Binary Lifting into LLVM IR with Large Language Models [*]

Sieun Jo, Jiwon Lee, Dong Hoon Lee, and Wonsuk Choi[†]

Korea University, Seoul, South Korea
{sejo, hisdory728, donghlee, beb0396}@korea.ac.kr

Abstract

The process of converting binaries into intermediate representation (IR), known as lifting, is a pivotal role in binary analysis. Existing lifting tools mainly rely on rule-based techniques that generate low-level labels in the resulting IR and are sensitive to environmental constraints such as version compatibility of the Operating System (OS), LLVM, and auxiliary tools. By contrast, recent Large Language Model (LLM) research focuses on text generation for high-level languages, with limited attention to low-level analysis. Also, support for Windows binaries is limited, and recompilation of the generated IR is often not guaranteed.

To bridge the gap, we present an LLM-based lifting framework that produces highly readable IR and is robust to environmental variation. By incorporating observed error patterns during IR generation and execution into the system prompt, the success rate is significantly improved, compared with a question-based prompt. The generated IR is validated and compared to McSema, a representative lifting tool. This study is the first attempt to apply LLMs to binary lifting for IR and suggests that LLMs could serve as a viable alternative to existing lifting tools.

Keywords: Binary Lifting, LLVM IR, Large Language Models, System Prompt

1 Introduction

Binary analysis of executable files is a key task in security research and software engineering, especially in environments where source code is not accessible. In particular, precise binary analysis is required to perform tasks such as binary patching, vulnerability detection, and reverse engineering. However, raw binaries and assembly code are low-level representations that are difficult for humans to understand and analyze. To address this, binary lifting techniques that convert executable files into IR have been studied.

McSema [2] and RetDec [7], which are representative lifting tools, use rule-based techniques. In this case, control structures such as loops and conditional statements are expressed as address-offset-based basic block labels (e.g., `inst_13ae`), as shown on the left of Figure 1, making intuitive understanding difficult for analysts. McSema and RetDec also require manually developed rules for lifting, and their development is currently suspended. They are sensitive to environmental constraints such as version compatibility of the OS, LLVM, and auxiliary tools. Although the official documentation states that McSema supports both Linux and Windows binaries, Windows support has remained at an experimental stage from the outset. Even now, lifting failures on Windows are repeatedly reported in GitHub issues, limiting practical use. The IR generated by RetDec cannot be recompiled, which makes equivalence verification difficult and constrains the workflow for modification, patching, and redistribution. In addition, toolchain integration is limited, so practical usability and reproducibility are low.

[*]Proceedings of the 9th International Conference on Mobile Internet Security (MobiSec'25), Article No. 84, December 16-18, 2025, Sapporo , Japan. © The copyright of this paper remains with the author(s).

[†]Corresponding author

On the other hand, LLMs can preserve high-level semantics and generate highly readable IR through learning and reasoning. In Figure1, the loop headers in the LLM-generated IR are expressed with meaningful block names such as outer_cond, inner_cond, and init_loop, which makes data flow easier to track. This approach also has fewer environmental constraints than rule-based tools, and IR can be generated with the same pipeline for both Windows and Linux.

However, LLMs are not suitable for directly processing large-scale binaries due to text-based tokenization and I/O token limits. Therefore, this study aims to overcome the limitations of existing rule-based tools and provide a way to make the most of the benefits of LLMs.

The main contributions of the paper can be summarized as follows:

- To generate IRs that are highly readable and robust against environmental constraints such as OS, LLVM, and auxiliary tools, we apply LLMs to lifting binaries into IR. Specifically, we propose a pipeline that converts the binary into assembly code, subdivides it into function units, and uses them as LLM input.
- After observing errors frequently occurring during IR generation and the execution process, we make rules to avoid the errors and reflect them in the system prompt. The proposed system prompt design is evaluated by comparing common question-based prompts. The success rate of validating whether each function-level IR adheres to LLVM syntax and can be successfully converted to bitcode (.bc) format increases from 80.9% to 94.1% in our evaluation. The success rate of validating whether the actual execution result matched the output of the original source code increased from 39.09% to 78.18%.
- The feasibility of the proposed IR generation method is verified against McSema with respect to (i) the residual phi/alloca/call counts and (ii) the structural and statistical similarity with the reference IR. It turns out that the LLM-based technique has fewer differences from the reference IR in all three instructions (phi, alloca, and call), and text-based similarity improves from 0.33% with McSema to 7.49% with the proposed method. There is no significant difference between McSema and LLM in Instruction Count Relative Error and Memory Access Proportion Difference, but LLM-based IR shows lower errors on average. These results demonstrate the potential of LLM-based lifting as a complement for existing tools.

