Large Language Models for Software Reverse Engineering

Miguel Garcia
DAF-MIT AI Accelerator
Cambridge, MA

***Abstract*—The role of Software Reverse Engineering (SRE) has immensely skyrocketed over the past decade due to increases in the complexity of software written by bad actors in the cyberspace domain. One key element of SRE is that of *code explanation*. However, despite the wide range of tools available for SRE, the task still remains a time-intensive and complex endeavor. Software binaries are often stripped and obfuscated, removing key information necessary for binary analysis. Additionally, these binaries come in a wide variety of instruction set architectures, requiring reverse engineers to understand low-level assembly code for multiple architectures. Adding to the complexity of the problem is the fact that SRE is multidisciplinary and requires knowledge not only in low-level programming but also networking, full stack development, mathematics, and more. Due to the extremely specialized combination of skill-sets necessary to reverse engineer software, we propose the use of finetuned Large Language models in conjunction with a software analysis package *CFG2VEC* in order to generate step-by-step explanations of stripped binary code.**[1]

I. Introduction

SOFTWARE Reverse Engineering (SRE) plays an essential role in the characterization of unknown systems and the detection of vulnerabilities within binaries [1]. Substantial effort has been placed into the development of software analysis tools, such as Ghidra [2]. These tools are capable of statically extracting relevant information from software binaries, such as debugging information, symbols, control-flow graphs, and function graphs. One feature of particular interest is Ghidra's *Decompiler*, a tool that lifts binaries that have successfully been disassembled into an intermediate pseudo-code representation (*High Function PCode*, referred to in this paper as *decompilation*) that strongly resembles C code. While the combination of symbols, code graphs, disassembly, and decompilation provides reverse engineers with bountiful information, SRE still remains a time-consuming endeavor requiring years of relevant experience in low-level programming. Additionally, these analyzed binaries can come from a wide range of both instruction set architectures (i.e. x86, MIPS, ARM) and domains (i.e. networking, graphical user interface, mathematics), often requiring an extremely specialized combination of skill-sets to successfully reverse engineer. Finally, these software binaries are often obfuscated and stripped, leading to either missing or misleading function symbols appearing in the disassembly and decompilation, even for commonly-used library functions that have been statically linked. This complication often prevents reverse engineers from relying on higher-level symbolic information extracted from a binary.

Much work has been done on *Machine Learning* (ML) techniques for understanding symbol-agnostic graphical features of code through static analysis. Existing methods include extracting features from the *Control Flow Graph* of binaries [3], in which branching operations within binaries are used to create *Basic Blocks*. These blocks enable the creation of Graphical Neural Network (GNN) models via serving as nodes in a graphical representation of the code flow. Prior research utilizing hierarchical GNNs has demonstrated the ability to create an embedding that reconstructs high-level function information in a cross-architectural and platform-independent way through identifying similarities between known and unknown executables [3].

Despite the limitations with analysis guided solely by symbols stated above, accurate symbols from binaries provide an invaluable speed-up to the SRE process, especially when reverse engineering systems consisting of many specialized domains. Large coding datasets [4], [5], [6], [7] curated from online repositories have led to breakthroughs in the ability of *Large Language Models* (LLMs) to summarize [8], [9] and complete [10], [9], [8] code. These finetuning datasets often include source code and documentation from publicly-available libraries and have been implemented in many widely-used applications today [11]. In fact, work has already been done on extending Language Models trained on source code to give a short summary of decompilation outputs [12]. However, these generated summaries are limited to 256 tokens, and the dataset does not include step-by-step analysis of decompiled software. Prior research [13] indicates that the reasoning capabilities of Large Language Models improve with *Chain-of-Thought* (CoT) prompting, a technique where a model is given a prompt requesting for a reasoning task to be broken down into a series of steps. Inversely, this style of prompting has been found to worsen performance for Language Models with fewer than 10-billion parameters. However, finetuning on a small (less than 1000 datapoints) dataset curated with relevant CoT step-by-step examples has been found to improve their performance on a variety of

