Large Language Models for Code Analysis: Do LLMs Really Do Their Job?

Chongzhou Fang[1], Ning Miao[1], Shaurya Srivastav[1], Jialin Liu[2], Ruoyu Zhang[1], Ruijie Fang[1], Asmita[1], Ryan Tsang[1], Najmeh Nazari[1], Han Wang[2], and Houman Homayoun[1]

[1]*University of California, Davis* [2]*Temple University*

Abstract

Large language models (LLMs) have demonstrated significant potential in the realm of natural language understanding and programming code processing tasks. Their capacity to comprehend and generate human-like code has spurred research into harnessing LLMs for code analysis purposes. However, the existing body of literature falls short in delivering a systematic evaluation and assessment of LLMs' effectiveness in code analysis, particularly in the context of obfuscated code.

This paper seeks to bridge this gap by offering a comprehensive evaluation of LLMs' capabilities in performing code analysis tasks. Additionally, it presents real-world case studies that employ LLMs for code analysis. Our findings indicate that LLMs can indeed serve as valuable tools for automating code analysis, albeit with certain limitations. Through meticulous exploration, this research contributes to a deeper understanding of the potential and constraints associated with utilizing LLMs in code analysis, paving the way for enhanced applications in this critical domain.

1 Introduction

The evolution of machine learning has brought major changes to a variety of fields in the past few decades [38,44,76,80]. In recent years, the emergence of large language models (LLMs) has revolutionized the field of natural language processing and brought great changes to many other areas. Ever since the access of ChatGPT [64] becomes publicly available in late 2022, there is a rocketing number of users as well as publicly-available services utilizing LLM APIs provided by OpenAI. These models have demonstrated their abilities to understand and process inputs including but not limited to natural languages, and have been involved in different types of tasks. A typical use case of LLMs is text generation, since it has the ability to produce high-quality natural language outputs [91]. This has caused a heated debate in academia and eventually resulted in major publishers announcing new regulations regarding the usage of generative AI [15,47]. Besides, LLMs have also manifested their extraordinary potential in tasks related to programming, including code generation [79] and code comprehension [88]. LLM-based assistant tools like Github Copilot [41] have received tremendous attention and have been widely employed by programmers all around the world. OpenAI has recently enabled more features in ChatGPT [65], making LLMs more versatile and capable of engaging in more real-world tasks.

Amidst all possibilities of LLMs, code analysis and obfuscation are tasks that are receiving more and more attention from computer scientists. Code analysis [21] is an important task in software engineering, including code quality assessment [72], vulnerabilities detection [84], etc. The huge size and complex architecture of modern software systems motivate the wide usage of automated code analysis, especially in security-related scenarios such as software reverse engineering [25]. In response, malicious parties have leveraged code obfuscation techniques to obscure source code and avoid being detected. In response, researchers have proposed automated code analysis tools like Abstract Syntax Tree (AST) [61] to analyze both non-obfuscated and obfuscated code. However, such tools are highly dependent on the pre-defined data types and not suitable for general software code analysis. Due to the great potential of LLMs in these areas, researchers have started using LLMs to perform related tasks [24,88]. However, as pointed out by a recent survey [20], there are few systematic analytical studies on this topic.

In this paper, we focus on evaluating the ability of LLMs to analyze input code samples, and test if LLMs can be utilized for defensive analysis tasks. We first construct a dataset containing code from real-world programs (on Github, etc.) and their obfuscated versions to feed to LLMs, then we analyze the outcomes and present our obtained key findings. We aim to answer two critical research questions through a systematic analysis:

RQ1: Do LLMs understand source code?

RQ2: Can LLMs comprehend obfuscated code or code with low readability?

After that, as real-world case studies, we showcase if publicly-available LLMs (e.g. ChatGPT) can be utilized for security-related tasks targeting real-world malicious software.

The contributions of this paper are listed as follows:

1. We construct two datasets consisting of code samples from popular programming languages and obfuscated samples from these languages for our evaluation;
2. We systematically evaluate the performance of publicly available state-of-the-art LLMs, including the most popularly used GPT and LLaMA families on our dataset, and present our findings;
3. We conduct case studies using LLMs to analyze real-world malicious software to showcase the capability and limitations of LLMs in these tasks.

To the best of our knowledge, this is the first paper that provides a systematic evaluation of the ability of LLMs to comprehend code and code with obfuscation.

The remainder of this paper is organized as follows. We first introduce related background knowledge in Section 2. Then we elaborate our experiment settings of our analysis in Section 3. Essential results and findings are presented in Section 4. In Section 5, we perform case studies and show how LLMs can be utilized to assist code analysis in practical cases. Then we briefly discuss future works in Section 6. Related works and a conclusion are provided in Section 7 and Section 8, respectively. Our online appendix is available at: https://github.com/aseec-lab/llms-for-code-analysis.

2 Background

In this part, we will provide an overview of the technical background of large language models, code analysis, and code obfuscation.

