C-ing Clearly: Enhanced Binary Code Explanations using C code

Anonymous ACL submission

Abstract

Large Language Models (LLMs) typically excel at coding tasks involving high-level programming languages, as opposed to lower-level programming languages, such as assembly. We propose a synthetic data generation method named C-ing Clearly, which leverages the corresponding C code to enhance an LLM's understanding of assembly. By fine-tuning on data generated through our method, we demonstrate improved LLM performance for binary code summarization and vulnerability detection. Our approach demonstrates consistent gains across different LLM families and model sizes.

1 Introduction

LLMs have shown strong performance in high-resource programming languages such as Python or C, while typically under-performing in low-resource languages (Orlanski et al., 2023; Cassano et al., 2022) such as assembly. Specializing LLMs in assembly language enhances binary analysis, a key capability for reverse engineering (Manuel et al., 2024) and malware detection (Alrabaee et al., 2022). These specialized models can be obtained by additional pretraining on unsupervised assembly data (Cummins et al., 2024) or through post-training with *{assembly code, report}* supervised data. We focus on the latter by generating better synthetic reports from general-purpose LLMs.

We propose **`C-ing Clearly`**, a method for generating synthetic datasets of *{assembly code, report}* pairs with LLMs, by leveraging the corresponding C source code. A high-level overview of our method can be seen in Figure 1. Using both assembly code and C code as inputs, the **`C-ing Clearly`** prompting strategy exploits the generator's knowledge of C code to improve binary code analysis. The C code acts as a high-resource proxy, effectively anchoring the models understanding of the assembly. Our approach is conceptually similar to the work of Zhang et al. 2024b, which use popular programming languages to improve the code generation abilities for less popular *high-level* programming languages. However, our work focuses *low-level* programming language understanding.

Binary Code Summarization (BCS) is the task of summarizing binary programs, usually based on the assembly code recovered from them. Most approaches in BCS target function-level summaries (Ye et al., 2023; Jin et al., 2023; Lu et al., 2024), usually by matching source code comments to the corresponding assembly code. However, due to their brevity and implementation-specific nature, source code comments may limit the models ability to generate thorough explanations. In contrast, our work focuses on producing more detailed binary code general analysis.

The general analysis reports created with **`C-ing Clearly`** can be further extended with task-specific analyses. To this end, we focus on the vulnerability detection (VD) task. We start from a dataset of *{C code, vulnerability labels}* pairs and convert them to *{assembly code, general analysis + vulnerability analysis}* pairs. The *vulnerability analysis* locates each *vulnerability label* within the assembly code and describes the associated security risks.

We further experiment with a multi-turn prompting strategy we refer to as **`Multi-turn C-ing Clearly`**, in which the LLM analyzes the C code first, then cross-references it with assembly and finally reuses previous completions to obtain the binary analysis.

Using our prompting strategies we generate synthetic data in the form of *{assembly code, report}* pairs and fine-tune **`Llama-3.1-8B-Instruct`**. We then evaluate the fine-tuned models' performance on the BCS and VD tasks.

Our contributions can be summarized as follows:

- We introduce **`C-ing Clearly`**, a synthetic data generation method that addresses the

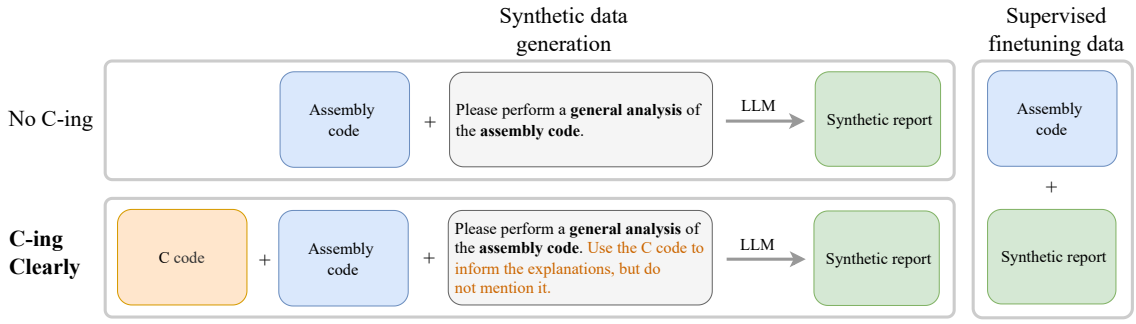

Figure 1: **`C-ing Clearly`** synthetic data generation versus direct assembly analysis (**`No C-ing`**): LLM generates synthetic reports of the assembly code by also using the C code. The *{assembly, report}* pairs are then used for supervised fine-tuning.

scarcity of in-depth binary code summaries. Our method leverages the LLM's better understanding of C code to obtain more informed assembly annotations.

- We show that training with **`C-ing Clearly`** improves the LLM's ability to generate assembly code summaries that accurately capture high-level program behavior.
- We demonstrate that training with **`C-ing Clearly`** also improves vulnerability detection performance, highlighting the potential of our method for other code-related applications. We confirm our findings on different model families as well (**`Qwen2.5-14B`**).

2 Method

In this section we introduce the prompting strategies used in **`C-ing Clearly`**, define the tasks along with their dataset construction, and provide model training details.

2.1 Prompting strategies

The goal is to generate high-quality completions from a pre-trained LLM M_{gen} for an assembly-related task. Given a compilable C program, we extract its corresponding assembly code and utilize M_{gen} along with a prompt template P.

The prompt template P contains two distinct components: the context c and the task-specific instruction I. We consider two data-generation setups: **`C-ing Clearly`** (the context contains the assembly code *and the original C code*) and **`No C-ing`** (the context is assembly code only). We prompt the LLM with $P(context, I)$, which results in a task-specific assembly output. In Figure 1, we illustrate a high-level overview of the two setups. The full set of prompts are listed in Tables 6-11 in the Supplementary.

We propose an additional strategy we refer to as **`Multi-turn C-ing Clearly`**. Instead of directly outputting the assembly report, we engage M_{gen} in a three-turn conversation. In the first turn, the LLM analyzes the C code. The assembly code is cross-referenced with the C code in the second turn. Finally, M_{gen} integrates observations from the previous two turns and outputs an assembly-only analysis. Both setups prevent mentions of the C code in the final output, but the final output of **`Multi-turn C-ing Clearly`** may draw on the C code explanations from the intermediate turns.