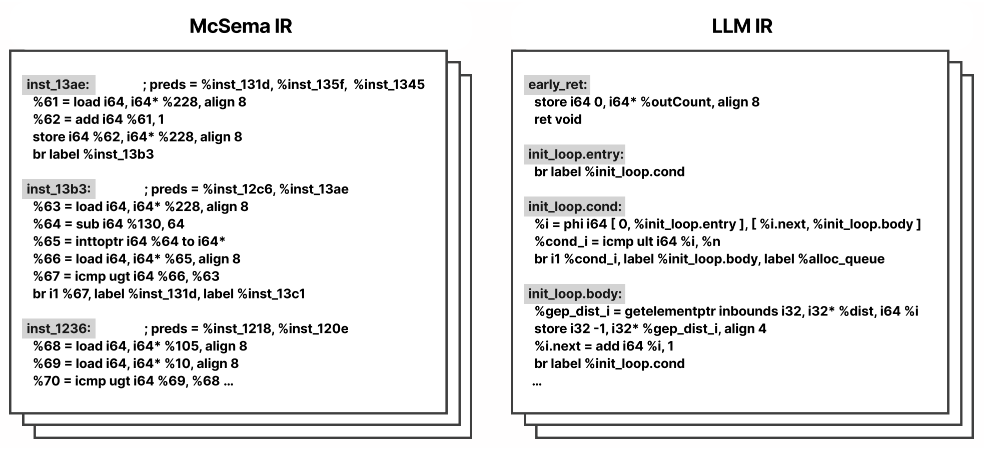
2 Background

2.1 LLVM IR

LLVM is the Low-Level Virtual Machine proposed in [8] and is a modular compiler infrastructure. LLVM performs code optimization using intermediate representation (IR). Since IR is language-independent and follows the form of static single assignment (SSA), each variable is assigned only once, making it easier to optimize and analyze. In addition, it has strong static typing, which allows modeling of language characteristics such as structures of high-level languages and exception handling, so it can be used not only for compiler optimization but also for binary analysis and security research.

2.2 LLM

LLMs are transformer-based AI models specialized in text generation. They are used for various tasks such as question answering, information retrieval, text generation and summarization, and



McSema IR

```
inst_13ae:              ; preds = %inst_131d, %inst_135f,  %inst_1345
  %61 = load i64, i64* %228, align 8
  %62 = add i64 %61, 1
  store i64 %62, i64* %228, align 8
  br label %inst_13b3

inst_13b3:              ; preds = %inst_12c6, %inst_13ae
  %63 = load i64, i64* %228, align 8
  %64 = sub i64 %130, 64
  %65 = inttoptr i64 %64 to i64*
  %66 = load i64, i64* %65, align 8
  %67 = icmp ugt i64 %66, %63
  br i1 %67, label %inst_131d, label %inst_13c1

inst_1236:              ; preds = %inst_1218, %inst_120e
  %68 = load i64, i64* %105, align 8
  %69 = load i64, i64* %10, align 8
  %70 = icmp ugt i64 %69, %68 ...
```

LLM IR

```
early_ret:
  store i64 0, i64* %outCount, align 8
  ret void

init_loop.entry:
  br label %init_loop.cond

init_loop.cond:
  %i = phi i64 [ 0, %init_loop.entry ], [ %i.next, %init_loop.body ]
  %cond_i = icmp ult i64 %i, %n
  br i1 %cond_i, label %init_loop.body, label %alloc_queue

init_loop.body:
  %gep_dist_i = getelementptr inbounds i32, i32* %dist, i64 %i
  store i32 -1, i32* %gep_dist_i, align 4
  %i.next = add i64 %i, 1
  br label %init_loop.cond
  ...
```