2.1 Large Language Models

Large language models (LLMs) are a groundbreaking innovation in the field of artificial intelligence, representing a significant milestone in natural language understanding and generation. Most LLMs are built on deep learning architectures, particularly transformer architectures and self-attention mechanisms [80], which are trained on massive datasets containing a diverse range of text from the internet, books, and Github. This extensive training enables them to grasp the intricacies of languages, including grammar, context, and semantics. Both companies and researchers have launched LLMs-driven products and pre-trained models, such as OpenAI GPT series [66, 74], Meta LLaMA [77], Standford Alpaca [75], etc. One of the critical features of the LLMs is their ability to perform a wide range of natural language understanding tasks, including text summarization, translation, sentiment analysis, question-answering, etc. Besides natural language understanding tasks, researchers also find that LLM models have significant performance across various domains, from security and code generation [29] to healthcare [28, 45, 90] and education [50]. These LLM-driven applications have the potential to automate analysis tasks and provide instant information and assistance [17]. Moreover, LLMs are being shown to have direct utility for the purposes of code explanation and summarization, which have immediate applications in education and industry. Leinonen et al. [52] have explored the use of LLMs in generating code explanations for the purposes of computer science (CS) education, finding that LLM-generated explanations are viewed as more accurate and comprehensible than student-written explanations. Ahmed et al. [16] leverage project-specific few-shot training to improve code summarization, demonstrating improvements across a variety of languages and projects. This approach significantly reduces the time programmers need to familiarize themselves with a codebase.

2.2 Code Analysis

Code analysis is a process in software engineering to examine source code, byte code, or binary code to ensure quality, reliability, and security. Code analysis has become a crucial practice to assist developers in identifying and addressing problems early in the software development life cycle. The increasingly large scale of modern software challenges motivates researchers to propose automated tools. Prior works have proposed to extract structural relationships, including inheritance, association, friend relationships, interface hierarchies, attributes, data types, etc., from source code [61] to build Abstract Syntax Tree (AST) for pattern-matching [89] to detect vulnerabilities [26] and malicious activities [55]. However, the approach is highly dependent on the pre-defined data types and can not be used for general code analysis. In response, some researchers propose to extract features from source code and leverage machine learning to build a vulnerability detection model [93].

2.3 Code Generation

Code generation has been one of the most popular applications of LLMs since the introduction of GPT-3 OpenAI's introduction of the Codex model [29]. Subsequent advances in this area have had major implications on how programming is now taught, practiced, and evaluated. Prather *et al.* [68] noted in their meta-analysis of LLM use in CS education that LLMs perform similar if not better than the average student on code writing tasks, and have since sought to incorporate such tools into CS curriculum [69] in anticipation of more widespread adoption. A number of studies have also been conducted on practical LLM-based code completion tools

like Github's Copilot [41], characterizing common interaction models [19] and experiences [22]. Such tools continue to be enhanced as well, with improvements made in code completion for repository-level projects [18, 53] and automated program repair [81, 83], expanding the scope of LLM-based generation tools. Notably, Bird et al. [22] have observed in their investigation that while Copilot can improve productivity and creativity for users, the tool can also be somewhat detrimental to security and programmer understanding, which is part of the motivation for our study.

2.4 Code Obfuscation

Code obfuscation refers to the process of leveraging transformations to make functionally equivalent programs that are difficult to understand, aiming to protect the intellectual property of developed software or hide malicious behaviors. Such transformations include encoding data, opaque predicates [85], flattening control flow [51], etc. For example, [54] proposed a Mixed Boolean-Arithmetic (MBA) expression to mix the bitwise operations (e.g., AND, OR, and NOT) and arithmetic operations (e.g., ADD and IMUL), thereby creating more difficulties for attackers to analyze programs. In modern software development, obfuscation has been used to protect critical code parts against reverse engineering [37]. However, the significant progress of LLMs challenges existing code obfuscation-based protection approaches, questioning whether existing obfuscation can still be effective against LLMs-based de-obfuscation and preserve sensitive information in developed software. Hence, it becomes urgent to evaluate the code analysis results of LLMs when code is obfuscated. In this work, we leverage an open-source JavaScript obfuscation tool [12] as well as a state-of-the-art obfuscator [70] to generate obfuscated code and investigate whether LLMs are able to understand their functionality. As presented in Listing 1 and Listing 2, a simple "Hello World" function written in JavaScript can be changed into an unintelligible one.

Listing 1: No obfuscation.

```
function hi() {
  console.log("Hello World!");
}
hi();
```

Listing 2: After obfuscation by [12].

```
function _0x1ec3(){var _0x3ed452=['259790KgLPlj','297688NTFutg','35ACWDkX','
    145716kEyGyf','18SFCPKB','1701952aKOEga','192jjwxUU','5LPjNwr','142417
    rtWDUq','Hello\x20World!','121610lhBPGW','2032200UghFpX','5nCOmEq','log'
    ];_0x1ec3=function(){return _0x3ed452;};return _0x1ec3();}(function(
    _0x22b342,_0x360ffb){var _0x5047be=_0xfb3c,_0x4c7c5c=_0x22b342();while
    (!![]){try{var _0x40c3be=parseInt(_0x5047be(0x90))/0x1*(-parseInt(
    _0x5047be(0x8e))/0x2)+-parseInt(_0x5047be(0x95))/0x3+parseInt(_0x5047be(0
    x97))/0x4*(parseInt(_0x5047be(0x99))/0x5)+parseInt(_0x5047be(0x8f))/0x6+
    parseInt(_0x5047be(0x94))/0x7*(parseInt(_0x5047be(0x93))/0x8)+parseInt(
    _0x5047be(0x96))/0x9*(-parseInt(_0x5047be(0x92))/0xa)+parseInt(_0x5047be
    (0x9a))/0xb*(-parseInt(_0x5047be(0x98))/0xc);if(_0x40c3be===_0x360ffb)
    break;else _0x4c7c5c['push'](_0x4c7c5c['shift']());}catch(_0x33f4b4){
    _0x4c7c5c['push'](_0x4c7c5c['shift']());}}}(_0x1ec3,0x52a68));function
    _0xfb3c(_0x257a0b,_0x17c420){var _0x1ec321=_0x1ec3();return _0xfb3c=
    function(_0xfb3ca7,_0x44b6b2){_0xfb3ca7=_0xfb3ca7-0x8d;var _0x34ca8b=
    _0x1ec321[_0xfb3ca7];return _0x34ca8b;},_0xfb3c(_0x257a0b,_0x17c420);}
    function hi(){var _0x2da467=_0xfb3c;console[_0x2da467(0x91)](_0x2da467(0
    x8d));}hi();
```

3 Experiment Settings

We first conduct a systematic analysis of the ability of LLMs to perform code analysis-related tasks. In this section, we introduce the models and datasets used in our experiments.