T	Model	Average	ARC	HellaSwag	MMLU	TruthfulQA
	meta-llama/Llama-2-70b-chat-hf	*66.8*	*64.59*	*85.88*	*63.91*	*52.8*
	Mistral-7B-OpenOrca	65.84	64.08	83.99	62.24	53.05
	posicube/Llama2-chat-AYB-13B	*65.79*	*63.4*	*84.79*	*59.34*	*55.67*
	OpenOrca-Platypus2-13B	64.56	62.88	83.19	59.5	52.69
	OpenOrcaxOpenChat-Preview2-13B	63.6	62.4	83.1	58.6	50.4
	stabilityai/StableBeluga-13B	62.9	62	82.3	57.7	49.6
	NousResearch/Nous-Hermes-Llama2-13b	62.5	61.3	83.3	55	50.4
	mistralai/Mistral-7B-v0.1	62.4	59.98	83.31	64.16	42.15
	AIDC-ai-business/Marcoroni-7B	60.1	58.11	80.08	51.36	50.85
	meta-llama/Llama-2-7b-hf	54.32	53.07	78.59	46.87	38.76
	Open-Orca/oo-phi-1_5	51.16	53.41	64.3	43.46	43.46
	huggingface/llama-7b	49.72	51.02	77.82	35.71	34.33

Fig. 1: Performance of baseline model *MistralOrca* across the *HuggingFace Leaderboard*.

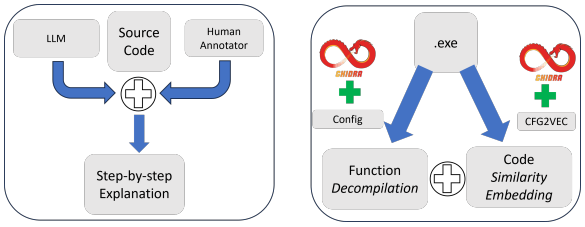

Fig. 2: Data generation pipeline utilizing LLMs to create step-by-step explanations of source code corresponding to decompilations generated by *Ghidra*.

metrics when given CoT prompts [14].

Unfortunately even with these improvements, LLMs still often suffer from *hallucinations* [15], a phenomenon where an LLM will offer an incorrect response with high-confidence. These hallucinations are misleading to users, since most complex tasks will not have a ground-truth label to verify outputs. In regards to Code LLMs, prior research has investigated methods in both adding reliability to LLM outputs [10] and in creating verification schemes using off-the-shelf GPT models [8] as judges for the code explanation task. A common technique for adding reliability to outputs is the use of *Retrieval Augmented Generation* (RAG), where content from vectorized documents are indexed by semantic similarity and appended to the prompt. RAG has been shown to improve LLM performance [16]. The primary limiting factor with existing RAG and verification methods for SRE is the fact that these methods require existing documentation of source code, and rely on semantic similarity from fully annotated code containing symbols for variable and function names.

In this paper, we investigate: (1) the creation of a custom finetuning CoT dataset, (2) the finetuning of LLMs on decompiled and partially-stripped binaries with CoT explanations to produce better step-by-step decompilation explanations, (3) the use of code structural similarity as an indexing scheme for retrieving function names and documentation on software binaries, and (4) the creation of an application assisting in the reverse engineering of stripped software binaries incorporating aspects of aforementioned topics.

II. Methodology

A. Model Selection and Finetuning

The base model selected was the publicly-available *Mistral-7B* model, finetuned with the *OpenOrca* dataset [7], [6]. This finetuned model, known as *MistralOrca*, was chosen due to its performance on the *HuggingFace Leaderboard* [17], where it outperformed all 7B and 13B models and reached 98.6% of *Llama2-70b-chat's* [18] average metric score, as shown in Fig 1. The base model and finetuning dataset were ideal for the creation of a SRE application as well, due to their less-restrictive Terms of Use and Apache 2.0 licenses [19]. A dataset consisting of decompiled code with stripped variable names, corresponding source code, and step-by-step explanations was created for this task. This dataset included: (1) code directly taken from a publicly available library in the ARM architecture for unmanned aerial vehicles, and (2) example code implementing library functions in the AMD64 architecture for computer networking. This second dataset was created in order to evaluate the model's performance in summarizing decompiled source code from a different domain and architecture. Additionally, the use of exemplar code not found directly in library source ensured that the model was not pretrained on documentation pertaining to that function.