Figure 1: Two IRs generated by McSema and LLM

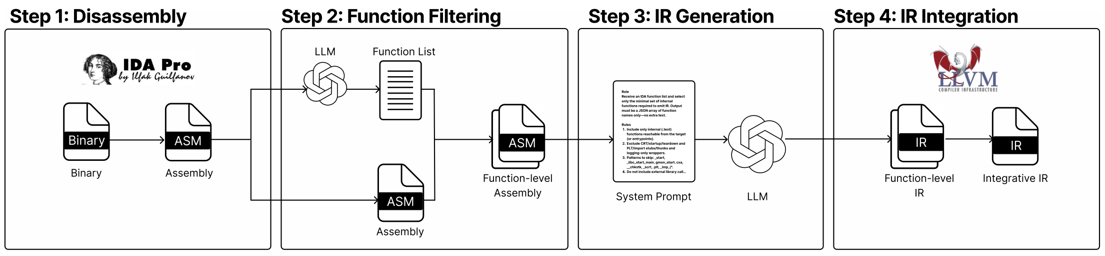
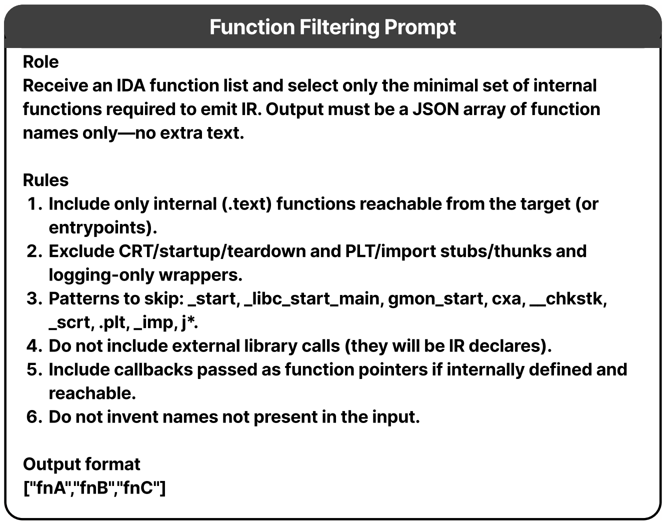


Figure 2: Overall structure of the LLM-based IR generation pipeline

code generation. Until now, research has mainly focused on high-level languages such as Python, C, and Java [9, 10], whereas research targeting low-level representations has been relatively lacking. Recently, in [6], the authors evaluate how well LLMs understand IR in terms of structural, syntactic, semantic, and execution reasoning, and studies in DeGPT[5] and LLM4Decompile[12] propose decompilation studies that restore assembly and binaries into high-level languages. However, decompiler aims to recover C-like source code for improved readability. Therefore, DeGPT does not assume recompilation, and LLM4Decompile's recompilation success rate is limited, averaging about 27.32–64.18%. Thus, obtaining IR from decompiler output has clear limitations. IR is architecture-independent, so all Instruction Set Architecture (ISA) can be handled with the same analysis pipeline. The two approaches differ in purpose and strengths and can be used in a complementary way. However, there has been little research on lifting binaries to IR using LLMs.

2.3 System Prompt Optimization

LLM input is divided into user prompts and system prompts. Among them, the system prompt provides guidelines to be followed when the model generates output, such as output format constraints and task descriptions. The user prompt varies with each call, while the system prompt remains fixed as the initial setting. Therefore, it plays a key role in enabling the model to generate consistent and reliable outputs in various environments. Recent studies have


Function Filtering Prompt

Role
Receive an IDA function list and select only the minimal set of internal functions required to emit IR. Output must be a JSON array of function names only—no extra text.

Rules
1. **Include only internal (.text) functions reachable from the target (or entrypoints).**
2. **Exclude CRT/startup/teardown and PLT/import stubs/thunks and logging-only wrappers.**
3. **Patterns to skip: _start, _libc_start_main, gmon_start, cxa, __chkstk, _scrt, .plt, _imp, j*.**
4. **Do not include external library calls (they will be IR declares).**
5. **Include callbacks passed as function pointers if internally defined and reachable.**
6. **Do not invent names not present in the input.**

Output format
["fnA","fnB","fnC"]

Figure 3: Prompt employed for function filtering with GPT-5[11]

empirically shown the effect of prompt quality on model performance, and accordingly, research on prompt engineering [13, 4] and system optimization techniques [14, 1] have been actively conducted. In this study, error logs generated during the IR verification stage were collected and analyzed, and rules were derived from them to minimize structural violations in IR generation.