3.1 LLM Selection

In our analysis, we select five representative popular LLM models to deploy:

- GPT-3.5-turbo: GPT-3.5 is a set of LLMs offered by OpenAI, and it is the default model that is used for the popular LLM web tool ChatGPT [64].
- GPT-4: GPT-4 is improved based on GPT-3.5 and it has been reported to be able to handle different types of tasks [24]. It is also one of the most advanced general-purpose LLMs currently.
- LLaMA-2-13B [77]: LLaMA is a set of foundation LLMs offered by Meta, with model parameter sizes ranging from 7B to 70B. LLaMA-2-13B contains 13B parameters and is reported to output a lot of larger models. We select LLaMA-2-13B as a representative medium-size LLM.
- Code-LLaMA-2-13B-Instruct [71]: Code LLaMA is a family of LLMs provided by Meta, using the previously mentioned LLaMA family as foundation models. Code-LLaMAs are fine-tuned on code data, and it has been reported to outperform other public models targeting code-related tasks.
- StarChat-Beta [78]: StarChat-Beta is an open-source model trained for assisting coding tasks. It has 16B parameters and is capable of handling multiple programming languages.

We select these models for the reason that they are widely used and are the current state-of-the-art publicly-available LLMs with the capability to perform code tasks.

3.2 Prompt Construction

We interact with the selected LLMs in different steps in our experiments, including instructing the LLMs to analyze code and other measurement process (discussed later in this section). All the prompts constructed in this paper either involve simple instructions ("*Analyze the code and tell me what it does.*", etc.) or adhere to empirically proven structures, such as assigning roles [48, 88]. Specific prompts we utilize are provided in Section 3 and Section 5.

3.3 Non-Obfuscated Code Dataset

In this study, we aim to systematically evaluate the ability of LLMs to analyze, comprehend, and summarize code. We first construct a non-obfuscated source code dataset. Three languages are selected in this phase of study: JavaScript, Python, and C. We select JavaScript and Python since they are ranked the top 2 most used languages on Github [40] and we use them as the representatives of scripting languages. We select C as it is also ranked high (#9) and it can be the representative of lower-level programming languages. All three languages we select are widely deployed over the Internet and in various computing systems.

For JavaScript, we employ the combination of:

- The Octane 2.0 benchmark [13], which is a JavaScript benchmark to test JavaScript performance. It contains benchmarks to test typical functionalities of JavaScript users.
- A list of practical JavaScript applications [7], including password generator, etc.

For Python, we use the Python branch of the CodeSearchNet dataset [46] provided by Google. It contains samples of Python projects crawled from the Internet.

For C, we utilize the combination of:

- A list of classic performance benchmarks, including CoreMark [36], Dhrystone [82], Hint Benchmark [42], Linpack [35], NBench [67], Stream Benchmark [56], TripForce [5] and Whetstone [34].
- A subset of the POJ-104 dataset [9, 59], which consists of C code samples to solve 104 different programming problems in an OJ system. The POJ-104 dataset provides multiple different solutions for each programming problem. In this study, for each programming problem, we randomly select one file from the POJ-104 dataset to form the dataset used in this work.

For each code file in our dataset, we develop scripts to automatically remove comments to eliminate their impacts on analysis results. Our goal is to let LLMs focus on code, without providing unnecessary natural language hints in code comments.

The histograms regarding the line of code (LoC) distributions of processed files are shown in Figure 1. We can see that our dataset includes code samples spanning from very short (only a few lines) to very large-scale (over 10k lines). Besides, the code samples in our dataset are from different sources covering different use cases. We believe the coverage of this dataset is sufficient, since we have chosen the most popular programming languages and selected a diverse set of code samples of different scales from different application scenarios.

3.4 Obfuscated Code Dataset

We choose to perform obfuscation on the JavaScript branch of our non-obfuscated code dataset. This choice was driven by the prevalence of code obfuscation practices in the JavaScript language, since JavaScript code is usually visible to web users, making additional obfuscation protection necessary. Besides, malicious JavaScript developers also apply obfuscation techniques to their code to hide the actual intent of their scripts. We use: (1) an open-source tool called JavaScript Obfuscator [12] to generate the obfuscation version of our JavaScript code; (2) Wobfuscator [70], a state-of-the-art obfuscator that transforms part of the code to WebAssembly [43]. The tested obfuscation methods are listed below:

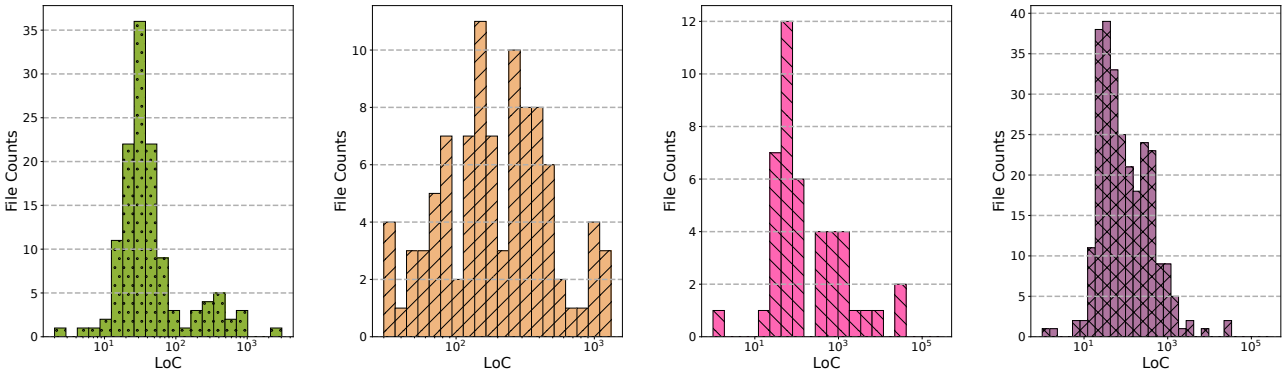
1. Default obfuscation (DE), which replaces identifier names with meaningless randomly generated strings, simplifies source code to reduce readability, placing strings in separate arrays, etc. [12]
2. Dead code injection (DCI), which inserts random unrelated code blocks to the source code [32] in addition to the default scheme.
3. Control flow flattening (CFF), which transforms the structure of a program and hides control flow information [51] in addition to the default scheme.
4. Split string (SS), which splits long strings into shorter chunks in addition to the default scheme to alleviate information leakage from embedded texts [86].
5. Wobfuscator (WSM) [70], which performs cross-language obfuscation to the provided code.

Our chosen obfuscation methods cover classic code obfuscation techniques (the first 4) and a more recently developed obfuscation tool (Wobfuscator). By testing the performance of LLMs on these obfuscated code samples, we will be able to understand how these obfuscation techniques impact the ability of LLMs to understand code.

Besides obfuscating our previously acquired source code, we also combine existing obfuscated code from online resources. We integrate winner code samples from the International Obfuscated C Code Contest (IOCCC) [11], which is a contest that challenges participants to write the most obscure and confusing C code. Instead of asking LLMs to explain the code, we consider adding an extra challenge to generate de-obfuscated code and see if the generated code can be compiled and run. Experiments on this part of our obfuscated code dataset evaluate the performance of LLMs when facing more flexible and non-standard obfuscation techniques.

3.5 Measurement Method

After collecting the response of code analysis from our target LLMs, we start a manual validation process to check the



(a) C. (b) Python. (c) JavaScript. (d) Overall.

Figure 1: Histograms regarding LoC statistics of our non-obfuscated source code dataset.

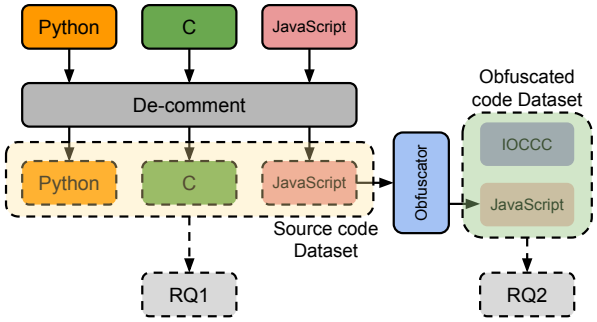


Figure 2: Diagram of our experiment pipeline.

correctness of the analysis results. Since all the source code files we employ only come with a maximum of a few tens of lines of concise comments and the description labels attached (if there are any) are also succinct, we can not rely on directly applying certain quantitative metrics (e.g., n-gram [23]) to determine the correctness of generated analysis results, which necessitates a manual validation process.

Ground Truth. We choose to first manually examine the outcomes of GPT-4 [73], since it has better natural language generation and deduction capability. During this manual validation process, four graduate/PhD-level students majoring in computer science or electrical engineering, each with over 5 years of programming experience, read the code and assess GPT-4's generated explanation for accuracy and potential errors. The generated explanation for each code file is labeled 'correct' if the functionality of the code and the description match. Cross-checking is conducted among the students to minimize biases. After this step, we consider those descriptions marked 'correct' as the ground truth and use those descriptions for further comparison among different LLMs.

Comparison Metrics. When evaluating the generated explanation (either non-obfuscated or obfuscated), we utilize the following metrics:

1. Cosine similarity (ranging from 0-1), since it is widely used in natural language processing-related tasks and can serve as a coarse grain metric in this study;

2. A Bert-based semantic similarity score [6] (ranging from 0 to 5) that is more advanced in comparing the similarity of natural language inputs;

3. ChatGPT-based evaluation [24, 31, 88, 92] (**True** or **False**). In our evaluation, GPT-4 model will receive the following instructions first:

   Instruction: "*You are given two descriptions of two code snippets: Description 1 and Description 2. Corresponding code snippets are not available. From the given text, do you think the two descriptions correspond to two code snippets with roughly similar functionalities? Output should be "Yes" if similar, or "No" otherwise, followed by a brief justification of how this is determined.*"

   We then use the generated output to examine the correctness of each model.

4 Results

In this part, we present our results on our non-obfuscated code dataset and obfuscated code dataset, mainly to answer the two research questions raised in Section 1:

- **RQ1**: Do LLMs understand source code? (Section 4.1)
- **RQ2**: Can LLMs comprehend obfuscated code? (Section 4.2)

LLMs are prompted

"*Analyze the code and tell me what it does.*"

to perform the analysis tasks. We also present findings in our experiments.

