Does Representation Matter? Evaluating IRs for LLM-based Binary Decompilation

Tomás Pelayo-Benedet
Universidad de Zaragoza
tpelayo@unizar.es

Kevin Borgolte
Ruhr University Bochum
kevin.borgolte@rub.de

Ricardo J. Rodríguez
Universidad de Zaragoza
ricardo@unizar.es

***Abstract*—Binary decompilation remains an open challenge in reverse engineering. While recent approaches have begun to leverage the capabilities of large language models (LLMs), most continue to focus exclusively on disassembly as input, ignoring the intermediate representations (IRs) employed by static binary analysis tools and traditional decompilers.**

In this paper, we present the first systematic evaluation of LLM-based decompilation using hierarchical IRs. In particular, we investigate how different levels of abstraction in IRs affect binary decompilation quality in five commercial LLMs. Our findings show that the choice of IR significantly influences performance: Smaller models benefit markedly from high-level structured IRs, while larger models show stable performance across IR levels. Our evaluation also reveals a significant trade-off between recompilation success and functional correctness. Code decompiled from disassembly tends to recompile more reliably, but it is less often functionally correct. In contrast, code decompiled from high-level IRs more often retains the original functionality, albeit with slightly lower recompilation success rates. Furthermore, we find that cognitive complexity metrics, such as Halstead measures, are strong predictors of decompilation difficulty, while traditional structural metrics, such as cyclomatic complexity, offer limited insight. We also highlight the main lines of research to improve binary decompilation by combining the advantages of static binary analysis techniques with the capabilities of modern LLMs.

I. Introduction

Despite decades of effort, converting binaries into understandable source code remains more art than science. Binary decompilation remains one of the most challenging problems in reverse engineering. It involves not only the interpretation of low-level machine instructions, but also the reconstruction of high-level programming abstractions (e.g., control structures, data types, and relationships between variables), which are often lost during the compilation process [5]. This task is complicated by the semantic gap between the high-level code and its compiled form, where much of the original algorithmic structure and variable semantics are irreversibly discarded.

Traditional decompilers, such as `IDA Pro` [9], `Ghidra` [14], and `Binary Ninja` [18], have made significant progress by leveraging heuristics and pattern matching. However, they often fail for highly optimized or obfuscated binaries, where conventional static analysis techniques reach their limits. Compiler optimizations and obfuscation techniques, like control flow flattening, can drastically alter the program structure, making it extremely difficult (if not impossible) for rule-based decompilers to accurately recover the original semantics.

Large language models (LLMs) provide new opportunities to tackle these limitations. LLMs have exhibited good performance on various code-related tasks [4], prompting researchers to explore their applicability to binary decompilation. For example, Tan et al. introduced `LLM4Decompile` [16], a system that optimizes LLMs specifically for binary-to-source translation. Similarly, Feng et al. [6, 7] and Liu et al. [11] proposed improvements to LLM-based decompilation techniques.

However, all these approaches predominantly rely on pure disassembly as input, ignoring the potential advantages of structured intermediate representations (IRs). Modern decompilers, like `Binary Ninja` [18], use a hierarchy of IRs, from low-level intermediate language (IL) to mid-level IL to high-level IL, providing progressively higher semantic abstractions while preserving the essential semantics of the original binary. These IRs could provide more efficient and structured input for LLM-based binary decompilation, potentially improving accuracy and interpretability. The motivation for using hierarchical IRs lies in their progressive abstraction: They serve as semantic bridges between low-level machine code and high-level constructs, facilitating more effective code reconstruction.

Building on this intuition, we explore whether and how these hierarchical IRs can be effectively leveraged by LLMs. In this paper, we investigate the potential of IRs to bridge the gap between low-level disassembly and source-code understanding in LLM-based decompilation. We present the first preliminary evaluation of hierarchical IRs in this context, systematically examining how different levels of abstraction affect decompilation quality across multiple commercial LLMs, and providing new insights into the relationship between recompilation reliability and functional correctness.

We evaluate five state-of-the-art LLMs from OpenAI and Anthropic and analyze their performance at four IR levels with a unified experimental framework. It covers the automated validation of the functional correctness of decompiled outputs and recompilation success, enabling robust comparative analysis.

Our results reveal several novel insights that inform the design of future LLM-based decompilation strategies. First, we

Workshop on Binary Analysis Research (BAR) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-08-4
https://dx.doi.org/10.14722/bar.2026.23077
www.ndss-symposium.org

observe that model size significantly influences the usefulness of different IRs: Smaller models benefit significantly from higher-level, more abstract IRs, while larger models exhibit consistent performance across IR levels. We also observe a fundamental trade-off: Higher-level IRs improve functional correctness, but at the cost of lower recompilation success. Our findings also show that there is no universally optimal IR abstraction level; rather, the best choice depends on model capacity and the specific goals of the decompilation task.

Our analysis of code complexity further reveals that cognitive complexity metrics, specifically Halstead measures, are significantly more predictive of LLM-based decompilation performance than structural metrics, like cyclomatic complexity. This suggests that LLM-based decompilation introduces new challenges that differ from conventional approaches, which calls for new evaluation criteria and highlights the need to consider how code comprehension should generally be assessed.

Overall, we lay the groundwork for designing more effective LLM-based decompilers. We recommend that future systems incorporate hierarchical IRs and tailor IR selection based on task requirements and computational resources. In parallel, decompiler quality assessments should consider incorporating cognitive complexity metrics to better capture the underlying difficulty of code reconstruction for LLMs.

