similar structural grounding approach: make the structure explicit, explain how to reason over it, and use external tools for feedback rather than baking everything into model weights.

### B. Limitations and Threats to Validity

Our evaluation has several limitations that are important to note.

**Benchmark construction.** We rely on HumanEval and MBPP programs compiled into C binaries and then decompiled with Ghidra. These problems are short and self-contained. They are a good fit for controlled measurement of compilability and test passing, but they do not capture the full complexity of large, multi-module software with extensive state, libraries, and build systems. As a result, our numbers should be interpreted as lower-level evidence that structural prompting helps with decompilation, not as a direct prediction of performance on arbitrary real-world codebases.

**Potential data contamination.** Like most work that evaluates commercial LLMs on public benchmarks, we cannot rule out that some HumanEval or MBPP tasks, or close variants, appear in the pre-training data. We see signs consistent with this. GPT-4.1 Mini exhibits noticeably higher functional correctness on `x86_64` than on other architectures, even though the underlying programs are identical. Our cross-architecture experiments partially mitigate this concern by showing that HELIOS improves performance in settings where simple memorization is less likely to help, but they do not eliminate it entirely.

**Dependence on Ghidra and the foundation model.** HELIOS assumes that the static analysis backend produces a reasonable control flow graph and pseudo-C. If the decompiler misidentifies basic blocks or control edges, the structural context we build will also be misleading. Likewise, our results are tied to the behavior of GPT-4.1 Mini and Gemini-2.0-Flash at the time of experimentation. Different model families, sizes, or decoding settings may shift the absolute numbers, even if the relative trends hold.

**Baselines.** We compare against two families of LLM-based decompilers (LLM4Decompile and Nova) and a text-only prompting baseline. Classical non LLM decompilers are not treated as competitors, since HELIOS is designed to sit on top of such tools and reuse their analyses. Systems such as DeGPT focus on readability rather than recompilability and functional correctness, so a direct comparison would require additional metrics and user studies that are outside the scope of this work. For Nova, we use a standardized preprocessing pipeline and greedy decoding for consistency across models, which can yield lower scores than those reported in the original paper.

These limitations mean that our results should be viewed as evidence that structural prompting can make LLM assisted decompilation more reliable under controlled conditions, rather than as a final answer on how such systems will behave in all reverse engineering settings.

### C. Future Work

Several extensions follow naturally from this work. A first step is to move beyond synthetic benchmarks and evaluate HELIOS on larger, real-world binaries drawn from open source projects, including code that uses complex libraries, system calls, and build chains. This would also make it possible to compare against mature non-LLM decompilers on tasks such as recovering higher-level types, data structures, or calling conventions. On the modeling side, it would be useful to incorporate more explicit data flow information, especially for code with heavy pointer arithmetic, aliasing, or complex memory layouts. Finally, from a tooling perspective, HELIOS can be integrated more tightly into reverse engineering platforms through interfaces such as the Model Context Protocol.

## VII. CONCLUSION

In this work, we addressed the structural blindness that limits large language models in graph-rich domains such as reverse engineering. We introduced HELIOS, a framework that provides a hierarchical textual representation of a binary's control flow and related context, and uses this structure, together with a small set of rules and an optional compiler feedback loop, to guide general-purpose LLMs during decompilation.

Our experiments on HumanEval-Decompile and MBPP-Decompile, across six architectures and multiple optimization levels, show that this context-aware approach substantially improves compilability and functional correctness compared to text-only prompting and to specialized fine-tuned decompilers, while requiring no task-specific training. More broadly, the results support a simple paradigm for applying LLMs to structure-dependent problems: make the underlying structure explicit, explain how it should be used, and rely on external tools for grounded feedback instead of relying solely on model weights.

## ACKNOWLEDGMENTS

This work was supported by Google.org and the Google Cloud Research Credits program through the Gemini Academic Program. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of Google.org or Google.

## REFERENCES


[1] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse Engineers' processes," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1875–1892. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-observational

[2] X. Xu, Z. Zhang, S. Feng, Y. Ye, Z. Su, N. Jiang, S. Cheng, L. Tan, and X. Zhang, "Lmpa: Improving decompilation by synergy of large language model and program analysis," *arXiv preprint arXiv:2306.02546*, 2023.

[3] J. L. Hennessy and D. A. Patterson, *A new golden age for computer architecture*. ACM New York, NY, USA, 2019, vol. 62.