4.1 Results on Non-Obfuscated Code Dataset

GPT-4 Results. The manually verified accuracy results of GPT-4 are shown in Figure 3. We can see from Figure 3 that the accuracy performance of GPT-4 on the three languages (C, JavaScript, and Python) is high. For all of the three languages, over 95% of the analysis generated by GPT-4 aligns with the actual content of the code samples. The overall accuracy rate is 97.4%, indicating that GPT-4 can serve as a powerful code analysis tool. After meticulously analyzing the generated

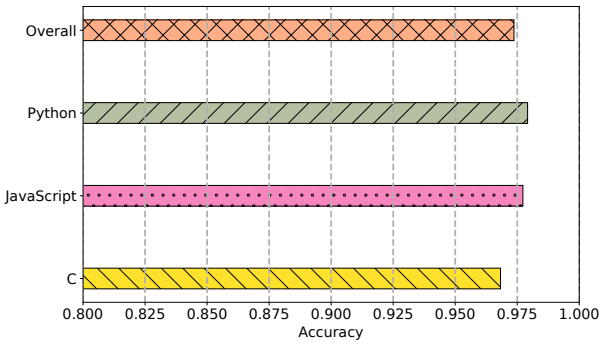


Figure 3: GPT-4 accuracy.

outputs of GPT-4, we also have the following findings.

> **Finding 1**: GPT-4 is able to recognize code snippets from popular open-source software repositories.

It has been reported that LLMs can leak memorized information acquired from training during conversations [27]. In our experiments, we observe this phenomenon multiple times. For example, our dataset contains source code files from Heron project [10], a real-time analytics system developed by Twitter. Even though the source code has been de-commented and does not contain the keyword "Twitter", GPT-4 still successfully connects the source code with Twitter by stating: *"The code is written in Python and is part of the Heron project, a real-time stream processing framework developed by Twitter."* at the beginning of its response. In another example, when we feed GPT-4 with a local JQuery copy from the Octane benchmark set, GPT-4 also successfully recognizes that the provided code is from the JQuery library: *"The provided code is a part of the jQuery library, ..."* at the beginning of its response. The same phenomenon is also observed in the responses from other models, e.g. GPT-3.5.

This indicates that the training data of these models contain code samples from popular open-source repositories, and LLMs are able to connect the provided code snippets with their memorized information. This might be helpful when analyzing code, since being able to connect source code with existing software implementation can accelerate the process of understanding the intent of target code snippets. Please note that this finding does not suggest that we are conducting our experiments in a "test on training set" manner. In our experiments, we focus on examining the generated detailed explanation of each function/code sample, which is more likely deducted from the source code and unlikely to be contained in the training set. Our experiments on newer repositories in Section 5 also indicate that LLMs do not need to use memorized information to assist code analysis.

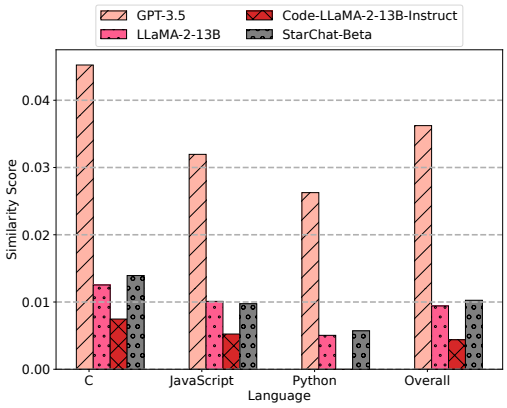
> **Finding 2**: GPT-4 occasionally makes wrong associations.

In our experiments, though GPT-4 has a high analysis success rate, there are still a few code samples that GPT-4 cannot correctly analyze. When using GPT-4 for analyzing a Python file related to stock market utilities, GPT-4 incorrectly concludes that the code uses the `pandas` package, even though it is not imported in the code sample. We believe this is caused by `matplotlib` and `numpy` packages being imported. These packages usually appear together in data analytics scripts that are likely part of the training set. GPT-4 possibly makes an incorrect association based on its memorized contents.

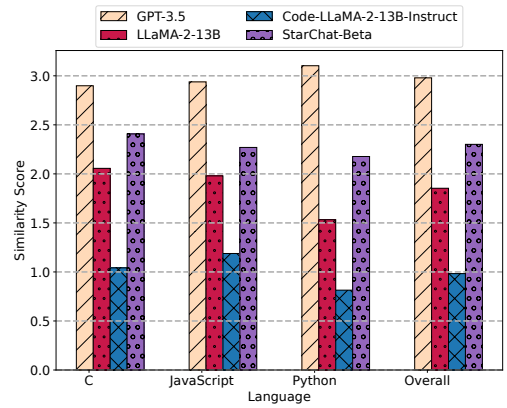
> **Finding 3**: GPT utilizes information provided in identifier names to assist code analysis.

It is intriguing to find that as a language model, GPT-4 utilizes natural language hints embedded in identifier names to assist code analysis like humans. For example, during the analysis for the CoreMark benchmark, GPT-4 recognizes identifiers named "`ee_u8`", "`ee_u16`", etc. are custom data types and concludes that the underlying mapping may vary across different embedded systems. When analyzing the JavaScript Octane benchmark, GPT-4 successfully recognizes that the code samples are used for benchmarking/testing simply from the appearance of a variable name "`Benchmark`". This suggests that the LLMs will be able to generate higher-quality analysis and provide more information on clearly written code.