1) Embedded Library Code Dataset: In order to prepare the model to create high-quality step-by-step decompilation explanations, *MistralOrca* was finetuned on a generated dataset of 100 code decompilations and corresponding CoT explanations. These datapoints consisted of (1) decompiled 32-bit ARM binary code from the *LibrePilot* project [20], an open-source library for model creation of unmanned aerial vehicles targetting the STM32 microcontroller, and (2) manually curated step-by-step English explanations of the disassembly functions. The initial dataset was split into a training set (80%) and validation set (20%). To expedite creation of code explanations, an off-the-shelf Mistral-7B model was used to generate step-by-step explanations of source code that corresponded to decompilations in the dataset, with length 512 tokens. Both the decompiled code and source code were referenced from the CAPYBARA [12] dataset. To offset any potential inaccuracies in source code explanations, a human annotator manually adjusted the output labels as needed.

2) Custom Code Dataset: In order to create a custom dataset for evaluating decompilation summarization across domains and architectures, a dataset generation pipeline, pictured above in Figure 2, was created. A *Docker* solution was created to allow for compilation across multiple architectures with different compiler settings. For initial testing, 50 C++ files containing hand-written functions utilizng the *Boost.Asio* library were gathered and filtered by a human for testing. After deduplication and compilation, 26 software binaries were generated and decompiled through *Ghidra*. An off-the-shelf Mistral-7B model was used to generate step-by-step explanations of source code that corresponded to decompilations in the dataset, with length 512 tokens. To offset any potential inaccuracies in source code explanations, a human annotator manually adjusted the output labels as needed.

3) Model Finetuning: Once the dataset was created, the model was finetuned for 1 epoch on a singular NVIDIA RTX4090 Ti. The finetuning technique selected was QLoRA

[21], [22], as it allows the finetuning of a quantized 4-bit model without any performance degradation, greatly reducing the compute power needed.

4) Evaluation: In order to assess the effectiveness of finetuning the model, the metric ROUGE (Recall-Oriented Understudy for Gisting Evaluation) was used. ROUGE is a standard metric [23] in natural language processing to evaluate the quality of machine-generated summaries. The metric evaluates the similarity of a summary to a reference through sequential token analysis, namely *n-grams*, a count of overlapping text within a single sequence. Both the base model and the finetuned model were tasked in summarizing 20 decompiled functions with stripped variable names. Generated summaries of the decompilation were then compared to summaries of the corresponding source code.

B. Retrieval Augmented Generation

A common technique for adding reliability to outputs is the use of *Retrieval Augmented Generation* (RAG), where ground-truth content from vectorized documents are indexed by semantic similarity and added to the prompt in an intelligent way. However, function names are often stripped or obfuscated in binaries, adding difficulty to the task of identifying functions from statically linked libraries due to reduced semantic similarity. Our methodology seeks to improve upon this in the code explanation domain. Rather than embedding the decompiled code through a sentence transformer and indexing by semantic similarity, we evaluate an embedding scheme that relies on the structural features of the compiled binaries.

CFG2VEC is a software package written in Python and Java that aims to create a hierarchical graph neural network for cross-architectural software reverse engineering. Binary functions are represented in a *Graph of Graph* model in which the control-flow and function-call graphs are embedded as features. By using these architecture-agnostic representations, *CFG2VEC* has demonstrated its capability to reconstruct function names and perform function matching across cross-compiled binaries [3].

In order to evaluate the effectiveness of *CFG2VEC* for the retrieval task, a test set was created consisting of 50 functions from a library cross-compiled into both AMD64 and ARM. A copy of this test set was then created, with function names completely stripped, in order to simulate real-world reverse engineering conditions. A lightweight *CFG2VEC* model was trained using a small dataset of 260 binaries provided in its GitHub repository. This *CFG2VEC* model and an off-the-shelf *all-MiniLM-L6-v2* model were evaluated against each other in their ability to match functions across architectures with graphical features and code decompilation, respectively. A successful "match" was considered when the solution function was listed in the model's top-k predicted matches. Both methods used cosine similarity as their indexing method.

III. Results

A. LLM Finetuning Performance

Table 1 shows the results of finetuning the base LLM on decompiled code. Step-by-step finetuning shows to substantially improve ROUGE scores across recall, precision, and F-Score. This was to be expected, as the finetuned model was able to predict class structures and variable naming schemes that it had encountered in training and finetuning. Inversely, the base model's code explanation outputs retained the temporary names given by the decompilation (i.e. instead of *server* it would show *local8*). However, one metric that could not be effectively measured was the impact of hallucinations on the output of the finetuned model. While the majority of variable names predicted by the finetuned LLM were correct, the model did show hallucinations in variable predictions. An intelligent metric for the impact of hallucinations on summary accuracy is needed, as a class *Server* being mislabeled as *AsynchronousServer* will still provide value to an analyst. Future work for improvements in this area include (1) using Shapley values to detect biases on symbols such as function names in the finetuning dataset, and (2) utilizing a retrieval scheme based off of structural similarity to generate code snippet descriptions on scale more granular than the function-level.