In this paper, we make the following technical contributions:

- We present the first systematic evaluation of how hierarchical IRs affect LLM-based binary decompilation, examining how different levels of abstraction impact decompilation performance and quality in five commercial LLMs.
- We investigate the relationship between model size and IR effectiveness, showing that smaller models benefit substantially from higher-level abstractions, while larger models maintain robust performance across all representation levels.
- We demonstrate that Halstead cognitive code complexity metrics are superior predictors of LLM-based decompilation difficulty compared to traditional code complexity metrics, such as cyclomatic complexity, suggesting fundamental differences in how LLMs approach code reconstruction.
- We identify significant trade-offs between recompilation success and its functional correctness across different IRs, and we provide first guidance on how to select the most suitable IR for LLM-based decompilation.

II. Experimental Setup

We designed our experimental framework to systematically evaluate the binary decompilation capabilities of LLMs across multiple IRs and varying levels of code complexity. Our primary goals were to ensure reproducibility, statistical rigor, and practical relevance. To do so, we carefully selected a diverse set of LLMs, developed an appropriate evaluation dataset, analyzed multiple levels of IR abstraction, and employed a minimal and consistent prompt design.

Table I: Overview of the evaluated LLMs.

Model	**Release**	**API Tag**
GPT-4.1[†]	Apr 2025	`gpt-4.1-2025-04-14`
GPT-4.1-nano[‡]	Apr 2025	`gpt-4.1-nano-2025-04-14`
GPT-4o-mini[‡]	Jul 2024	`gpt-4o-mini-2024-07-18`
Claude Sonnet 4[†]	May 2025	`claude-sonnet-4-20250514`
Claude Haiku 3.5[‡]	Oct 2024	`claude-3-5-haiku-20241022`

[†] Larger models. [‡] Smaller models counterparts.

a) LLM Selection: We evaluated five LLMs in total, which we selected for their architectural diversity, coverage of different parameter scales, and accessibility via public APIs. Specifically, we included Claude Sonnet 4 and Claude Haiku 3.5 from Anthropic [1], and GPT-4.1, GPT-4.1-nano, and GPT-4o-mini from OpenAI [15], as shown in Table I. These models represent the state-of-the-art of commercial LLM offerings, frequently used in real-world software engineering and code analysis tasks. They also cover a range of parameter scales and capabilities, allowing us to explore how model size influences the effectiveness of different levels of IR abstraction. We conducted our experiments in June 2025.

b) Dataset Selection and Sampling: To support a rigorous and meaningful evaluation of LLM-based decompilation, we considered several established software engineering datasets [2, 16, 17]. Ultimately, we selected ExeBench [2] because it is one of the only datasets that includes unit tests for every function, allowing us to validate the functional correctness of the decompiled results. This validation capability is necessary for our evaluation framework, but it is notably lacking in most publicly available decompilation datasets; a limitation we discuss in more detail in Section IV.

To ensure representative evaluation across different levels of code complexity, we categorized the functions in ExeBench based on their cyclomatic complexity [13], which quantifies the number of independent execution paths through a program. It provides a structured and objective basis for sampling and avoids bias toward overly simple functions. Our goal was to ensure that the evaluation included both simple and complex cases of decompilation scenarios.

We applied stratified sampling in the official ExeBench test split, restricting our analysis to functions with cyclomatic complexity 1 to 11. This is the vast majority of the dataset, as only 42 functions (2.3%) exceed this threshold. The remaining functions with complexity >11 are distributed too sparsely across 18 individual complexity levels (≈2.3 functions each), preventing reliable analysis. For each complexity class, we selected 10% of the available samples or 25 functions, whichever was greater. For classes with fewer than 25 functions, we included all (Table II). Our sampling strikes a balance between coverage and statistical consistency across complexity levels.

The resulting selected dataset contains 306 functions, each of which passes all associated unit tests. This ensures that we start from a functionally correct code base, allowing for a reliable and fair comparison of decompiled results generated

Table II: Functions per cyclomatic complexity level.

Complexity	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
# Functions	98	30	25	25	25	25	25	16	11	12	14

by different LLMs. To promote reproducibility, the complete list of selected functions are publicly available on GitHub.[1]

c) Intermediate Representations: To systematically evaluate the impact of semantic abstraction on LLM-based decompilation, we analyzed three hierarchical IRs in addition to the raw disassembly, which serves as a baseline. We used `Binary Ninja` [18] as our static analyzer, as it supports three distinct IR levels: Low-Level IL (LLIL), Medium-Level IL (MLIL), and High-Level IL (HLIL).

`Binary Ninja` builds this IR hierarchy using traditional static analyses, progressively elevating disassembly to LLIL, from LLIL to MLIL, and from MLIL to HLIL. This structured lifting allows us to evaluate how different levels of abstraction influence decompilation quality while minimizing the confounding effects that arise from using unrelated or non-hierarchical IR. The semantically consistent IRs at all levels also provide a lens through which we can evaluate the complementarity of static analyses and LLMs.

d) Prompt Construction: Our review of previous work revealed a notable absence of systematic prompt design for LLM-based decompilation. Existing approaches tend to rely on simplistic instructions (e.g., "decompile this assembly code") [6, 16] or use vague formatting constraints, without exploring how prompt structure or IR specificity might influence results. To address this shortcoming, we developed a minimal, structured prompt template suitable for use across multiple IR and LLM levels (see Appendix A).