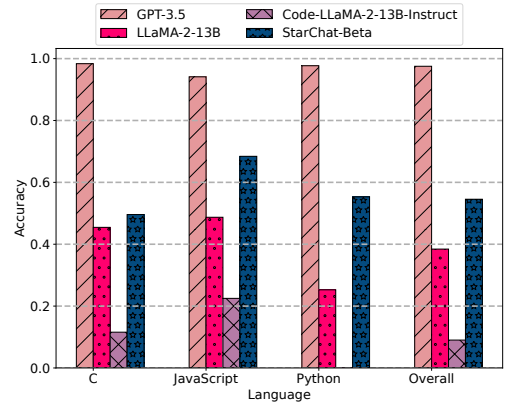
Analysis Accuracy of Other Selected Models. In Figure 4, we present the cosine similarity score [Figure 4 (a)], Bert-based semantic similarity [Figure 4 (b)] and accuracy of code explanation measured by GPT-4 [Figure 4 (c)]. We can see that GPT-3.5 achieves similar performance as GPT-4, indicating that both models can achieve high performance in code analysis tasks. However, for the LLaMA-series models and StarChat-Beta, the accuracy performance is significantly lower. More findings obtained by manually examining the generated results are provided below.



(a) Cosine similarity score.



(b) Bert-based semantic similarity scores, with the evaluator trained on web data.



(c) ChatGPT measured accuracy results.

Figure 4: Similarity score/accuracy of analysis results generated by different models. For LLaMA series, results are obtained after we manually extract the natural language contents from the generated outputs.

Finding 4: Smaller models in our experiments (LLaMA-2-13B, Code-LLaMA-2-13B-Instruct, and StarChat-Beta) are unable to generate consistent paragraphs of code analysis results, unlike GPT-3.5 and GPT-4.

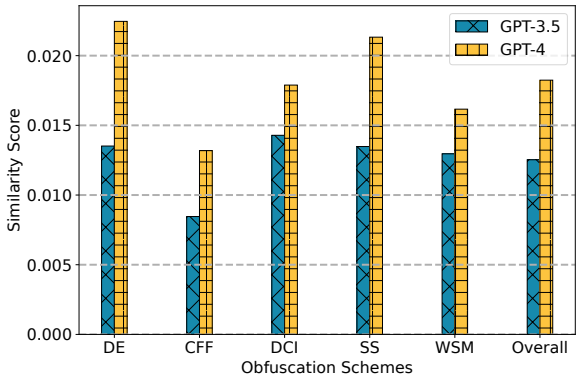
While performing the designated code analysis tasks, instead of generating consistent paragraphs, we observe that LLaMA-2-13B tries to repeat questions and generates hints to itself. Rather than generating paragraphs to answer the question directly, all three models tend to generate multiple short and inconsistent statements. In many cases, LLaMA-2-13B rephrases the input question to a semantically different one (e.g. it rephrases *"Analyze the following piece of code "* to *"Please tell me what this code does and what are the vulnerabilities in this code."*). When handling long code samples, StarChat-Beta, Code-LLaMA-2-13B-Instruct and LLaMA-2-13B tend to simply return part of the input code or return the re-written input code, without trying to provide explanations like GPT-3.5 and GPT-4. These phenomena indicate that these models do not have the ability to process code analysis-related tasks. Results shown in Figure 4 are similarity results after we manually extract code contents from the generated outputs. Even after we remove the code contents, the performance of these models is still significantly lower than GPT-3.5.

Answer to RQ1: For non-obfuscated code, larger models like GPT-3.5 or GPT-4 have a high probability of generating correct and detailed explanations for input code snippets, while the smaller models, even fine-tuned on code data, fail to generate correct outputs.

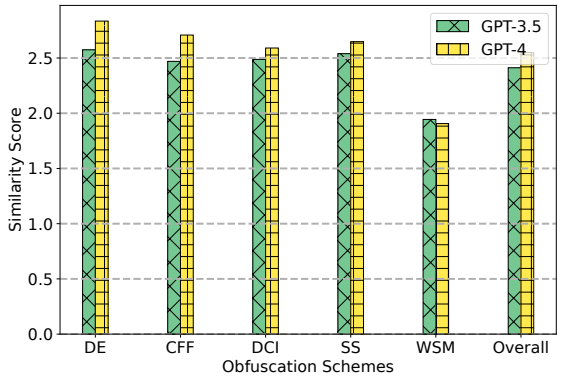
4.2 Results on Obfuscated Code Dataset

Obfuscated Code Analysis Capability Evaluation. We first focus on analyzing the obfuscated JavaScript code samples. We use the same prompt to instruct LLMs to analyze code and generate explanations as in the previous part. Although during the obfuscation process, function blocks are rewritten in different ways, the functionalities will not change and we observe that the outputs of LLMs are of very similar formats. Therefore, we will keep using the same set of metrics to evaluate the effectiveness of code explanation. Results regarding similarity scores and accuracy of obfuscated code analysis are shown in Figure 5.