For both tasks, we use M_{gen}=LLama-3.1-Nemotron-70B-Instruct (Wang et al., 2024b) to generate the synthetic data.

2.2 Tasks

We use the synthetic data generated with the previously described prompting strategies to perform supervised fine-tuning. We focus on two tasks: Binary Code Summarization (BCS) and Vulnerability Detection (VD). Our goal in BCS is to obtain *general analysis* reports that split the code into assembly blocks, highlight their functionality, and conclude with an overview. For BCS, we fine-tune using *{task prefix + assembly, general analysis}* pairs. We hypothesize that retaining the *general analysis* in the output also benefits the VD task.

VD is typically a multi-label vulnerability classification task, performed over popular high-level programming languages such as C++ or Java (Sun et al., 2024). However, we are interested in enhancing the binary code analysis capabilities of an LLM, therefore we cast this as an **assembly** $\rightarrow$ **vulnerability labels** generative problem. The vulnerability labels are based on existing taxonomies

	Binary Code Summarization	Vulnerability Detection
task prefix	Describe the overall goal of this assembly code. What task does it perform?	Describe the overall goal of this assembly code. What task does it perform? Include any major security concerns if they're evident
assembly code	**Assembly code:** `push byte +0x1e` ...	**Assembly code:** `push byte +0x1e` ...
synthetic report	**General analysis**	**General analysis** **Vulnerability analysis**

Figure 2: Supervised fine-tuning data format for the two tasks

such as Common Weakness Enumeration (CWE) [1]. We fine-tune the model using *{task prefix + assembly, general analysis + vulnerability analysis}* pairs, where the vulnerability analysis is a report containing potential CWE labels, security implications and the relevant assembly code where they occur. In Figure 2 we depict the structure of the supervised fine-tuning data for both tasks.

The full set of prompts used to generate the training data is provided in both the code repository and the Supplementary for each method: **`C-ing Clearly`** (Tables 6-7), **`Multi-turn C-ing Clearly`** (Tables 8-9), and **`No C-ing`** (Tables 10-11). For *vulnerability analysis* we use slightly different prompts for vulnerable and non-vulnerable samples. For vulnerable samples we feed the ground truth CWEs and their description, while for non-vulnerable samples we specify that the code is most likely benign.

2.3 Supervised fine-tuning datasets

We generate the supervised fine-tuning data from both vulnerable and non-vulnerable code snippets. Specifically, we start from two datasets, DiverseVul (Chen et al., 2023) and VDISC (Russell et al., 2018). They both contain C/C++ functions, collected from open source projects, with vulnerability annotations obtained either through static analysis tools or by crawling security issues and mapping the corresponding commits to affected functions. DiverseVul contains 19K vulnerable functions spanning 150 CWEs.

After discarding samples with length above 16,384, we obtain ∼8.5K vulnerable samples. We keep a held-out test set of 500 samples and use the other samples for training. From the VDISC dataset, we use ∼9K non-vulnerable samples for training. For all the C samples, we generate the corresponding assembly code by running the `gcc -S source.c` command. We generate a supervised fine-tuning dataset for each of the two tasks, as described in subsection 2.2. Each dataset contains examples from both DiverseVul and VDISC.

2.4 Models and training details

We note that training with **`C-ing Clearly`** data is model agnostic and benefits any base model that has sufficient knowledge of C, but may lack assembly knowledge. We choose **`Llama-3.1-8B-Instruct`** (Grattafiori et al., 2024) as the main base model for our experiments, which we fully fine-tune on each dataset resulted from the **`No C-ing`**, **`C-ing Clearly`** and **`Multi-turn C-ing Clearly`** data generation methods. We fix the training budget to 1B tokens for every experiment.

We also use **`Llama-3.1-8B-Instruct`** as the baseline in a zero-shot setting. To generate reports that are similar to our fine-tuned models' outputs, we use the prompt in Table 12 in the Supplementary. The quality of these reports provides insight into the models initial assembly capabilities.

For model training we set the maximum sequence length to 16,384 and discard examples where the combined length of the prompt and completion exceeds this limit. On average, the assembly code in the supervised datasets is approximately 2,530 tokens long. We set the global batch size to 160. The optimizer is AdamW with a base learning rate of 3e-4. We use linear warmup with cosine annealing, with an initial learning rate of 1e-5 and warmup ratio of 4%. We perform all the experi-

[1] https://cwe.mitre.org

vs.	Llama	NOC	CC	MCC	avg. winrate
`Llama-3.1-8B-Instruct` (Llama)	-	16.45	13.30	21.05	16.93
`No C-ing` (NOC)	82.60	-	41.15	52.40	58.71
`C-ing Clearly` (CC)	**85.95**	**57.09**	-	**57.45**	**66.80**
`Multi-turn C-ing Clearly` (MCC)	78.40	46.65	40.15	-	55.07

Table 1: Win rate matrix for all pairwise comparisons between binary code summaries. `C-ing Clearly` outputs are consistently preferred by the GPT4-o judge.

Model	Output tokens	Number of sections	Avg. win rate
`Llama-3.1-8B-Instruct`	1034.7	3.54	16.93
`No C-ing`	1074.0	3.87	58.71
`C-ing Clearly`	873.0	3.33	**66.80**
`Multi-turn C-ing Clearly`	894.4	2.94	55.07

Table 2: Average length statistics and winrates for binary code summaries generated from different models

ments on one NVIDIA 8xA100-80 node.

3 Binary code summarization

In this section, we evaluate how well `Llama-3.1-8B-Instruct` and the fine-tuned models can analyze and explain the assembly code.

3.1 Evaluation

To evaluate report quality, we employ LLM-as-a-judge (Zheng et al., 2023). More specifically, we ask GPT-4o (Hurst et al., 2024) to select the best out of two summaries (each from a different model), based on how accurately it captures the main purpose, high-level functionality and logic of the assembly code. The evaluation prompt is illustrated in Table 13 in the Supplementary. We perform pairwise comparisons between all models.

For each of the 500 test samples, we generate two different outputs from each BCS model, using nucleus sampling (Holtzman et al., 2019) and a temperature of 0.4. This results in a total of 2000 comparisons for each pair of models. The order of the two reports in the evaluation prompt is random.

3.2 Results

The win rate matrix is reported in Table 1. The model trained with `C-ing Clearly` data is preferred over all other models, achieving the highest average win rate. Most importantly, `C-ing Clearly` outputs surpass those from the `No C-ing` model, showcasing the importance of the C code anchor in the synthetic data generation process.