Rather than tailoring prompts for models or IRs, we prioritized consistency and interpretability, designing it based on:

1) *Task Definition and IR Context*: We explicitly state that the goal is to decompile a given IR and briefly describe the abstraction level to help the model contextualize.
2) *IR Input Presentation*: For low-level representations, we include instruction addresses, as they are essential for reconstructing control flow because of jump targets.
3) *Output Constraints*: We instruct the model to generate C code for a function named `func0`, with parameters named `var1`, `var2`, etc., and to include all `#include` directives necessary for successful recompilation and execution.

While recent work suggests that model-specific prompt tuning can yield performance improvements by leveraging model internals [12], we deliberately standardized the prompts across all models to ensure a fair comparison. This allows us to evaluate the intrinsic decompilation capabilities of each model, isolating performance from model-specific optimizations.

e) Evaluation Pipeline: We systematically evaluate LLM-based decompilation using a five-stage evaluation process, from source code compilation to functional correctness testing. It allows for a detailed analysis of decompilation results across representations, models, and complexity levels:

1) *Source Code Compilation:* We begin by compiling all functions in our dataset using GCC version 13.3.0 with optimizations disabled (`-O0`). Disabling compiler optimizations preserves source code structures in the binary and maintains the original complexity characteristics.
2) *Extracting Intermediate Representation:* For each compiled binary, we use `Binary Ninja` to extract four different code representations: disassembly, LLIL, MLIL, and HLIL.
3) *LLM Decompilation:* Next, we ask each LLM to decompile each function in each representation.
4) *Compilation Validation:* We recompile each decompiled function with the same compiler configuration as for the original source. This ensures syntactic validity and that a standard C compiler can process the LLM output.
5) *Functional Testing:* For all successfully recompiled decompilations, we run the original unit tests to assess functional correctness. We only consider decompilations that pass all associated test cases as functionally correct.

f) Statistical Rigor: To ensure the reliability of our results, we perform ten decompilation tests per function, for each model and for each representation. Since identical prompts can produce different results on each run, evaluations that consider just a single can be highly misleading [3].

Repeating the decompilations further allows us to estimate not only the average success rates, but also investigate the variance in each model's performance. This enables an even more accurate comparison across conditions and facilitates statistical analyses, such as confidence interval estimation and significance testing, following established best practices for LLM evaluation [19].

III. Results

We aim to answer three questions: (a) How do different IRs affect LLM-based decompilation? (b) How does source code complexity affect LLM-based decompilation? and (c) How do IR abstraction levels and code complexity interact for LLM-based decompilation?

A. LLM Performance

Figure 1 summarizes the decompilation performance across the five LLMs and four input representations that we analyzed.

a) Impact of Model Size: Model size significantly influences decompilation results. Within each family, larger models consistently outperform smaller models. Claude Sonnet 4 outperforms Haiku 3.5 across all IRs with higher functional correctness and fewer recompilation errors. Similarly, GPT-4.1 outperforms GPT-4o-mini and GPT-4.1-nano, indicating that larger models are more robust at all abstractions, possibly due to a greater ability to reconstruct code.

[1]https//github.com/rub-softsec/does-representation-matter-dataset

b) Recompilation vs. Correctness: We observe a clear trade-off between recompilation success and functional correctness across all IR levels. Disassembly has the lowest recompilation error rates (e.g., Claude Sonnet 4 at 1.5%, GPT-4o-mini at 10.6%), while higher-level IRs, such as HLIL, result in a higher number of recompilation failures (up to 28.4% for GPT-4o-mini). However, for successfully recompiled results, functional correctness improves significantly at higher IR levels, especially for smaller models. For example, GPT-4o-mini improves from 15.3% to 45.8%, and GPT-4.1-nano from 16.9% to 56.9%, when moving from disassembly to HLIL. Larger models show more consistent correctness across all IRs.

Overall, higher-level IRs yield decompiled code that recompiles less often, but if it does, then it is more often functionally correct, especially for smaller models.

c) All-or-Nothing Testing Outcomes: Our functional tests reveal a consistent and surprising pattern: Recompiled outputs from LLMs tend to either pass all unit tests or fail completely. Across all models and IRs, the proportion of partially correct decompilations (functions that pass some, but not all, tests) remains low (3.3%–8.8%). Most functions are either completely correct (6.3%–74.8%) or completely incorrect (19.6%–69.3%). This pattern of binary results indicates that comprehensive functional testing is more appropriate for evaluating LLM-based decompilation than partial correctness metrics, since LLMs tend to either fully reconstruct semantics or fail entirely.

B. Source Code Complexity

Next, we examine how source code complexity affects decompilation performance. Figure 2 shows functional correct decompilation accuracy at cyclomatic complexity levels (1–11) for four representative model-IR combinations. We omit GPT-4.1-nano for brevity, as its performance is very similar to GPT-4o-mini and remains consistently low (see Appendix B).

Surprisingly, we observe only weak and inconsistent correlations between cyclomatic complexity and decompilation success. While performance tends to degrade at higher complexity levels, especially for smaller models, there is no clear monotonic trend. Larger models exhibit more stable performance, although some irregular fluctuations persist. These results suggest that cyclomatic complexity, which captures control-flow branching, does not adequately explain the challenges LLMs face when reconstructing semantics.

To investigate this surprising behavior, we analyzed Halstead complexity metrics [8], which reflect the semantic and cognitive complexity of code. Specifically, they are: (a) Halstead Effort: Estimated mental effort required to understand the code; (b) Halstead Difficulty: Probability of introducing bugs. (c) Halstead Volume: Overall information content.

Figure 3 shows the relationship between functional correctness and each Halstead metric for GPT-4.1, which is representative for well performing models (see Appendix C).