TABLE I: Base Model vs Finetuned Model Summarization

	ROUGE-N	Recall	Precision	F-Score
Base Model	ROUGE-1	0.327148	0.387597	0.349154
	ROUGE-2	0.125934	0.150812	0.133981
	ROUGE-L	0.311290	0.368640	0.332137
Finetuned Model	ROUGE-1	0.545182	0.490934	0.506353
	ROUGE-2	0.351794	0.301850	0.316971
	ROUGE-L	0.527760	0.474253	0.489710

Average ROUGE performance of base model and finetuned model across test dataset.

B. Structural Similarity Performance

Table 2 shows the results of function matching for both the *CFG2VEC* GNN embedding and the *all-MiniLM-L6-v2* embedding. For this experiment, k values of 1 and 5 were used as thresholds for when an embedding had successfully "matched" a function in the test set. Four series of tests were performed in order to evaluate model effectiveness in matching stripped and unstripped functions across multiple architectures. The first two tests compared each model's ability to match stripped and unstripped functions within the same architecture. As shown in Table 2, *CFG2VEC* greatly outperformed *all-MiniLM-L6-v2* by orders of magnitude in this function matching task. This was to be expected due to the reduced semantic similarity between stripped and unstripped functions, and the fact that the function call graphs between the two binaries were almost identical. The second two tests compared each model's ability to perform the difficult task of matching stripped and unstripped functions across different architectures. In these tests, both models performed poorly, with *CFG2VEC* only slightly outperforming the *all-MiniLM-L6-v2* model. Despite being created for cross-architectural code similarity, *CFG2VEC* was limited in its ability to match cross-architectural functions whose call graphs were not similar in structure. Surprisingly, both models did not see an improvement in performance with the introduction of relevant symbols. *CFG2VEC*'s performance did not increase with the introduction of function symbols

TABLE II: *CFG2VEC* vs *all-MiniLM-L6-v2* Match Success

	CFG2VEC		***all-MiniLM-L6-v2***	
	top1%	**top5%**	**top1%**	**top5%**
Symbol-Agnostic EXE	77%	89%	15%	35%
Symbol-Agnostic ELF	74%	99%	2%	12%
Cross-Compiled Stripped	16%	33%	7%	30%
Cross-Compiled Unstripped	16%	33%	8%	23%

as the model does not rely on symbol data. On the other hand, semantic similarity did not take advantage of a few cases where cross-compiled functions had seemingly similar names and lengths in the unstripped experiment. Further experimentation is needed on additional software domains to investigate this finding. One interesting item of note is the fact that semantic similarity performed worse when comparing stripped and unstripped ELF binaries, as opposed to when comparing cross-compiled stripped/unstripped functions. This most likely is due to the use of similar "modalities" in the cross-compiled case (ie, both sets of functions being stripped/unstripped), which can lead to an increase in semantic similarity in general.

C. Application Development

In order to take advantage of our findings in both finetuning and retrieval, a standalone application utilizing the Ghidra API was created. The developed application allows users to take any Ghidra decompilation and utilize either *CFG2VEC* or *all-MiniLM-L6-v2* to "fixup" any stripped symbols. Within the application, a suite of commonly-used libraries has been compiled with symbols, and placed into a database, along with relevant documentation. Additionally, users are able to upload their own binaries with supporting documentation as needed. A lightweight *CFG2VEC* model is provided along with the model in order to index decompilations with symbols.

Additionally, the application provides a text editor for users to manually adjust any variable names that they are confident about. Both a base and finetuned *MistralOrca* model are provided for users to run inference on decompiled code.

Future work for this application involves hosting all models on a larger standalone server to allow for Code LLMs with over 13B parameters. Additionally, this architectural change will allow users to upload corpora of text with source code examples for model finetuning.