Outputs from the `Multi-turn C-ing Clearly` model, although frequently preferred over the baseline outputs, are less competitive. Despite its sub-optimal BCS performance, the multi-turn setup has other advantages, discussed in section 4.4.

To better understand the summary structure and check for possible LLM judge biases, we analyze in Table 2 the average response statistics for the different models in relation to their average winrates. `C-ing Clearly` outputs are both the highest ranked and the shortest on average. In contrast, while `Multi-turn C-ing Clearly` produces similarly short summaries, the judge ranks them second to last. This suggests that length is not a primary factor in the judge's decisions.

3.3 Qualitative analysis

We investigate the answers from the LLM judge to gain insights into the differences between outputs. A sample of the `C-ing Clearly` output is shown in Table 14 in the Supplementary. In Figure 3 we notice that the judge prefers the `C-ing Clearly` output, highlighting aspects such as core functionality, accuracy and conciseness. The `No C-ing` output on the other hand is comprehensive but doesn't capture high-level insights as effectively.

We next examine several key phrases that frequently appear in the judge's responses and are invoked when ranking the outputs. For each phrase, we count how often it occurs when the winning report is produced by either `C-ing Clearly` or `No C-ing`. These counts are then normalized by the total number of occurrences of each phrase.

We notice in Figure 4 that phrases linked to high-level functionality ('core functionality', 'main purpose', 'more focused') and correctness ('more correctly') are more often linked to cases where `C-ing Clearly` is ranked first. Notably, when 'core functionality' is mentioned, `C-ing Clearly` outputs

Judge output: report 1 (No C-ing) vs report 2 (C-ing Clearly)

The better report is 2. Reasoning:
Focus on Main Functionality:
- Report 2 focuses more directly on the *main functionality* of the assembly code, which is the implementation of a function `_Z15tls1_alert_codei` that utilizes a jump table based on an input value to return specific codes.
- Report 1, while comprehensive, includes additional details about file and sections, variable and other unrelated functions, which can detract from the main analysis.
Clarity and Conciseness:
- Report 2 describes the function prologue, conditional branching, jump table usage, and function epilogue *clearly and concisely*.
- Report 1, although thorough, provides a lot of extra information that might not be directly relevant to understanding the primary logic and purpose of the function.
High-Level Insight:
- Report 2 accurately captures the *high-level insight* that the function maps input values to specific return values using a jump table.
- Report 1 includes additional context and background but doesn't emphasize the *core functionality* as effectively.
Therefore, based on these points, **Report 2** provides a *more accurate* and focused description of the main purpose, high-level functionality, and logic of the assembly code.

Figure 3: Excerpts from the output of the GPT4o judge: The model trained with **`C-ing Clearly`** data produces a binary code summary that better captures the main functionality of the code.

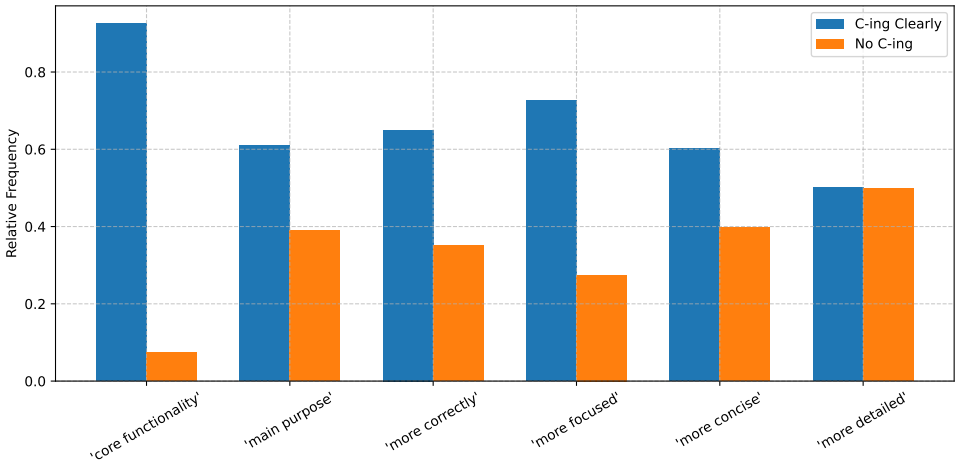

Figure 4: Relative occurrence count for different phrases in the winning report. Mentions of core functionality and correctness are more often linked with **`C-ing Clearly`** binary code summaries

are ranked first in 92% of the situations. Phrases linked to brevity are the least discriminative as a winning factor: when 'more detailed' occurs, both models are nearly equally likely to be ranked first.

4 Vulnerability detection

The BCS evaluation indicates improved general understanding of assembly. We also assess whether the datasets resulted from our methods can improve vulnerability detection capabilities of LLMs.

4.1 Results

We use greedy decoding to obtain outputs from models trained for the VD task. CWE label predictions are then extracted from the completions. Finally, we compute micro F1 scores on the test set using ground truth labels from DiverseVul.

We show the results in Table 3. While the **`Llama-3.1-8B-Instruct`** baseline has a relatively high

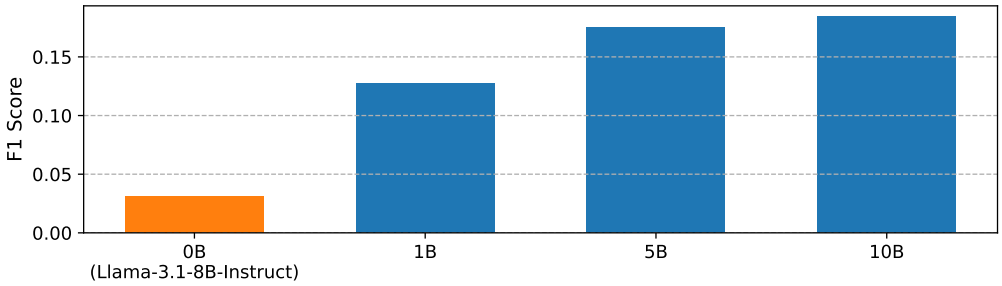

Figure 5: F1 score by token budget for **`Multi-turn C-ing Clearly`** for vulnerability detection

Model	Metrics		
	$F1$	P	R
`Llama-3.1-8B-Instruct`	3.08	1.78	11.46
`No C-ing`	4.90	11.76	3.09
`C-ing Clearly`	<u>9.89</u>	11.52	8.66
`Multi-turn C-ing Clearly`	**12.78**	13.39	12.23

Table 3: F1, Precision and Recall for vulnerability detection for Llama-3.1-8B-Instruct-based models. The best F1 results are shown in bold and second-best results are underlined.