Compared to cyclomatic complexity, Halstead metrics show stronger and more consistent correlations. Specifically, success rates decrease monotonically with increasing Halstead effort, from approximately 86.3%±3.7% at the lowest complexity interval to 11.3%±4.0% at the highest. We observe similar trends for Halstead difficulty and volume, although less pronounced. The confidence intervals remain narrow, supporting the robustness of these trends.

Our findings suggest that semantic and cognitive complexity metrics are better predictors of LLM decompilation difficulty than structural metrics. This has implications for how we evaluate decompilation difficulty and train models. Future work should explore incorporating human-centered complexity measures into LLM training objectives or prompting strategies, and investigate hybrid systems that combine static analysis with LLMs to improve the decompilation of complex binaries.

C. IR Comparative Analysis

Following, we examine the trade-offs between recompilation reliability and functional correctness at different abstraction levels, which directly impact the practical feasibility of using different IRs for LLM-based decompilation.

a) Disassembly Excels at Recompilation, not Always at Semantics: Disassembly consistently exhibits the lowest recompilation error rates across all evaluated models: 1.5% for Claude Sonnet 4 and 4.1% for GPT-4.1. This robustness might be due to the straightforward, low-level semantics of disassembly, which LLMs can more easily map to syntactically valid C code. However, the recompilation success advantage does not always translate into semantic correctness. Especially smaller models can produce more functionally correct code from high-level IRs, despite increased recompilation errors.

Furthermore, the apparent robustness of disassembly might (partially) also be because of overlapping training data (see Section IV), as LLMs are more likely to have found assembly-like patterns during pre-training than structured IRs, which could bias performance in favor of disassembly.

b) HLIL Performance for Smaller Models: For smaller models, HLIL offers significant improvements in functional correctness, with an increase of more than 30 percentage points compared to disassembly (for GPT-4o-mini from 15.3% to 45.8%, and for GPT-4.1-nano from 16.9% to 56.9%). These substantial improvements occur despite higher recompilation error rates for HLIL (e.g., for GPT-4o-mini, 10.6% for disassembly vs. 28.4% for HLIL), indicating that smaller models particularly benefit from the detail and structure provided by high-level IRs. In contrast, larger models improve modestly and inconsistently with HLIL, sometimes even favoring disassembly in terms of overall functional correctness (GPT-4.1: 74.8% vs. 67.4%). This disparity suggests that smaller models might require explicit semantic scaffolding of high-level IRs for reliable decompilation, while larger models efficiently process low-level representations without assistance, possibly due to similar internal transformations.

c) HLIL Performance Across Code Complexity: Our complexity analysis reveals that HLIL-based decompilation is more resilient to increasing code complexity than decompilation from low-level representations. While disassembly offers better performance on low-complexity code across all three Halstead metrics, its performance degrades significantly

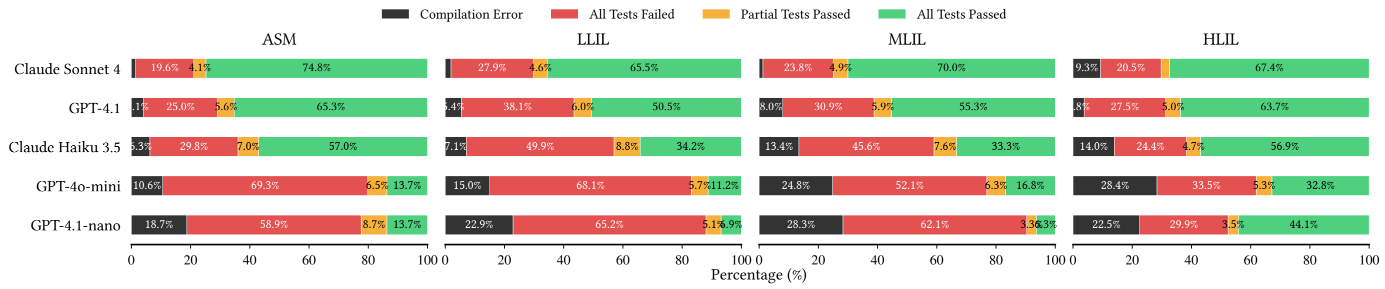

Figure 1: Decompilation performance across the five LLMs and four IR levels that we analyze. This stacked horizontal bar chart shows the distribution of results for each model and IR combination: Compilation errors (black), complete test failures (red), partial test success (orange), and complete test success (green). Models are grouped by size (larger models first).

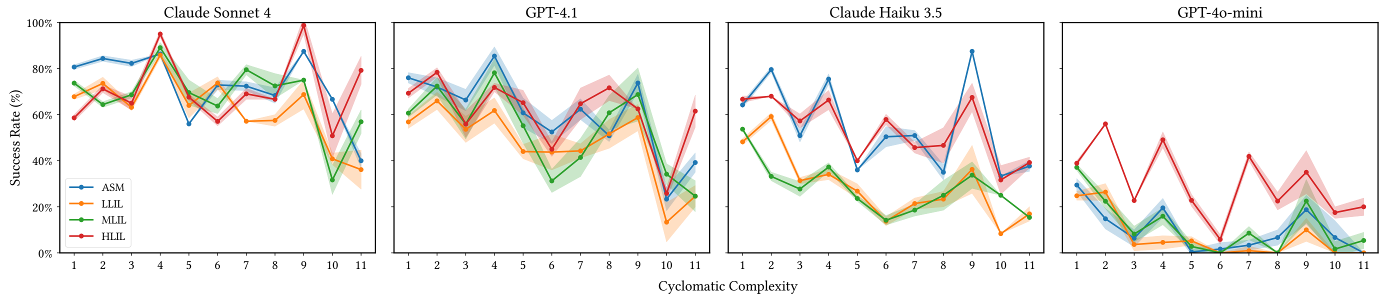

Figure 2: Average decompilation accuracy across cyclomatic complexity levels for four representative LLM-IR combinations. Shaded regions show 95% confidence intervals across ten experimental runs. GPT-4.1-nano is omitted for clarity, as its performance is very similar to that of GPT-4o-mini and remains consistently low.