3 LLM-based IR Generation Pipeline

In this section, we explain the overall structure of the LLM-based IR generation pipeline proposed in this paper. The pipeline consists of two main parts: the data processing step and the IR generation step.

3.1 Phases of the Pipeline

The IR generation pipeline using LLM proposed in this paper can be seen in Figure 2. The pipeline is largely divided into two phases: data-processing phase and IR-generation phase.

Data processing. Since LLMs cannot directly interpret the raw binary format, it is necessary to convert it into a text format that they can process. In this study, function-level assembly, which serves as the unit of LLM input, is extracted from the target binary using IDA Pro[3].

1. Function Enumeration: IDA Pro was used to collect the complete list of functions in the binary. The list includes not only user-defined functions but also standard library functions such as printf, symbols provided by external libraries, and import/Procedure Linkage Table (PLT) entries.

2. Essential Function Selection: Due to the input token size limitation of LLMs, it is not feasible to provide the entire binary assembly at once. Therefore, the assembly is divided into function units, and only the functions directly required for IR generation are selected. This process uses GPT-5, and for this purpose, the list of functions collected from IDA

Pro is organized and provided in JSON format. The JSON includes metadata such as function name, start and end address, size, segment, category, thunk, and function call list. The prompt employed is shown in Figure 3.

3. Assembly code at the function level: Assembly code is extracted for each selected function using IDA Pro. These extracted units serve as independent inputs to the LLM, and the resulting function-level IRs are subsequently merged into a unified IR representation.

LLM-Based IR Generation

1. **Function-level IR generation:**
   Each function's assembly code is provided as input to the LLM to generate function-level LLVM IR. Since LLMs cannot produce binary outputs, we constrain the output to text-based LLVM IR (.ll).

2. **Module-level IR integration:**
   This research ultimately aims to obtain module-level IR. The function-level IR merging process follows LLVM's standard procedure:

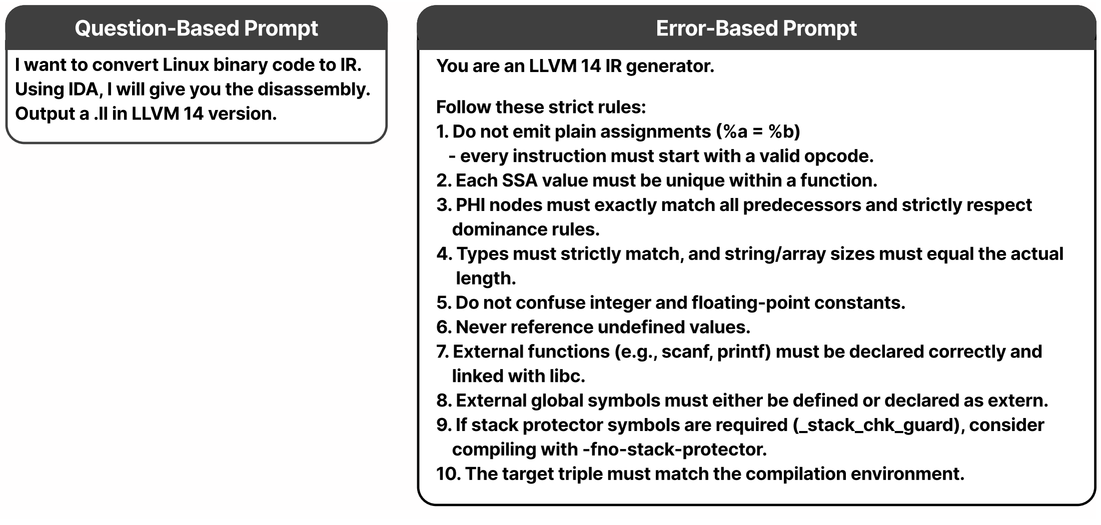
   (a) **Text IR (.ll → bitcode .bc):** Each function-level `.ll` file was converted to bitcode (`.bc`) using `llvm-as`.