Model	filtered	Metrics		
	train	$F1$	P	R
`Qwen2.5-14B-Instruct`	-	4.0	5.2	3.2
`No C-ing`	×	21.4	23.8	19.4
`C-ing Clearly`	×	18.5	25.3	14.6
`No C-ing`	✓	<u>21.8</u>	22.8	20.9
`C-ing Clearly`	✓	**24.0**	25.7	22.6

Table 4: F1, Precision and Recall for vulnerability detection for Qwen2.5-14B-based models. The best F1 results are shown in bold and second-best results are underlined.

Recall, suggesting some pre-existing CWE knowledge, it has the worst F1 score. Training with **`No C-ing`** shows little improvement over the baseline.

The model trained with **`Multi-turn C-ing Clearly`** data achieves the best F1, Precision and Recall, while the model trained on **`C-ing Clearly`** data obtains the second best F1 score. This suggests that our method benefits from multi-turn interactions, such that refining the assembly understanding in two steps leads to better downstream performance. However, for BCS, **`Multi-turn C-ing Clearly`** performs worse than **`C-ing Clearly`**, suggesting that **`C-ing Clearly`** strikes a more effective balance across both tasks.

4.2 Token budget analysis

Since fine-tuning on **`Multi-turn C-ing Clearly`** data yields the best performing model, we investigate the effect of scaling the fine-tuning token budget in this scenario. The evolution of the performance with a varying token budget is presented in Figure 5. In addition to the 1B result in Table 3, we compute F1 scores after training on 5B and 10B tokens of **`Multi-turn C-ing Clearly`** data. Increasing the token budget enhances detection performance, with a notable gain observed when scaling from 1B to 5B tokens.

4.3 Impact of data quality and model

In this section we address three questions: **i)** are the results an effect of model distillation? **ii)** do the results hold for a different model family than LLama? **iii)** does rejection sampling affect performance?

To answer **i)** and isolate the effect of the prompt design from the generator's capacity, we evaluate a setting where the base model is the same as the generator model M_{gen}. We set M_{gen} = **`Qwen2.5-14B-Instruct`**, which addresses question **ii)** as well.

The ground truth CWEs are provided to the generator model to produce the synthetic VD (assembly, report) data. To answer iii), we filter out reports that do not contain the corresponding CWEs. While completions from **`Llama-3.1-Nemotron-70B-Instruct`** have near-perfect match rates between CWEs in the generated analysis and ground truth CWEs (97-99% match), Qwen completions result in lower matches (58-59% match rate), requiring rejection sampling. We compare the performance of **`Qwen2.5-14B-Instruct`** models when fine-tuned on the original vs the filtered data.

The results are presented in Table 4. All the finetuned models are better than the zero-shot prompted M_{gen} model, which shows that **i)** the improvements are not a consequence of distilling from a larger model and **ii)** the results generalize

Embedding	Metrics		
	$F1$	P	R
Base Model	15.63 ± 2.73	21.03 ± 2.48	13.14 ± 3.81
No C-ing	16.53 ± 2.08	19.86 ± 1.41	14.43 ± 2.72
C-ing Clearly	**17.07** ± 1.54	17.81 ± 2.33	17.38 ± 4.36
MCC	**17.07** ± 3.04	20.84 ± 1.20	14.93 ± 3.30

Table 5: Results of multi-label classifiers trained over embeddings from different models. MCC stands for **Multi-turn C-ing Clearly**, while **Base Model** refers to Llama-3.1-8B-Instruct.

across different model families.

Regarding **iii)**, training on filtered synthetic data generally improves downstream performance. **No C-ing** outperforms **C-ing Clearly** when training on the raw data, but its performance shows modest improvement with rejection sampling (+0.4 F1). However, we observe that **C-ing Clearly** benefits more from rejection sampling, showing a larger F1 boost (+5.5) and outperforming **No C-ing**.

4.4 Multi-label classifier for VD

Since VD is a classification task, we can train multi-label classifiers on top of assembly-level features extracted from the input program. While not specifically finetuned for assembly representations, we can extract features from the **Llama-3.1-8B-Instruct**-based models evaluated in Section 4.1.

Specifically, we encode the assembly using the contextual embedding of size 4,096 corresponding to the last token in the *{task prefix + assembly}* instruction prompt. The label set spans 135 unique CWEs in the training data. The classifier is a two-hidden-layer MLP, for which we select the best performing detection threshold based on a held-out validation set. We extract features from both the baseline and fine-tuned models.

The classification results averaged over 20 independent training runs are presented in Table 5. As in the generative setup, the best performing models are those trained on **Multi-turn C-ing Clearly** and **C-ing Clearly** data, outperforming **Llama-3.1-8B-Instruct** and **No C-ing**. This suggests that fine-tuning on synthetic data enhanced with C anchors results in more discriminative assembly features for the VD task. While this is not the focus of our work, it indicates a potential direction for future research on optimizing assembly embeddings.

5 Related Work

LLMs for assembly are typically designed to learn meaningful assembly representations for tasks such as Binary Code Similarity Detection or to assist with binary analysis tasks such as Binary Code Decompilation or Binary Code Summarization.

5.1 Models for learning semantic representations of assembly

To learn assembly representations, both encoder and decoder Transformer-based models (Vaswani et al., 2017) are pretrained then fine-tuned contrastively with different signals. Encoders are trained with Masked Language Modeling (MLM) signal and specialized pretraining tasks such as Jump Target Prediction (Wang et al., 2022).

For instance, Wang et al. 2024a introduce CLAP, in which they supervise a RoBERTa encoder (Liu et al., 2019) with natural language explanations generated by an LLM from the original source code. ASMA-tune (Wang et al., 2025) learns a projector module to align the assembly encoder to the decoder's latent space, to benefit from both the encoder's representation learning and the decoder's explanation capability.

Jiang et al. 2023 add hierarchical self-attention heads to a pre-trained decoder and train it with both next token prediction loss and source code - assembly contrastive loss. This model then transfers well to both embedding tasks and binary analysis tasks such as Binary Decompilation.