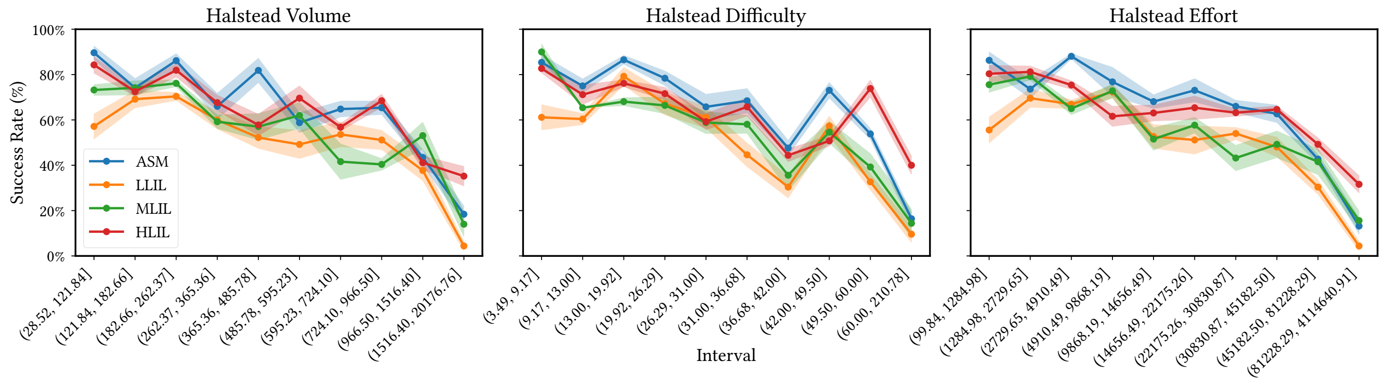

Figure 3: Average decompilation accuracy versus Halstead complexity metrics for GPT-4.1. Shaded regions represent 95% confidence intervals across ten experimental runs.

with increasing complexity (see Figure 3). One possible reason why disassembly offers such good performance on low-complexity code could be dataset leakage (see Section IV). HLIL maintains more stable performance across the Halstead complexity metrics, with much less drastic performance drops as code complexity increases. These complexity-dependent performance characteristics suggest that practical LLM-based decompilers could benefit from selecting the IR based on the target code's expected complexity and the model size.

d) LLIL and MLIL Show Mixed Results: LLIL and MLIL occupy a middle ground between disassembly and HLIL, with performance characteristics that vary significantly across models and complexity levels. In some scenarios, these intermediate levels are promising, but they do not consistently outperform disassembly or HLIL on our metrics. To fully characterize their potential and usefulness for decompilation, further research is needed, which should especially focus on where disassembly's current advantage over LLIL and MLIL lies, in order to possibly adapt it to LLIL/MLIL.

IV. Limitations

While our evaluation provides valuable insights into the effectiveness of hierarchical IRs for LLM-based decompilation and the role of code complexity, some limitations affect the applicability and generalizability of our findings.

a) Time Scope: Our experiments were conducted in June 2025, using publicly available LLM APIs. However, LLMs evolve rapidly: Vendors frequently retrain or update models, often without public change logs, which can lead to variations in performance characteristics over time. Therefore, our findings are a snapshot of current model capabilities, and results may vary for future model versions or newer models.

b) Dataset Limitations and Functional Validation: One obstacle of our work was the scarcity of datasets that allow for functional validation. Most existing benchmarks, such as Human-Eval [4] and Decompile-Eval [16], lack tests or other mechanisms to verify if a decompiled function is semantically equivalent to the original code. Evaluations using them have to rely on superficial or syntactical metrics, which might not accurately reflect true decompilation correctness.

We used `ExeBench` [2] for its unique support for unit testing, allowing verification of functional correctness. However, it only includes ten unit tests per function and is oriented toward simple code with limited algorithmic complexity [17], limiting generalizability to more complex real-world scenarios. To partially address this shortcoming, we applied stratified sampling across complexity levels, though the dataset may still underrepresent the challenges posed by highly complex code. However, our conclusions may not be fully generalizable to commercial-of-the-shelf software.

c) Dataset Leakage and Training Data Overlap: As with any work involving proprietary LLMs, we cannot definitively rule out dataset leakage, that is, overlap between our evaluation dataset and the LLMs' pre-training corpora, as ExeBench is hosted on GitHub and could have been included during pre-training. However, we believe it would have minimal impact on our results for two key reasons: First, our evaluation is solely on binary-level inputs (i.e., disassembly and IR), which are derived from binaries compiled locally and analyzed with `Binary Ninja`. These representations are unlikely to have appeared in the training data, even if the source code has. Second, we never provided the LLMs with the original source code during inference. We constructed our prompts exclusively from the disassembled or IRs (see Appendix A), aiming to prevent direct pattern matching with known code.

d) Compiler Optimization Limitations: We performed our analysis on binaries compiled with GCC without optimizations (`-O0`). We disabled optimizations to preserve most source-level structure and complexity. Although compiler variation could also provide additional insights, systematically exploring these parameters is a separate research question and beyond our focus on hierarchical IRs. Real-world binaries are typically compiled with aggressive optimizations (e.g., `-O2` or `-O3`) though, which introduce structural transformations, such as through loop unrolling and inlining. These optimizations may affect the effectiveness of high-level IRs or the observed trade-offs between levels of abstraction and functional correctness. Therefore, our results may not be directly applicable to optimized binaries or alternative build configurations. Future work should investigate how the usefulness of IRs varies across optimization configurations and build ecosystems.