   (b) **Bitcode linking (.bc → module-level .bc):** We merged the per-function `.bc` files into a module-level (`module.bc`) with `llvm-link`. No optimization passes were applied.

   (c) **Disassembly for inspection (.bc → text .ll):** We disassembled the integrated module back to text IR (`module.ll`) using `llvm-dis` and used this as the final integrated IR.

3.2 Error-Driven System Prompt Design

In order to design effective system prompts, we collected and analyzed error logs generated from IR produced using question-based prompts. The frequently observed error types were then mapped into ten rules and incorporated into the system prompt. Both the question-based and error-driven prompts we used are illustrated in Figure 4. LLVM IR requires that all variables be declared prior to use. During the process of converting source code into IR, variable declarations and type checks are already completed. Therefore, errors corresponding to Rule 6 do not occur. In contrast, when converting binaries into IR, variable and function declaration information is absent, making undefined value errors structurally more likely. Furthermore, additional errors may arise during the integration of function-level IR. For instance, conflicts can occur when functions are declared redundantly or when the signatures of external functions are defined inconsistently. Errors related to Rule 10 arise when the target triple does not match the actual compilation environment. During the function-level IR integration stage, the presence of multiple target triples or data layouts does not immediately produce errors, as a single triple is selected. However, when converting a binary into an executable, inconsistencies in the Application Binary Interface (ABI), such as differences in structure layout, alignment rules, or calling conventions-may cause incorrect memory accesses at runtime, ultimately resulting in program failures. Among the proposed ten rules, Rules 1 through 6 address errors associated with IR syntax violations, whereas Rules 7 through 10 focus on errors that arise during IR integration, linkage with external libraries, and the execution process.



Question-Based Prompt

I want to convert Linux binary code to IR.
Using IDA, I will give you the disassembly.
Output a .ll in LLVM 14 version.

Error-Based Prompt

You are an LLVM 14 IR generator.

Follow these strict rules:
1. Do not emit plain assignments (%a = %b)
   - every instruction must start with a valid opcode.
2. Each SSA value must be unique within a function.
3. PHI nodes must exactly match all predecessors and strictly respect dominance rules.
4. Types must strictly match, and string/array sizes must equal the actual length.
5. Do not confuse integer and floating-point constants.
6. Never reference undefined values.
7. External functions (e.g., scanf, printf) must be declared correctly and linked with libc.
8. External global symbols must either be defined or declared as extern.
9. If stack protector symbols are required (_stack_chk_guard), consider compiling with -fno-stack-protector.
10. The target triple must match the compilation environment.

Figure 4: Error-driven rules in the system prompt

4 Implementation

The experimental environment comprises LLVM/Clang 14.0.6 and IDA Pro 9.1, utilizing OpenAI GPT-5 API as LLM model. During the preprocessing phase for LLM input data, function-level assembly extraction employed IDA Pro's Python API modules (idc, idautils, idaapi, ida_lines). The extracted assembly includes function names and start and end addresses. For IR integration, we utilized LLVM/Clang utilities including `llvm-as`, `llvm-link`, and `llvm-dis` commands.

5 Evaluation

This study conducted experiments on Linux binaries, selecting a total of 11 algorithms for evaluation. The test suite comprises representative sorting and search algorithms, all implemented in C and compiled with the `-O0` optimization flag, i.e., no optimization used. Each algorithm consists of two functions: a main function responsible for program execution and a functional component that performs the algorithmic operations. The target algorithms include BFS, DFS, Binary Search, Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Heap Sort, Merge Sort, Linear Search, and Dijkstra's algorithm.

5.1 Experimental Validation of System Prompt Effectiveness

The performance of the designed prompt was evaluated by comparison with commonly used question-based prompts (Figure 4). The comparative evaluation consists of two primary components: (i) function-level IR validation and (ii) module-level IR executability and equivalence to ground-truth binary results. To account for the stochastic nature of LLM outputs, each IR was generated ten times per test case.