IV. Conclusion

In conclusion, while finetuning LLMs has demonstrated improved results across the ROUGE metric in decompilation step-by-step explanation generation, further work is needed to reduce hallucinations of variable names. Techniques analyzing the attention patterns of the model must be implemented in order to create more relevant finetuning datasets that encourage the model to be less reliant on debugging symbols, such as function names. This continued development of relevant binary datasets with step-by-step explanation metadata is critical to the development of any application hoping to take advantage of the RAG schema proposed in this paper. Despite this, the use of symbol-agnostic and architecture-agnostic of static code, such as the *Control Flow Graph*, has demonstrated the ability to outperform semantic embeddings in cross-architectural stripped function matching tasks necessary for retrieval.

Acknowledgment

I am extremely grateful to Jarrod Manguiat for his assistance and expertise in this project, especially with the creation of the custom finetuning dataset. Additionally, I am extremely thankful to Dr. Ana Smith, Paul Gibby, and Ashok Kumar at MIT Lincoln Laboratory for volunteering their time to mentor me throughout my research. Finally, I would like to acknowledge the *Edison Grant* program, whose funding provided the hardware used to run all experimentation.

References

[1] M. L. Nelson, "A survey of reverse engineering and program comprehension," *arXiv preprint cs/0503068*, 2005.

[2] R. Kurtz *et al.*, "Ghidra," https://github.com/NationalSecurityAgency/ghidra, 2019.

[3] S.-Y. Yu *et al.*, "Cfg2vec: Hierarchical graph neural network for cross-architectural software reverse engineering." *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2023.

[4] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps. corr abs/2105.09938 (2021)," *NeurIPS Datasets and Benchmarks*, 2021.

[5] S. Longpre, L. Hou, T. Vu, A. Webson, H. W. Chung, Y. Tay, D. Zhou, Q. V. Le, B. Zoph, J. Wei, and A. Roberts, "The flan collection: Designing data and methods for effective instruction tuning," 2023.

[6] S. Mukherjee, A. Mitra, G. Jawahar, S. Agarwal, H. Palangi, and A. Awadallah, "Orca: Progressive learning from complex explanation traces of gpt-4," *arXiv preprint arXiv:2306.02707*, 2023.

[7] W. Lian, B. Goodson, E. Pentland, A. Cook, C. Vong, and "Teknium", "Openorca: An open dataset of gpt augmented flan reasoning traces," https://https://huggingface.co/Open-Orca/OpenOrca, 2023.

[8] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[9] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," *arXiv preprint arXiv:2305.02309*, 2023.

[10] S. Zhang, Z. Chen, Y. Shen, M. Ding, J. B. Tenenbaum, and C. Gan, "Planning with large language models for code generation," *arXiv preprint arXiv:2303.05510*, 2023.

[11] M. Wermelinger, "Using github copilot to solve simple programming problems," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 172–178.

[12] A. Al-Kaswan, T. Ahmed, M. Izadi, A. A. Sawant, P. Devanbu, and A. van Deursen, "Extending source code pre-trained language models to summarise decompiled binarie," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 260–271.

[13] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," *arXiv preprint arXiv:2210.03493*, 2022.

[14] L. C. Magister, J. Mallinson, J. Adamek, E. Malmi, and A. Severyn, "Teaching small language models to reason," *arXiv preprint arXiv:2212.08410*, 2022.

[15] V. Rawte, A. Sheth, and A. Das, "A survey of hallucination in large foundation models," *arXiv preprint arXiv:2309.05922*, 2023.

[16] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[17] W. Lian, B. Goodson, G. Wang, E. Pentland, A. Cook, C. Vong, and "Teknium", "Mistralorca: Mistral-7b model instruct-tuned on filtered openorcav1 gpt-4 dataset," https://huggingface.co/Open-Orca/Mistral-7B-OpenOrca, 2023.

[18] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.
[19] A. Sinclair, "License profile: Apache license, version 2.0," *IFOSS L. Rev.*, vol. 2, p. 107, 2010.
[20] A. Morale *et al.*, "Librepilot," https://github.com/librepilot/LibrePilot, 2015.
[21] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *arXiv preprint arXiv:2305.14314*, 2023.
[22] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Llm. int8 (): 8-bit matrix multiplication for transformers at scale," *arXiv preprint arXiv:2208.07339*, 2022.
[23] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," *Text summarization branches out*, 2004.