V. Conclusion

In this paper, we provide the first systematic analysis of the potentials of hierarchical IRs for LLM-based decompilation. Our results show that model size is a key factor for decompilation quality: Larger LLMs exhibit consistent performance across IRs, while smaller models significantly benefit from higher-level IRs.

Overall, HLIL-based decompilation achieves higher functional correctness for complex code, while decompiled disassembly offers greater recompilation success. Our complexity analysis also shows that disassembly is more effective for simple code, while HLIL is more resilient to increases in code complexity. For real-world code that is more complex than our dataset, higher-level IRs could show even greater improvements over disassembly. Complexity-driven IR selection could also improve LLM-based decompilation.

Our results show that semantic and cognitive complexity measures, especially Halstead metrics, more accurately predict decompilation difficulty than structural metrics (i.e., cyclomatic complexity). This suggests that traditional measures do not fully capture the challenges LLMs face, and that evaluation standards for LLM-based decompilers should be reconsidered, prioritizing cognitive and semantic complexity metrics.

Future work should explore optimized binaries, more complex and real-world targets, and adaptive IR selection strategies that consider model capacity and code complexity. Leveraging extensible IR frameworks, such as MLIR [10], and tailoring IRs to the capabilities of LLMs could further improve decompilation performance. Finally, a systematic investigation of the error and failures at different IR levels and model sizes would provide valuable insights for improving decompilation reliability. Understanding whether failure types differ between, for instance, disassembly and HLIL, can guide the development of more robust hybrid decompilation processes.

Acknowledgements

This research was supported in part by the Agencia Estatal de Investigación (AEI, Spanish State Research Agency) and the European Regional Development Fund (ERDF) of the European Union (EU) under grant PID2023-151467OA-I00 (CRAPER), by AEI and the EU Next Generation EU fund (NGEU) via the Recovery, Transformation and Resilience Plan (PRTR) under grant TED2021-131115A-I00 (MIMFA), by the Spanish National Cybersecurity Institute (INCIBE) and NGEU via PRTR under grant *Proyecto Estratégico Ciberseguridad EINA UNIZAR*, by the Dirección General de Promoción Industrial e Innovación of the Gobierno de Aragón (University, Industry and Innovation Department of the Aragonese Government) under grant *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-23R), and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972.

Any opinions, findings, and conclusions or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the respective funding agencies.

References

[1] Anthropic. *Claude*. URL: https://www.anthropic.com (visited on 06/15/2025).

[2] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. d. S. Magalhães, and M. F. P. O'Boyle. "ExeBench: An ML-scale Dataset of Executable C Functions." In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. ACM. 2022. DOI: 10.1145/3520312.3534867.

[3] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell. "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?" In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. 2021. DOI: 10.1145/3442188.3445922.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. "Evaluating Large Language Models Trained on Code." July 14, 2021. arXiv: 2107.03374 [cs.LG].

[5] C. Cifuentes and K. J. Gough. "Decompilation of Binary Programs." In: *Software: Practice and Experience* 25.7 (1995), pp. 811–829. DOI: 10.1002/spe.4380250706.

[6] Y. Feng, B. Li, X. Shi, Q. Zhu, and W. Che. "ReF Decompile: Relabeling and Function Call Enhanced Decompile." Feb. 17, 2025. arXiv: 2502.12221 [cs.SE].

[7] Y. Feng, D. Teng, Y. Xu, H. Mu, X. Xu, L. Qin, Q. Zhu, and W. Che. "Self-Constructed Context Decompilation with Fine-grained Alignment Enhancement." Oct. 3, 2024. arXiv: 2406.17233 [cs.SE].

[8] M. H. Halstead. *Elements of Software Science*. Operating and Programming Systems Series. Elsevier Science Inc., 1977. ISBN: 0444002057.

[9] Hex-Rays. *IDA Pro: Interactive Disassembler Professional*. URL: https://hex-rays.com/ida-pro/ (visited on 06/15/2025).

[10] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation." In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021. DOI: 10.1109/CGO51591.2021.9370308.

[11] P. Liu, J. Sun, L. Chen, Z. Yan, P. Zhang, D. Sun, D. Wang, and D. Li. "Control Flow-Augmented Decompiler Based on Large Language Model." Mar. 10, 2025. arXiv: 2403.05286v1 [cs.PL].

[12] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende. "Prompt Engineering in Large Language Models." In: *Data Intelligence and Cognitive Informatics*. 2024. DOI: 10.1007/978-981-99-7962-2_30.

[13] T. J. McCabe. "A Complexity Measure." In: *IEEE Transactions on Software Engineering* 4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[14] National Security Agency (NSA). *Ghidra Software Reverse Engineering Framework*. URL: https://ghidra-sre.org/ (visited on 06/15/2025).

[15] OpenAI. *ChatGPT*. URL: https://chat.openai.com/ (visited on 06/15/2025).