Algorithm	Question-Based Prompt		Error-Based Prompt	
	main	function	main	function
BFS	**90%**	**90%**	80%	**90%**
Binary Search	**100%**	80%	**90%**	**100%**
Bubble Sort	**100%**	**100%**	**100%**	**100%**
DFS	70%	80%	80%	**100%**
Dijkstra	50%	70%	**100%**	**90%**
Heap Sort	40%	80%	**90%**	**100%**
Insertion Sort	**90%**	**100%**	**100%**	**100%**
Linear Search	**100%**	**90%**	**90%**	**100%**
Merge Sort	40%	**100%**	70%	**90%**
Quick Sort	50%	**90%**	**100%**	**100%**
Selection Sort	80%	**90%**	**100%**	**100%**
Average	73.6%	88.2%	**90.9%**	**97.3%**

Table 1: Comparison of function-level IR validation outcomes under question-based prompt and error-based prompt.

5.2 Function-level IR Validation

IR validation was conducted based on whether each function-level IR adheres to LLVM syntax and can be successfully converted to bitcode (.bc) format. We collected 220 results by performing 10 trials for each of the 22 functions. The question-based prompt achieved an 80.9% success rate, while the error-based prompt demonstrated a 94.1% success rate. Furthermore, algorithms achieving at least 80% success rates across 10 trials per function increased from 14 out of 22 (63.64%) for the question-based prompt to 21 out of 22 (95.45%) for the error-based prompt. These findings demonstrate that the error-based prompt substantially enhances the stability of function-level IR generation. The detailed results are presented in Table 1.

5.3 Module-level IR Executability and Equivalence Evaluation

For each prompt, the module-level IR was compiled to generate an executable file, and it was evaluated whether the actual execution result matched the output of the original source code. The total metric in Table 2, Table 3 was calculated in two ways. The left value reports the share of runs where all function-level IRs passed validation. The right value reports the share that successfully produced an executable and yielded output identical to the reference executable.

The question-based prompts showed a relatively high success rate in some simple algorithms (bubble sort, insertion sort, and linear search), but their consistency with execution results dropped significantly for more complex algorithms. In addition, the rate of output equivalence with the reference executable was only 39.09% on average. On the other hand, error-based prompts exhibited overall stability. The proportion of runs in which all function-level IRs passed validation exceeded 90% for most algorithms, the average proportion that successfully produced an integrated IR and yielded output identical to the reference executable was 78.18% Although performance for BFS was somewhat lower, error-based prompts still demonstrated clear improvements in executability and functional consistency compared to the question-based prompts. Also, Conditioned on all function-level IRs passing validation for each algorithm, the proportion of runs that produced output identical to the reference executable increased substantially-from 60.56% with question-based prompts to 87.75% with error-based prompts.

Algorithm	1	2	3	4	5	6	7	8	9	10	$p(\circ)$	$p(\circledcirc)$
BFS			○	◎	◎	◎	◎	○	◎	◎	80%	60%
Binary Search	◎	◎	◎	◎	◎	○	○	○			80%	50%
Bubble Sort	○	○	◎	◎	○	◎	◎	◎	○	◎	**100%**	60%
DFS	◎	◎			◎	◎	◎	○			60%	50%
Dijkstra						◎			◎	◎	30%	30%
Heap Sort				○		○				◎	30%	10%
Insertion Sort	◎	◎	◎		◎	◎	○	◎	○	○	**90%**	60%
Linear Search	◎	◎	○	○	○	◎	◎	○	○		**90%**	40%
Merge Sort		◎		◎		○				◎	40%	30%
Quick Sort					○	◎		◎		◎	40%	30%
Selection Sort	○	○			○	○		○	○	◎	70%	10%

Table 2: Question-Based Prompt (◎: Module-level success, ○: Function-level success)

Algorithm	1	2	3	4	5	6	7	8	9	10	$p(\circ)$	$p(\circledcirc)$
BFS	◎		◎	◎		○	◎	○		○	70%	40%
Binary Search	◎	◎	◎	◎	◎	◎		◎	◎	◎	**90%**	**90%**
Bubble Sort	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	**100%**	**100%**
DFS		◎	◎	○	◎	◎		◎	○	◎	80%	60%
Dijkstra	◎	◎	◎	◎	◎	◎		◎	◎	◎	**90%**	**90%**
Heap Sort	◎	◎	◎	◎	◎	◎		◎	◎	◎	**90%**	**90%**
Insertion Sort	◎	◎	◎	◎	◎	◎	○	◎	○	◎	**100%**	80%
Linear Search	◎	◎	◎	◎	◎	◎		○	◎	◎	**90%**	80%
Merge Sort			◎	◎	◎	○	◎	◎	○		70%	50%
Quick Sort	○	◎	◎	◎	◎	◎	◎	◎	◎	◎	**100%**	**90%**
Selection Sort	◎	◎	◎	◎	◎	◎	○	◎	◎	◎	**100%**	**90%**