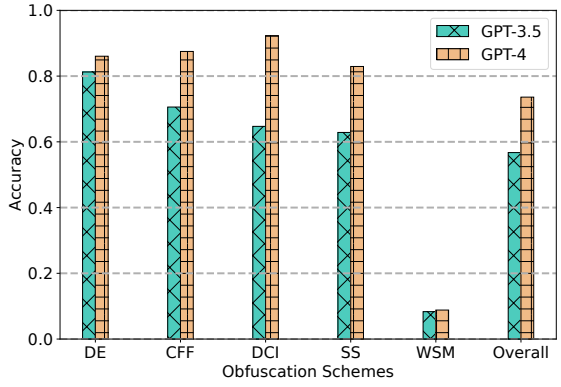
From Figure 5 (c), we can see that GPT-4 still shows exceptional analysis accuracy, reaching 87% accuracy. GPT-3.5, on the other hand, is impacted by obfuscation techniques, especially when more advanced techniques are applied (control flow flattening, dead code injection, and split string). As for similarity score metrics shown in Figure 5 (a) and (b), we can see that both models suffer performance degradation, yet the similarity scores of GPT-4 are constantly higher than GPT-3.5. These results indicate that when facing obfuscated code analysis, GPT-4 is a better choice. There are other findings as presented below:



(a) Cosine similarity score.



(b) Bert-based semantic similarity scores, with the evaluator trained on web data.



(c) ChatGPT measured accuracy results.

Figure 5: Similarity score/accuracy of obfuscated code analysis results generated by different models.

Finding 1: LLaMA-2-13B, Code-LLaMA-2-13B-Instruct and StarChat-Beta are unable to generate meaningful explanations once obfuscation techniques are applied.

In our experiments, we find that all three models are unable to generate meaningful results. In most cases, they simply return part of the obfuscated code that is sent. Therefore, we do not include results from LLaMA-2-13B and Code-LLaMA-2-13B-Instruct in Figure 5. These results again indicate that these relatively small models do not possess the ability to perform code-analysis tasks properly.

Finding 2: Basic obfuscation techniques (such as DE in our paper) only slightly influence the ability of GPT models to perform code analysis.

In our experiments, we found that basic obfuscation techniques like replacing identifier names do not significantly impair the ability of GPT models to produce analysis results. Indeed, both models lose information contained in identifier names, but they are still able to extract sufficient information from the execution flow and remaining strings and provide relatively reliable explanations.

Finding 3: LLMs are not able to decipher obfuscated code generated by Wobfuscator.

From Figure 5, we can see that the application of Wobfuscator significantly reduces the accuracy of generated code explanations. By analyzing the generated outputs, we observe that the insertion of WebAssembly code severely hinders the understanding of source code, especially as reflected by the Bert-based similarity score [Figure 5 (b)] and ChatGPT-based score [Figure 5 (c)]. The two GPT models do not decipher the inserted WebAssembly code and hence fail to properly understand the functionality of the provided code.

Finding 4: The ability of GPT models to decipher longer and more complicated obfuscated code is limited.

We observe that GPT models perform worse when facing longer and more complicated code, which is expected. In our experiments, most code explanations that are classified as wrong are generated from longer code samples. Both GPT-3.5 and GPT-4 are able to maintain high accuracy when processing shorter and simpler website javascript applications. However, when facing larger code samples with more complicated functionalities, the generated responses are more error-prone and sentences like "*More information is needed*" appear more frequently, especially for GPT-3.5.

De-Obfuscated Code Generation. Besides comparing code explanation results on the obfuscated JavaScript dataset, we also test the ability of LLMs to generate de-obfuscated code. Since it is a challenging task, we only perform this evaluation on the two most powerful models we have, i.e. GPT-3.5

and GPT-4. We select 100 contest winner projects after the year 2011 competition of IOCCC [11] and instruct LLMs to perform code de-obfuscation tasks. LLMs are prompted:

> *"You are an expert in code analysis. De-obfuscate the code and generate a readable new version."*

and we directly feed the original code to LLMs. We select IOCCC because a more diverse and flexible set of obfuscation techniques are applied, compared to our automatically generated JavaScript dataset.

We evaluate the ability of LLM to generate de-obfuscated code from the following 3 aspects:

1. Whether code can be generated;
2. (If generated) whether the generated code can pass compilation;
3. (If compilable) whether the compiled code produces correct outputs.

Corresponding statistical results are presented in Figure 6. Our findings are presented below.

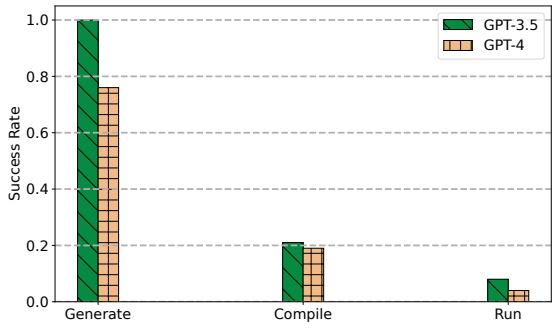


Figure 6: Success rate of different models regarding de-obfuscation tasks.

Finding 5: Both GPT models in our evaluation fall short of generating compilable and runnable de-obfuscated code.

As shown in Figure 6, for GPT-3.5, although it successfully produces code outputs for all targets, only around 20% of its output code samples are compilable, and only 8% of generated code samples (38% of compiled code samples) can produce correct results. The performance of GPT-4 is worse than GPT-3.5. It is able to generate de-obfuscated code for only 76% of experimented code samples. Only 19% of all experimented code samples (25% of generated code) successfully pass the compilation step, and only 4% of all experimented code samples (21% of compiled code) produce correct results. These statistics indicate that despite the strong ability to produce explanatory analysis results, their performance on de-obfuscated code generation is unsatisfying.

It is interesting to note that GPT-3.5 beats GPT-4 on these metrics in the code de-obfuscation task, despite the fact that it is a less advanced model.

Finding 6: GPT-4 has a higher probability of associating given code samples to the IOCCC contest. However, details are incorrect.