[16] H. Tan, Q. Luo, J. Li, and Y. Zhang. "LLM4Decompile: Decompiling Binary Code with Large Language Models." Oct. 22, 2024. arXiv: 2403.05286 [cs.PL].

[17] H. Tan, X. Tian, H. Qi, J. Liu, Z. Gao, S. Wang, Q. Luo, J. Li, and Y. Zhang. "Decompile-Bench: Million-Scale Binary-Source Function Pairs for Real-World Binary Decompilation." May 19, 2025. arXiv: 2505.12668 [cs.SE].

[18] Vector 35. *Binary Ninja: A Reverse Engineering Platform*. URL: https://binary.ninja/ (visited on 06/15/2025).

[19] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing, et al. "Judging LLM-as-a-judge with MT-bench and Chatbot Arena." In: *Procedings of the Advances in Neural Information Processing Systems*. 2023.

Appendix

A. Prompts

Following, we provide the full text of all prompts that we used in our LLM-based decompilation experiments. We designed each prompt based on the minimal, structured approach that we described in Section II. This ensures consistency across models and IRs while maintaining interpretability and reproducibility of our process.

1) Disassembly Prompt

> You are an expert reverse engineer. Decompile the following assembly code to equivalent C source code. Analyze the assembly instructions, registers, memory operations, and control flow to produce functional and compilable C source code. Assembly code to decompile:
>
> — asm code —
>
> Output format: Only C code, no explanations, no markdown formatting. Function name: func0. Variable names: var0, var1, var2, etc. Include necessary headers (#include) if needed. Do not include “‘c or “‘ markers.

2) Low-Level IL (LLIL) Prompt

You are an expert reverse engineer. Decompile the following LLIL code to equivalent C source code. LLIL is Binary Ninja's low-level IR that abstracts away architecture-specific details while preserving low-level semantics. Analyze the LLIL operations, memory accesses, and control structures to produce functional and compilable C source code. LLIL code to decompile:

— llil code —

Output format: Only C code, no explanations, no markdown formatting. Function name: func0. Variable names: var0, var1, var2, etc. Include necessary headers (#include) if needed. Do not include "'c or "' markers.

3) Medium-Level IL (MLIL) Prompt

You are an expert reverse engineer. Decompile the following MLIL code to equivalent C source code. MLIL is Binary Ninja's medium-level IR that performs data flow analysis and eliminates many low-level artifacts. Analyze the MLIL operations, variables, and control flow to produce functional and compilable C source code. MLIL code to decompile:

— mlil code —

Output format: Only C code, no explanations, no markdown formatting. Function name: func0. Variable names: var0, var1, var2, etc. Include necessary headers (#include) if needed. Do not include "'c or "' markers.

4) High-Level IL (HLIL) Prompt

You are an expert reverse engineer. Decompile the following HLIL code to equivalent C source code. HLIL is Binary Ninja's highest-level IR that recovers high-level constructs like loops, conditionals, and function calls. Analyze the HLIL operations and structures to produce functional and compilable C source code. HLIL code to decompile:

— hlil code —

Output format: Only C code, no explanations, no markdown formatting. Function name: func0. Variable names: var0, var1, var2, etc. Include necessary headers (#include) if needed. Do not include "'c or "' markers.

B. *GPT-4.1-nano Complete Results*

We excluded GPT-4.1-nano from the main text for brevity, as its performance is consistently at the lower end across all evaluated metrics and IRs. Figure 4 shows its decompilation performance. Overall, GPT-4.1-nano shows similar patterns to GPT-4o-mini, but with inferior results.

C. *Halstead Complexity Metrics for All Models*

In Section III, we present the Halstead complexity analysis for GPT-4.1 as a representative model. In this appendix, we provide the complete Halstead metrics results for the

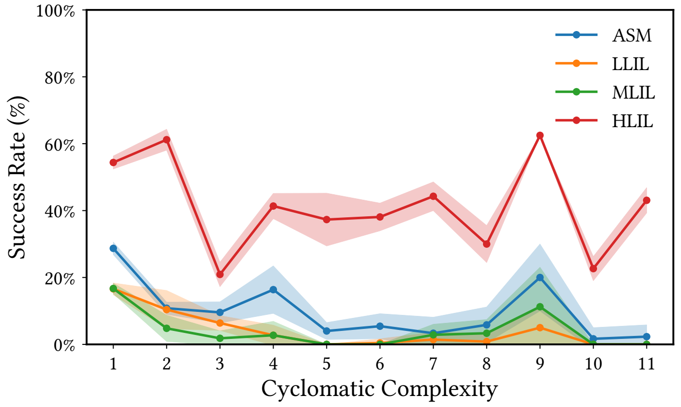

Figure 4: Decompilation performance results for GPT-4.1-nano across all evaluated IR levels and complexity metrics.

remaining evaluated LLMs. Figures 5 to 8 (on the next page) show the relationship between decompilation accuracy and Halstead effort, difficulty, and volume for each model. These results confirm that cognitive complexity metrics consistently outperform structural metrics, like cyclomatic complexity, as predictors of LLM decompilation difficulty across all model sizes and architectures.