[4] ISACA, "State of cybersecurity 2021: Global update on workforce efforts, resources and cyberoperations," ISACA, Tech. Rep., 2021.

[5] DARPA, "Hardening development toolchains against emergent execution engines (harden)," 2023, accessed: 2023-11-06. [Online]. Available: https://www.darpa.mil/program/hardening-development-toolchains-against-emergent-execution-engines

[6] ——, "Verified security and performance enhancement of large legacy software (v-spell)," 2023, accessed: 2023-11-06. [Online]. Available: https://www.darpa.mil/program/verified-security-and-performance-enhancement-of-large-legacy-software

[7] Y. G. Achamyeleh, T. Zhang, J. H. Kim, G. Garcia, S.-Y. Yu, A. Kocheturov, and M. A. A. Faruque, "Agnomin–architecture agnostic multi-label function name prediction," *arXiv preprint arXiv:2509.25514*, 2025.

[8] NSA, "Ghidra - software reverse engineering framework." 2019. [Online]. Available: https://www.nsa.gov/resources/everyone/ghidra/

[9] S. Hex-Rays, "Ida disassembler," 2017.

[10] W. K. Wong, H. Wang, Z. Li, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "Refining decompiled c code with large language models," 2023. [Online]. Available: https://arxiv.org/abs/2310.06530

[11] Z. Liu and S. Wang, "How far we have come: testing decompilation correctness of c decompilers," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 475–487.

[12] Z. L. Basque, A. P. Bajaj, W. Gibbs, J. O'Kain, D. Miao, T. Bao, A. Doupé, Y. Shoshitaishvili, and R. Wang, "Ahoy {SAILR}! there is no need to {DREAM} of c: A {Compiler-Aware} structuring algorithm for binary decompilation," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 361–378.

[13] H. Tan, Q. Luo, J. Li, and Y. Zhang, "LLM4Decompile: Decompiling Binary Code with Large Language Models," Jun. 2024, arXiv:2403.05286 [cs]. [Online]. Available: http://arxiv.org/abs/2403.05286

[14] N. Jiang, C. Wang, K. Liu, X. Xu, L. Tan, and X. Zhang, "Nova: Generative language models for assembly code with hierarchical attention and contrastive learning," *arXiv preprint arXiv:2311.13721*, 2023.

[15] P. Hu, R. Liang, and K. Chen, "DeGPT: Optimizing Decompiler Output with LLM," in *Proceedings 2024 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2024. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2024-401-paper.pdf

[16] Y. Feng, D. Teng, Y. Xu, H. Mu, X. Xu, L. Qin, Q. Zhu, and W. Che, "Self-Constructed Context Decompilation with Fined-grained Alignment Enhancement," Oct. 2024, arXiv:2406.17233 [cs]. [Online]. Available: http://arxiv.org/abs/2406.17233

[17] Y. Feng, B. Li, X. Shi, Q. Zhu, and W. Che, "ReF Decompile: Relabeling and Function Call Enhanced Decompile," Feb. 2025, arXiv:2502.12221 [cs]. [Online]. Available: http://arxiv.org/abs/2502.12221

[18] A. Mantovani, S. Aonzo, Y. Fratantonio, and D. Balzarotti, "{RE-Mind}: a first look inside the mind of a reverse engineer," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2727–2745.

[19] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychology*, vol. 19, no. 3, pp. 295–341, Jul. 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0010028587900077

[20] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *23rd USENIX Security Symposium (USENIX Security 16)*, 2016.

[21] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

[22] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *arXiv preprint arXiv:2012.08680*, 2020.

[23] S.-Y. Yu, Y. G. Achamyeleh, C. Wang, A. Kocheturov, P. Eisen, and M. A. Al Faruque, "Cfg2vec: Hierarchical graph neural network for cross-architectural software reverse engineering," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2023, pp. 281–291.

[24] J. Zhao, L. Zhuo, Y. Shen, M. Qu, K. Liu, M. Bronstein, Z. Zhu, and J. Tang, "Graphtext: Graph reasoning in text space," 2023. [Online]. Available: https://arxiv.org/abs/2310.01089

[25] B. Fatemi, J. Halcrow, and B. Perozzi, "Talk like a graph: Encoding graphs for large language models," *arXiv preprint arXiv:2310.04560*, 2023.