Table 3: Error-Based Prompt (◎: Module-level success, ○: Function-level success)

5.4 Benchmarking Against McSema

In order to verify the feasibility of the LLM-based IR generation method proposed in this study, a comparative evaluation was performed with McSema, a representative lifting tool. The comparative experiments consisted of (i) the residual phi/alloca/call counts and (ii) the structural and statistical similarity with the reference IR. In order to compare the performance of McSema and LLM, the IR file converted from the source code was set as the reference. Since both techniques generate IR by estimating information lost during the lifting process, differences from the reference IR may occur. Accordingly, we compared how similar the IR generated by each technique was to the reference IR. Normalization was applied to each experiment, and in the case of LLM, the results of five repeated experiments under the same conditions were averaged and used.

5.4.1 Residual phi/alloca/call counts

In LLVM IR, phi represents variable selection at control-flow joins in SSA form, alloca represents memory allocation for a function's stack frame, and call represents a function call. This study

Algorithm	Original			McSema			LLM		
	phi	alloca	call	phi	alloca	call	phi	alloca	call
BFS	0	21	28	14	0	53	8	4	8
Binary Search	0	14	17	10	0	38	4	2	3
Bubble Sort	0	10	13	9	0	40	5	1	3
DFS	0	19	32	16	0	62	5.6	3	13.8
Dijkstra	0	21	27	12	0	39	8.8	9	7
Heap Sort	2	20	27	18	0	46	10	1	7
Insertion Sort	1	9	12	11	0	38	3	1	3
Linear Search	0	10	12	9	0	41	2	1	3
Merge Sort	0	18	24	15	0	50	10	1	6
Quick Sort	0	11	16	20	0	42	7	1	5
Selection Sort	0	10	13	9	0	38	4	1	3
Average	0.27	14.82	20.09	13	0	44.27	6.13	2.27	5.62

Table 4: Remaining phi/alloca/call counts after normalization (LLM: average of 5 repetitions)

evaluated structural similarity by comparing the number of corresponding instructions with the reference. However, differences in expression, such as replacing alloca with an SSA variable or arranging phi differently after branching, may lead to variations in the counts, so normalization was performed before comparison.

According to Table 4, the reference IR had on average 0.27 phi, 14.82 alloca, and 20.09 call instructions. McSema generated more instructions than the reference in phi (13.00) and call (44.27), and did not include alloca at all, thus failing to reflect stack variable modeling.This is because McSema manages stack variables via an internal memory model rather than emitting alloca instructions. On the other hand, the LLM-based IR was closer to the reference values than McSema in phi (6.13) and alloca (2.27). The call showed a decreasing trend, but numerically the difference from the reference was relatively smaller than that of McSema. This indicates that LLM produces a more compact IR overall, with high quantitative similarity to the reference IR in terms of statistical distribution.

Taken together, LLM has the limitation of restoring fewer function calls, but compared to McSema, it did not generate unnecessary nodes and maintained better numerical consistency with the reference IR. Therefore, our results show that LLM produces IR that is overall more structurally similar to the reference IR than McSema and suggest that LLMs could serve as a complementary alternative to existing lifting tools.

5.4.2 Structural and Statistical Similarity Analysis

In order to quantitatively evaluate the similarity with the reference IR, text-based similarity analysis using the diff command was performed. In the results of Table 5, McSema showed 0.33% and LLM 7.49%, indicating that LLM achieved relatively higher similarity to the reference. In addition, Instruction Count Relative Error was calculated to measure the structural complexity of the IR, and Memory Access Proportion Difference, which measures the difference in the proportion of memory access-related instructions (e.g., load, store) among all commands, was also evaluated. In both indicators, the performance difference between McSema and LLM does not appear to be significant, but the IR generated by LLM was closer to the reference IR.