5.2 Models for binary analysis

LLMs have been more recently used for assembly analysis tasks such as Binary Decompilation and Binary Code Summarization, finetuned from general LLMs (Grattafiori et al., 2024) or code LLMs (Guo et al., 2024). Efforts to build a foundational LLM for assembly have emerged, with Cummins et al. 2024 adapting a code LLM (Roziere et al., 2023) and further specializing it for downstream tasks such as compiler flag optimization and disassembly into LLVM-IR.

For Binary Decompilation, Tan et al. 2024 train end-to-end models called LLM4Decompile to produce the decompiled output based on the assembly input. They outperform classic decompilation tools such as Ghidra (National Security Agency, 2025) and can even further refine Ghidra's output.

Compared to Source Code Summarization, Binary Code Summarization suffers from a scarcity

of high-quality training data. To address this, recent papers have introduced large-scale datasets and explored the use of LLMs: BinSum (Jin et al., 2023), CAPYBARA (Al-Kaswan et al., 2023) and CP-BCS (Ye et al., 2023). These datasets typically provide function-level summaries by pairing source code comments with the corresponding assembly code. In contrast, Lu et al. 2024 construct MalS, a malware-focused BCS dataset where summaries are obtained with LLMs based on the malicious source code. Our work follows a similar direction and focuses on producing more in-depth binary code summaries via LLMs rather than relying on potentially terse and implementation-specific source code comments.

5.3 Vulnerability detection

Prior to LLMs, a common approach to identifying vulnerable code was to use features from a domain-tuned Transformer (Thapa et al., 2022).

Despite leveraging LLMs, recent VD approaches have not yet resulted in consistently high performance (Ding et al., 2024). Through in-context learning (Brown et al., 2020; Dong et al., 2022), LLMs can be adapted using auxiliary information in the prompt, such as data-flow (Zhang et al., 2024a) or static analysis tools (Li et al., 2023).

DLAP (Yang et al., 2025) employs Retrieval-Augmented-Generation (Liu et al., 2020) to integrate the most similar labeled code fragments with static analysis outputs. This retrieved information is further used to select a predefined Chain-of-Thought (Wei et al., 2022; Kojima et al., 2022) to guide the vulnerability analysis.

6 Conclusion

In this paper, we present a synthetic data generation method named **`C-ing Clearly`** to enhance binary code analysis for LLMs. By providing both assembly code and its corresponding C code, we leverage the LLM's in-context abilities to guide it towards better outputs.

Experiments show that training with **`C-ing Clearly`** improves binary code summaries, in terms of correctness, conciseness and understanding of high-level functionality. Moreover, our method improves the performance on the vulnerability detection task, highlighting its effectiveness across related code analysis problems. Finally, ablations on the vulnerability detection task demonstrate that **`C-ing Clearly`** is robust across different model families and can help models improve by bootstrapping on self-generated data.

7 Limitations and Future Work

Our method leverages high-level C code to support in-context learning of low-level machine representations. We acknowledge that other high-level languages may also aid in reasoning about low-level code such as assembly or bytecode.

Our experiments are based on assembly programs compiled using `gcc -S`, which applies no optimization. We hypothesize that our proposed method could be also beneficial when applied to optimized assembly code, where the C anchor may help interpret more compact representations.

Additionally, our results are demonstrated on the x86-64 instruction set, without covering other instruction sets such as ARM or RISC-V.

Finally, a practical limitation lies in the size of the assembly programs. These require substantial context capacity from the LLM. Due to resource constraints, we restricted our experiments to inputs with a maximum sequence length of 16K tokens. Future work should explore the scalability and robustness of our approach on longer binaries.

References

Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Prem Devanbu, and Arie van Deursen. 2023. Extending source code pre-trained language models to summarise decompiled binarie. *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 260–271.

Saed Alrabaee, Mourad Debbabi, and Lingyu Wang. 2022. A survey of binary code fingerprinting approaches: taxonomy, methodologies, and features. *ACM Computing Surveys (CSUR)*, 55(1):1–41.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, and 1 others. 2022. Multiple: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David A. Wagner. 2023. Diversevul: A

new vulnerable source code dataset for deep learning based vulnerability detection. *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, and 1 others. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, Xiangyu Zhang, and Petr Babkin. 2023. Nova: Generative language models for assembly code with hierarchical attention and contrastive learning.

Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. 2023. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. *ArXiv*, abs/2312.09601.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting static analysis with large language models: A chatgpt experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 2107–2111.

Shangqing Liu, Yu Chen, Xiaofei Xie, J. Siow, and Yang Liu. 2020. Retrieval-augmented generation for code summarization via hybrid gnn.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Haolang Lu, Hongrui Peng, Guoshun Nan, Jiaoyang Cui, Cheng Wang, and Weifei Jin. 2024. Malsight: Exploring malicious source code and benign pseudocode for iterative binary malware summarization. *ArXiv*, abs/2406.18379.

Dylan Manuel, Nafis Tanveer Islam, Joseph Khoury, Ana Nunez, Elias Bou-Harb, and Peyman Najafirad. 2024. Enhancing reverse engineering: Investigating and benchmarking large language models for vulnerability analysis in decompiled binaries. *arXiv preprint arXiv:2411.04981*.

National Security Agency. 2025. Ghidra software reverse engineering framework. `https://github.com/NationalSecurityAgency/ghidra`. Accessed: 2025-03-26.

Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. 2023. Measuring the impact of programming language distribution. In *International Conference on Machine Learning*, pages 26619–26645. PMLR.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Rebecca L Russell, Louis Kim, Lei H Hamilton, Tomo Lazovich, Jacob A Harer, Onur Ozdemir, Paul M Ellingwood, and Marc W McConley. 2018. Automated vulnerability detection in source code using deep representation learning. *arXiv preprint arXiv:1807.04320*.

Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. 2024. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *ArXiv*, abs/2401.16185.

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

Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-based language models for software vulnerability detection. In *Proceedings*

of the 38th Annual Computer Security Applications Conference, pages 481–496.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. 2024a. Clap: Learning transferable binary code representations with natural language supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 503–515.

Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13.

Xinyi Wang, Jiashui Wang, Peng Chen, Jinbo Su, Yanming Liu, Long Liu, Yangdong Wang, Qiyuan Chen, Kai Yun, and Chunfu Jia. 2025. Asma-tune: Unlocking llms' assembly code comprehension via structural-semantic instruction tuning. *arXiv preprint arXiv:2503.11617*.