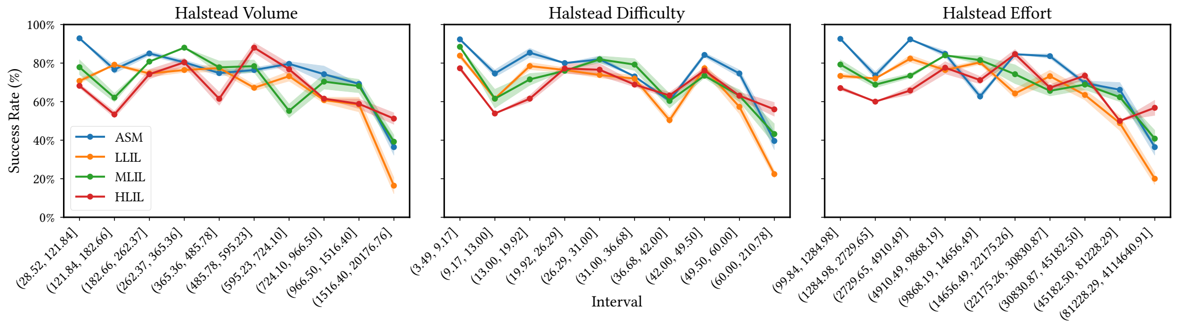

Figure 5: Average decompilation accuracy versus Halstead complexity metrics for Claude Sonnet 4. Shaded regions represent 95% confidence intervals across ten experimental runs.

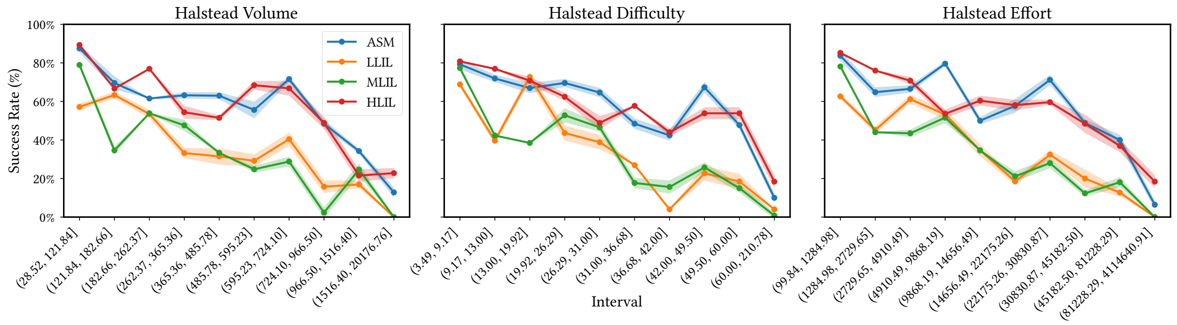

Figure 6: Average decompilation accuracy versus Halstead complexity metrics for Claude Haiku 3.5. Shaded regions represent 95% confidence intervals across ten experimental runs.

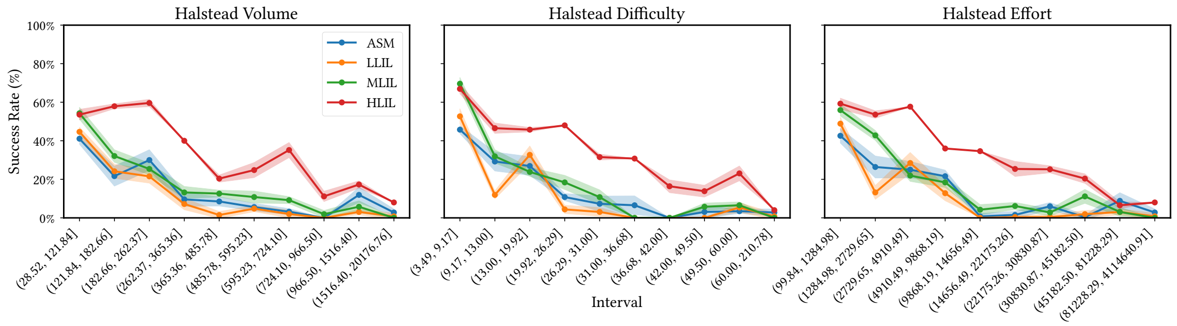

Figure 7: Average decompilation accuracy versus Halstead complexity metrics for GPT-4o-mini. Shaded regions represent 95% confidence intervals across ten experimental runs.

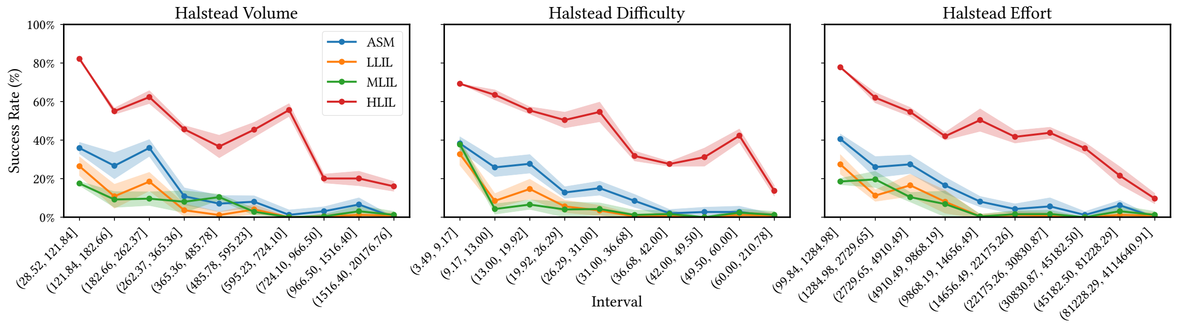

Figure 8: Average decompilation accuracy versus Halstead complexity metrics for GPT-4.1-nano. Shaded regions represent 95% confidence intervals across ten experimental runs.