[26] J. Guo, L. Du, and H. Liu, "Gpt4graph: Can large language models understand graph structured data? an empirical evaluation and benchmarking," *ArXiv*, vol. abs/2305.15066, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:258865990

[27] F. E. Allen, "Control flow analysis," in *Proceedings of a symposium on Compiler optimization*, 1970, pp. 1–19.

[28] B. G. Ryder, "Constructing the call graph of a program," in *IEEE Transactions on Software Engineering*, no. 3. IEEE, 1979, pp. 216–226.

[29] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 480–491.

[30] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 363–376.

[31] E. Eilam, *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

[32] K. Burk, F. Pagani, C. Kruegel, and G. Vigna, "Decomperson: How humans decompile and what we can learn from it," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2765–2782.

[33] J. Armengol-Estapé, J. Woodruff, C. Cummins, and M. F. P. O'Boyle, "Slade: A portable small language model decompiler for optimized assembly," in *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '24. IEEE Press, 2024, p. 67–80. [Online]. Available: https://doi.org/10.1109/CGO57630.2024.10444788

[34] NSA, "Ghidra pcode reference," {https://ghidra.re/ghidra_docs/languages/html/pcoderef.html}, 2020, accessed: 2025-07-01.

[35] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using {Semantics-Preserving} structural analysis and iterative {Control-Flow} structuring," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 353–368.

[36] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[37] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732


## APPENDIX

This section provides supplementary evaluations.

### A. NOVA Evaluation Methodology

To ensure a rigorous and fair comparison, all models in our study were evaluated under an identical, reproducible pipeline. For the NOVA baseline, we preserved the core model architecture from the original work [14] but standardized the preprocessing using the pipeline established by LLM4Decompile [13]. All experiments use greedy decoding (temperature=0) to eliminate sampling variance as a confounding factor.

As shown in Table VI, our evaluation yields lower Pass@1 scores for NOVA than those reported in the original paper. We

attribute this discrepancy to differences in preprocessing and the original paper's use of a sampling-based evaluation. Our standardized methodology, while resulting in lower scores for this specific baseline, ensures that all comparisons within this paper are conducted under identical conditions, providing a fair assessment of relative performance.

TABLE VI
COMPARISON OF PASS@1 RESULTS FOR NOVA MODELS UNDER OUR STANDARDIZED EVALUATION VERSUS THE RESULTS REPORTED IN THE ORIGINAL PAPER [14].

| Model | O0 | O3 | AVG |
|---|---|---|---|
| **This Work (Standardized Preprocessing)** | | | |
| Nova 1.3B | 1.22% | 2.44% | 2.13% |
| Nova 6.7B | 3.96% | 2.44% | 2.97% |
| **Original NOVA Results from [14]** | | | |
| Nova 1.3B | 37.53% | 18.75% | 25.17% |
| Nova 6.7B | 48.78% | 27.23% | 34.36% |

This section presents the comprehensive results for both the HumanEval-Decompile and MBPP datasets, broken down by architecture, optimization level, and evaluation metric.

### *B. MBPP Dataset Results*

To validate our findings on a different benchmark, we conducted experiments on the MBPP dataset. Table II provides a detailed breakdown of performance on `x86_64`, while Table IV summarizes the cross-architecture performance at the `-O0` optimization level. The results are consistent with our primary findings, confirming that HELIOS substantially outperforms both text-only and fine-tuned baselines.

Table VII presents the ablation study for MBPP. The incremental improvements from each prompt component mirror the trends observed on the HumanEval-Decompile dataset, confirming the robustness of our hierarchical prompt design.

TABLE VII
ABLATION STUDY ON THE INCREMENTAL IMPACT OF PROMPT COMPONENTS FOR THE GEMINI 2.0 MODEL ON THE MBPP DATASET (`-O0`). EACH ROW CUMULATIVELY ADDS A NEW LAYER OF CONTEXT, CULMINATING IN THE COMPILER FEEDBACK LOOP WHICH NEARLY ACHIEVES PERFECT SYNTACTIC CORRECTNESS.

| Prompt Configuration | Obj. Compilability (%) |
|---|---|
| (A) Basic (Decompiled Code + Task) | 69.0 |
| (B) + CFG Information | 75.84 |
| (C) + Critical Rules | 95.65 |
| (D) + Function Overview | 95.15 |
| **(E) + Compiler Feedback** (Full HELIOS) | **99.0** |