Zhilin Wang, Alexander Bukharin, Olivier Delalleau, Daniel Egert, Gerald Shen, Jiaqi Zeng, Oleksii Kuchaiev, and Yi Dong. 2024b. Helpsteer2-preference: Complementing ratings with preferences. *Preprint*, arXiv:2410.01257.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Yanjing Yang, Xin Zhou, Runfeng Mao, Jinwei Xu, Lanxin Yang, Yu Zhang, Haifeng Shen, and He Zhang. 2025. Dlap: A deep learning augmented large language model prompting framework for software vulnerability detection. *Journal of Systems and Software*, 219:112234.

Tong Ye, Lingfei Wu, Tengfei Ma, Xuhong Zhang, Yangkai Du, Peiyu Liu, Shouling Ji, and Wenhai Wang. 2023. Cp-bcs: Binary code summarization guided by control flow graph and pseudo code. In *Conference on Empirical Methods in Natural Language Processing*.

Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2024a. Prompt-enhanced software vulnerability detection using chatgpt. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 276–277.

Jipeng Zhang, Jianshu Zhang, Yuanzhe Li, Renjie Pi, Rui Pan, Runtao Liu, Ziqiang Zheng, and Tong Zhang. 2024b. Bridge-coder: Unlocking llms' potential to overcome language gaps in low-resource code. *ArXiv*, abs/2410.18957.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and 1 others. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.

A Supplementary

C-ing Clearly: single turn (non-vulnerable)

Assembly Code Analysis Guide

C code
```c
{c_code}
```

ASM code
```asm
{assembly_code}
```

Purpose
- Provide a **high-quality, detailed analysis of the given assembly code**, focusing on the **relationship between C constructs and their compiled assembly** forms.
- Importantly, you can use the C code to inform your explanations for the assembly code, but do not mention the C code in your explanation.
- Assume that the recipient of your output can only see the assembly code, but if understanding the assembly code is tricky, you can look at the C code to get a better idea of what it's doing.
- Ensure the explanation is tailored to experts in assembly analysis.

Key Considerations
- Account for compiler optimizations and transformations from high-level code to assembly.
- Recognize how compile-time optimizations impact the generated assembly code.
- Include relevant assembly snippets with detailed inline comments for clarity.
- Highlight common patterns and standard practices in assembly.

Analysis Structure
Start with a general analysis, focusing on detailed and clear explanations for the reader. You will follow it up with a vulnerability-specific analysis, which will probably be empty.

General Analysis
- Divide the code into logical sections, numbering each (e.g., Section 1).
- Provide a very detailed analysis of each section, deep dive into details
- Do not focus on security issues yet.
- For each section, include 1015 lines of assembly code with detailed inline comments for key instructions.

General Analysis Conclusion
- Give an overview of what the code is trying to concretely achieve.

Vulnerability-Specific Analysis
It is unlikely the code contains vulnerabilities. You should mention the lack of vulnerabilities in this section. Do not add any additional section headers here.

Table 6: The prompt for generating the general analysis and the vulnerability analysis in a single turn, for non-vulnerable samples

C-ing Clearly: single turn (vulnerable samples)

Assembly Code Analysis Guide

C code
```c
{c_code}
```

ASM code
```asm
{assembly_code}
```

Purpose
- Provide a **high-quality, detailed analysis of the given assembly code**, focusing on the **relationship between C constructs and their compiled assembly** forms.
- Importantly, you can use the C code to inform your explanations for the assembly code, but do not mention the C code in your explanation.
- Assume that the recipient of your output can only see the assembly code, but if understanding the assembly code is tricky, you can look at the C code to get a better idea of what it's doing.
- Ensure the explanation is tailored to experts in assembly analysis.

Key Considerations
- Account for compiler optimizations and transformations from high-level code to assembly.
- Recognize how compile-time optimizations impact the generated assembly code.
- Include relevant assembly snippets with detailed inline comments for clarity.
- Highlight common patterns and standard practices in assembly.

Analysis Structure
Start with a general analysis, focusing on detailed and clear explanations for the reader. You will follow it up with a vulnerability-specific analysis, which will probably be empty.

General Analysis
- Divide the code into logical sections, numbering each (e.g., Section 1).
- Provide a very detailed analysis of each section, deep dive into details
- Do not focus on security issues yet.
- For each section, include 1015 lines of assembly code with detailed inline comments for key instructions.
General Analysis Conclusion
- Give an overview of what the code is trying to concretely achieve.

Vulnerability-Specific Analysis
Focus on the following vulnerabilities:
{CWEs_and_their_description}
For each unique vulnerability:
1. **Description**
- Explain in depth very precisely the vulnerability in assembly.
- Append the relevant assembly chunk near the explanation.
- Reference CWEs as needed and assess security risks.
2. **Exploitation**
- Outline potential attack vectors, severity, and provide at least one example exploit scenario.
3. **Mitigations**
- Suggest security improvements, including compiler flags or write secure assembly code.

Table 7: The prompt for generating summary and CWE tags in a single turn for vulnerable samples

C-ing Clearly: multi-turn (non-vulnerable samples)

Analyze the following C code from a functional perspective, focusing on **high level functionality** and system engineerings aspects:
“‘
{c_code}
“‘
Please follow these steps in your analysis:
Identify the **main purpose and functionality of the code**. Break down the code into logical sections or functions.
For each section:
a. Explain its purpose
b. Note potential system-level interactions
Summarize the overall implications of this code from a systems engineering standpoint.
Discuss any assumptions made in the code that could impact security or performance.
It's unlikely this code contains any major security issues, but unless there are any obvious issues, mention the lack of vulnerabilities in your explanation.
Summarize the overall implications of this code from a security and systems engineering standpoint.
Provide a detailed explanation for each step, ensuring a comprehensive understanding of the code's functionality.

Now, **examine the assembly code produced by the C code you just analyzed**:
“‘
{asm_code}
“‘
Please follow these steps in your analysis:
Identify the **main sections of the assembly code and their correspondence to the C code functions**.
For each significant block of assembly:
a. Explain its purpose and functionality
b. Relate it back to the corresponding C code
c. Add line-by-line comments explaining the assembly instructions
Highlight any compiler optimizations or unexpected translations from C to assembly.
Identify key assembly blocks where specific features or functionalities from the C code are implemented.
Discuss the operations regarding memory manipulation.
Summarize the key insights gained from examining the assembly version of the code.
Provide detailed explanations and comments throughout your analysis, ensuring a **clear understanding of the relationship between the C code and its assembly representation**.