As a result of analyzing the diff similarity results of LLM by algorithm, higher-than-average similarity was observed in simple algorithms such as Binary Search, Bubble Sort, and Insertion

Algorithm	Diff Similarity(%)		Instruction Count Relative Error(%)		Memory Access Proportion Difference(%)	
	McSema	LLM	McSema	LLM	McSema	LLM
BFS	0.34	4.02	0.77	0.73	0.053	0.010
Binary Search	0.37	9.81	0.87	0.69	0.055	0.001
Bubble Sort	0.37	10.78	0.88	0.71	0.057	0.016
DFS	0.22	1.52	0.76	0.73	0.043	0.014
Dijkstra	0.32	4.50	0.77	0.70	0.052	0.050
Heap Sort	0.30	5.95	0.78	0.72	0.078	0.047
Insertion Sort	0.39	9.95	0.89	0.67	0.062	0.022
Linear Search	0.40	11.79	0.92	0.70	0.034	0.023
Merge Sort	0.32	3.95	0.81	0.69	0.062	0.074
Quick Sort	0.27	10.54	0.84	0.70	0.081	0.064
Selection Sort	0.38	9.54	0.88	0.72	0.067	0.015
Average	0.33	7.49	0.83	0.71	0.059	0.031

Table 5: Comparison of McSema and LLM in diff similarity, instruction count relative error, and memory access proportion difference. (LLM: average of 5 repetitions)

Sort. On the other hand, values below 5% were observed in recursive or graph-based algorithms such as DFS, BFS, Merge Sort, Dijkstra. This can be interpreted as resulting from the high diversity of expressions in implementation methods due to recursive or stack/queue based iterations, priority queues, arrays, and the use of various data structures. However, in the case of Quick Sort, despite being a recursive algorithm, it showed a relatively high similarity of 10.54 because the control flow of left and right recursive calls remains relatively consistent after pivot selection and partitioning. Additionally, in the Memory Access Proportion Difference analysis, Merge Sort showed the largest difference of 0.074, which can be explained by the fact that copy operations using auxiliary arrays are inherent in the algorithm's structure.

6 Conclusion

This study introduced an LLM-based lifting framework to generate IR that is both analyzable and executable. Using a system prompt derived from errors observed during the IR validation and execution process, the effective IR generation rate improved by about 13.2% compared to question-based prompts, and about 90% of the generated valid IRs produced the same execution results as the original. In addition, it showed higher structural similarity to the original IR compared to McSema, the existing lifting tool. However, this study was limited to evaluating small-scale binaries composed of up to two functions, and further research is required for binaries containing multiple functions or complex call graphs. In the future, we plan to expand the scope to environments with diverse architectures, optimization options, and obfuscation techniques.

References

[1] Yumin Choi, Jinheon Baek, and Sung Ju Hwang. System prompt optimization with meta-learning. *arXiv preprint arXiv:2505.09666*, 2025.

[2] Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[3] Chris Eagle. *The IDA pro book*. no starch press, 2011.

[4] Louie Giray. Prompt engineering with chatgpt: a guide for academic writers. *Annals of biomedical engineering*, 51(12):2629–2633, 2023.

[5] Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium*, volume 267622140, 2024.

[6] Hailong Jiang, Jianfeng Zhu, Yao Wan, Bo Fang, Hongyu Zhang, Ruoming Jin, and Qiang Guan. Can large language models understand intermediate representations in compilers? *arXiv preprint arXiv:2502.06854*, 2025.

[7] Jakub Křoustek, Peter Matula, and Petr Zemek. Retdec: An open-source machine-code decompiler. *July*, 2018.

[8] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[9] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024.

[10] Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. Lms: Understanding code syntax and semantics for code analysis. *arXiv preprint arXiv:2305.12138*, 2023.

[11] OpenAI. Introducing gpt-5, 2025. Accessed: 2025-11-10.

[12] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. *arXiv preprint arXiv:2403.05286*, 2024.

[13] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[14] Lechen Zhang, Tolga Ergen, Lajanugen Logeswaran, Moontae Lee, and David Jurgens. Sprig: Improving large language model performance by system prompt optimization. *arXiv preprint arXiv:2410.14826*, 2024.