TURN3 # Assembly Code Security Analysis Guide
Purpose
- Perform an analysis of assembly code, **based on prior C code analysis**, **focusing on the relationship between C constructs and their compiled assembly** forms.
Key Considerations
- !! Assume the receiver of your output cannot see the previous messages before the TURN3 text and therefore write complete explanations
Analysis Structure
Start with a general analysis, focusing on detailed and clear explanations for the reader. You will follow it up with a vulnerability-specific analysis.
General Analysis
- Divide the code into logical sections, numbering each (e.g., Section 1).
- Provide a very detailed analysis of each section, deep dive into details. Use your previous analysis to inspire your text, but do not reference C code.
- Do not focus on security issues yet.
- For each section, include 1015 lines of assembly code with detailed inline comments for key instructions.
General Analysis Conclusion
- Give an overview of what the code is trying to concretely achieve.
Vulnerability-Specific Analysis
It is unlikely the code contains vulnerabilities. You should mention the lack of vulnerabilities in this section. Do not add any additional section headers here.

Table 8: The multi-turn prompt for generating summary and CWE tags for non-vulnerable samples

C-ing Clearly: multi-turn (vulnerable samples)

Analyze the following C code from a functional perspective, focusing on **high level functionality** and system engineerings aspects: “‘{c_code}“‘
Please follow these steps in your analysis:
Identify the **main purpose and functionality of the code**. Break down the code into logical sections or functions.
For each section: a. Explain its purpose b. Note potential system-level interactions
Summarize the overall implications of this code from a systems engineering standpoint. Discuss any assumptions made in the code that could impact security or performance.
The code contains the following CWEs:
{CWEs_and_their_descriptions}
Knowing that, mention each vulnerable code chunk, and write a detailed step-by-step analysis of the code and vulnerability. A code chunk should containt at least 3 lines of code, and surround it with markdown syntax.
For each CWE, write: a. A short description of why the code has that vulnerability and the CWE name. Preface this the text "Vulnerability description:" b. How the CWE might be exploited and how dangerous it would be in practice. Preface this with the text "Possible ways to exploit:" c. Possible ways to solve this issue. Preface this with the text "Mitigations:". Highlight any potential security concerns or optimizations from a systems perspective.
Discuss any assumptions made in the code that could impact security or performance. Summarize the overall implications of this code from a security and systems engineering standpoint.
Provide a detailed explanation for each step, ensuring a comprehensive understanding of the code’s functionality and potential security implications.

Now, **examine the assembly code produced by the C code you just analyzed**:
“‘{asm_code}“‘
Please follow these steps in your analysis:
Identify the **main sections of the assembly code and their correspondence to the C code functions**.
For each significant block of assembly: a. Explain its purpose and functionality b. Relate it back to the corresponding C code c. Add line-by-line comments explaining the assembly instructions
Highlight any compiler optimizations or unexpected translations from C to assembly.
Identify key assembly blocks where specific features or functionalities from the C code are implemented. Discuss the operations regarding memory manipulation. Summarize the key insights gained from examining the assembly version of the code.
Provide detailed explanations and comments throughout your analysis, ensuring a **clear understanding of the relationship between the C code and its assembly representation**.

TURN3 # Assembly Code Security Analysis Guide
Purpose
- Perform an analysis of assembly code, **based on prior C code analysis**, **focusing on the relationship between C constructs and their compiled assembly** forms.
Key Considerations
- !! Assume the receiver of your output cannot see the previous messages before the TURN3 text and therefore write complete explanations
Analysis Structure
Start with a general analysis, focusing on detailed and clear explanations for the reader. You will follow it up with a vulnerability-specific analysis.
General Analysis
- Divide the code into logical sections, numbering each (e.g., Section 1).
- Provide a very detailed analysis of each section, deep dive into details. Use your previous analysis to inspire your text, but do not reference C code.
- Do not focus on security issues yet.
- For each section, include 1015 lines of assembly code with detailed inline comments for key instructions.
General Analysis Conclusion
- Give an overview of what the code is trying to concretely achieve.
Vulnerability-Specific Analysis
Focus on the following vulnerabilities:
{CWEs_and_their_description}
For each unique vulnerability:
1. **Description**
- Explain in depth very precisely the vulnerability in assembly.
- Append the relevant assembly chunk near the explanation.
- Reference CWEs as needed and assess security risks.
2. **Exploitation**
- Outline potential attack vectors, severity, and provide at least one example exploit scenario.
3. **Mitigations**
- Suggest security improvements, including compiler flags or write secure assembly code.

Table 9: The multi-turn prompt for vulnerable samples

no C-ing Clearly (non-vulnerable samples)

Assembly Code Security Analysis Guide
ASM code
‘‘‘asm
{asm_code}
‘‘‘
Purpose
- Perform a security analysis of assembly code, identifying vulnerabilities (CWEs), focusing on the compiled assembly sources.
- Write very high quality explanations of each section in the general analysis part.
Key Considerations
- Map CWE vulnerabilities to assembly.
- Account for compiler optimizations and C-to-assembly transformations.
- Recognize compile-time optimizations that may appear as missing checks.
- Include relevant assembly snippets with each vulnerability.
- Group similar vulnerabilities and avoid redundancy.
- Standard optimizations may omit some runtime checks.
- Fixed-size allocations may lack bounds checks; differentiate optimizations from true vulnerabilities.
Analysis Structure
Start with a general analysis, focusing on detailed and clear explanations for the reader. You will follow it up with a vulnerability-specific analysis.
General Analysis
- Divide the code into logical sections, numbering each (e.g., Section 1).
- Provide a very detailed analysis of each section, deep dive into details
- Do not focus on security issues yet.
- For each section, include 1015 lines of assembly code with detailed inline comments for key instructions.
General Analysis Conclusion
- Give an overview of what the code is trying to concretely achieve.
Vulnerability-Specific Analysis
It is unlikely the code contains vulnerabilities. You should mention the lack of vulnerabilities in this section. Do not add any additional section headers here.

Table 10: The prompt for generating summary and CWE tags from assembly code only for non-vulnerable samples

no C-ing Clearly (vulnerable samples)

Assembly Code Security Analysis Guide
ASM code
‘‘‘asm
{asm_code}
‘‘‘
Purpose
- Perform a security analysis of assembly code, identifying vulnerabilities (CWEs), focusing on the compiled assembly sources.
- Write very high quality explanations of each section in the general analysis part.
Key Considerations
- Map CWE vulnerabilities to assembly.
- Account for compiler optimizations and C-to-assembly transformations.
- Recognize compile-time optimizations that may appear as missing checks.
- Include relevant assembly snippets with each vulnerability.
- Group similar vulnerabilities and avoid redundancy.
- Standard optimizations may omit some runtime checks.
- Fixed-size allocations may lack bounds checks; differentiate optimizations from true vulnerabilities.
Analysis Structure
Start with a general analysis, focusing on detailed and clear explanations for the reader. You will follow it up with a vulnerability-specific analysis.
General Analysis
- Divide the code into logical sections, numbering each (e.g., Section 1).
- Provide a very detailed analysis of each section, deep dive into details
- Do not focus on security issues yet.
- For each section, include 1015 lines of assembly code with detailed inline comments for key instructions.
General Analysis Conclusion
- Give an overview of what the code is trying to concretely achieve.
Vulnerability-Specific Analysis
Focus on the following vulnerabilities:
{CWEs_and_their_description}
For each unique vulnerability:
1. **Description**
- Explain in depth very precisely the vulnerability in assembly.
- Append the relevant assembly chunk near the explanation.
- Reference CWEs as needed and assess security risks.
2. **Exploitation**
- Outline potential attack vectors, severity, and provide at least one example exploit scenario.
3. **Mitigations**
- Suggest security improvements, including compiler flags or write secure assembly code.

Table 11: The prompt for generating summary and CWE tags from assembly code only for vulnerable samples

Zero-shot `Llama-3.1-8B-Instruct` prompt

Assembly Code Security Analysis Guide
ASM code ‘‘‘asm
{asm_code}
‘‘‘
Purpose
- Perform a security analysis of assembly code, identifying vulnerabilities (CWEs), focusing on the compiled assembly sources.
- Write very high quality explanations of each section in the general analysis part.
Key Considerations
- Map CWE vulnerabilities to assembly.
- Account for compiler optimizations and C-to-assembly transformations.
- Recognize compile-time optimizations that may appear as missing checks.
- Include relevant assembly snippets with each vulnerability.
- Group similar vulnerabilities and avoid redundancy.
- Standard optimizations may omit some runtime checks.
- Fixed-size allocations may lack bounds checks; differentiate optimizations from true vulnerabilities.
Analysis Structure
Start with a general analysis, focusing on detailed and clear explanations for the reader.
You will follow it up with a vulnerability-specific analysis.
General Analysis
- Divide the code into logical sections, numbering each (e.g., Section 1).
- Provide a very detailed analysis of each section, deep dive into details
- Do not focus on security issues yet.
- For each section, include 10-15 lines of assembly code with detailed inline comments for key instructions.
General Analysis Conclusion
- Give an overview of what the code is trying to concretely achieve.
Vulnerability-Specific Analysis
For each unique vulnerability:
1. **Description**
- Explain in depth very precisely the vulnerability in assembly.
- Append the relevant assembly chunk near the explanation.
- Reference CWEs as needed and assess security risks.
2. **Exploitation**
- Outline potential attack vectors, severity, and provide at least one example exploit scenario.
3. **Mitigations** - Suggest security improvements, including compiler flags or write secure assembly code.

Table 12: Zero-shot prompt for the `Llama-3.1-8B-Instruct` baseline.

LLM-as-a-judge: evaluation prompt

Here is a piece of assembly code:
‘‘‘
{asm_code}
‘‘‘
The corresponding C/C++ code is:
‘‘‘
{c_code}
‘‘‘
I will give you two reports by two differents systems.
Your task is to choose which of the reports (1 or 2) more correctly describes the main purpose, high-level functionality and logic of the assembly code.
Report 1:
{report1}
Report 2:
{report1}
The experts had no access to the C code so they do not reference it. However, you may use it to better understand what the assembly program does, and use the insights in your judgement.
Please finish your answer with ’The better report is x’, where x is either 1 or 2, depending on your final choice, or ’none’, if you consider the reports equivalent.

Table 13: The prompt for BCS evaluation. The judge is GPT-4o (Hurst et al., 2024)

C-ing Clearly output for Binary Code Summarization

General Analysis
Section 1: Function Prologue and Setup

```
_Z15tls1_alert_codei:
.LFB1464:
    .cfi_startproc
    endbr64
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq %rsp, %rbp
    .cfi_def_cfa_register 6
    movl %edi, -4(%rbp)
```

- **Function Entry**: The `_Z15tls1_alert_codei` function begins with a prologue, setting up the stack frame and saving the base pointer (`rbp`).
- **Parameter Storage**: The first parameter (`%edi`) is stored at `-4(%rbp)`, indicating a 32-bit integer parameter.

Section 2: Conditional Branching and Jump Table

```
cmpl $115, -4(%rbp)
ja .L2
movl -4(%rbp), %eax
leaq 0(,%rax,4), %rdx
leaq .L4(%rip), %rax
movl (%rdx,%rax), %eax
cltq
leaq .L4(%rip), %rdx
addq %rdx, %rax
notrack jmp *%rax
```

- **Comparison and Branching**: The code compares the input value (`-4(%rbp)`) with `$115`. If the value is above `$115`, it jumps to label `.L2`.
- **Jump Table Calculation**: For values less than or equal to `$115`, the code calculates an index into a jump table (`L4`) using the input value. The table is located at `.L4(%rip)`, and the index is computed as `0(,%rax,4)`.
- **Indirect Jump**: The calculated index is used to perform an indirect jump to the corresponding label in the table. The `notrack` instruction suggests this is a no-op or a placeholder, potentially indicating a compiler optimization or a specific instruction set architecture (ISA) requirement.

Section 3: Jump Table and Function Epilogue

```
.L3:
    movl $0, %eax
    jmp .L34
...
.L2:
    movl $-1, %eax
.L34:
    popq %rbp
    .cfi_def_cfa 7, 8
ret
```

- **Jump Table**: The table contains various labels, each corresponding to a specific return value. The table is defined in the read-only data section (`rodata`).

- **Function Epilogue**: The function concludes with an epilogue, restoring the base pointer and returning control to the caller.

General Analysis Conclusion
The analyzed assembly code appears to implement a function that maps input values to specific return values based on a predefined table. The function takes a single 32-bit integer parameter, compares it against a threshold value, and uses a jump table to determine the return value. The code seems to be generated by a compiler, likely GCC, and includes various optimizations and specific instructions for a particular CPU architecture.

Table 14: Output from a model trained with **`C-ing Clearly`** for Binary Code Summarization