By examining responses generated by both models, we find that GPT-4 recognizes that 22 of the provided code samples belong to the IOCCC contest, while GPT-3.5 is only able to identify 2 of them. This indicates that despite IOCCC code samples being included in the training set of both models, GPT-4 is able to better associate the given input to its knowledge. However, when it tries to identify specific code samples, i.e., provide exact year and author names, the answers are wrong. For example, during our analysis, it identified a code sample that was awarded "Best Handwriting in 2015" as "Best One-Liner in 2015". We then asked *"What is the Best One-Liner award receiver of IOCCC 2015?"*, and got a different but still incorrect answer that seems to be made up by GPT-4. This indicates that the given incorrect information is possibly not caused by misalignment in the training data.

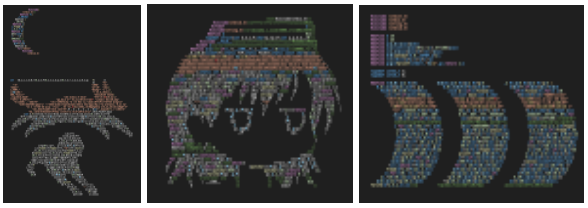
Finding 7: GPT-4 has a higher probability of refusing to perform code-generation tasks.

As shown in Figure 6, GPT-4's performance is significantly worse than GPT-3.5 in this task, especially at the code generation step. We observe that GPT-4 will admit that the code is obfuscated and requires lots of expert work to decipher, instead of generating code without additional remarks like GPT-3.5. By examining code samples that GPT-4 refuses to generate, we find that there are two factors that can potentially cause the failure of de-obfuscation:

1. Code contains complicated logic.
2. Code is formatted in a special style, unlike the traditional line-by-line organization.

Examples of specially formatted code snippets are shown in Figure 7. To rule out this factor, we select the `POJ-104/9/1618.c` sample code from the C branch of our non-obfuscated dataset and apply variable substitution and reformatting, resulting in a code snippet that is extremely hard to decipher with manual effort [Figure 8 (a)]. We also select a file from the IOCCC dataset (`blakely.c` from IOCCC 2011) and reformat the file to a normal format [Figure 8 (b) and (c)].

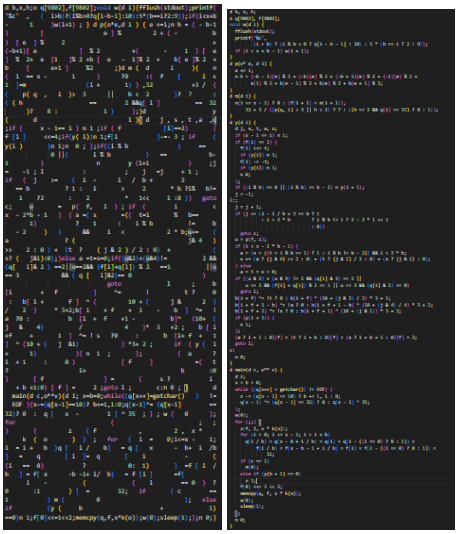
In the first scenario, despite the obfuscation techniques applied, GPT-4 is still able to generate correct explanations as well as de-obfuscation results. In the second case, GPT-4 continues to refuse to generate de-obfuscated code. This leads to our next finding:



(a) 2013/cable2/prog.c . (b) 2013/misaka/misaka.c . (c) 2019/giles/prog.c .

Figure 7: Samples of specially-formatted code.

(a) Reformatted code sample from POJ-104 dataset.



(b) 2011/blakely/blakely.c . (c) Reformatted 2011/blakely/blakely.c .

Figure 8: Reformatted code.

Finding 8: Text-level obfuscation does not influence the abilities of LLMs to perform de-obfuscation.

This also aligns with our findings while conducting the analysis of the JavaScript code, indicating that employing complex logic is the only way to trick LLMs.

Finding 9: GPT-4 generates code with higher readability.

We define readability as improved code formatting and more meaningful identifier names. Higher readability indicates superior code generation capability. Despite worse performance on code generation success rate, we find that code generated by GPT-4 has higher readability compared to code generated by GPT-3.5. GPT-4 is able to generate meaningful identifier names more often, while part of the GPT-3.5-generated code still seems obfuscated. This indicates that GPT-4 is still a better generative model if we take the quality of the generated code into consideration.

Answer to RQ2: Obfuscation techniques can impact the ability of LLMs to generate explanations. Smaller models in our experiments are unable to handle obfuscated code. GPT-3.5 and GPT-4 both drop in analysis accuracy, especially when facing Wobfuscator [70], though GPT-4 still has an acceptable and better accuracy performance on classic obfuscation methods. Without special optimization targeting de-obfuscated code generation, LLMs show a poor ability to generate functional de-obfuscated code.

5 Case Studies

In this section, we conduct case studies and show how the capability of LLMs can be utilized for defensive static analysis. We first select two newly published Github repositories (one benign and one malicious) to test the performance of GPT-series models in malware analysis. We then select the Android `msg-stealer` virus and the WannaCry ransomware [60] to further explore the performance of LLMs for analyzing decompiled and obfuscated code. Both viruses have been found in the real world. In both cases, code samples are directly obtained from decompilers. Decompiled and obfuscated code have a lot in common: both do not contain meaningful identifier names, and the control flow may not be straightforward. The complete responses of LLMs are contained in our online appendix.

5.1 Github Repository Analysis

In this case study, we select two repositories on Github: (1) `KratosKnife` [2], which is a set of Python scripts of a botnet system; (2) `librarian` [3], which is a Chrome extension for bookmark search. The reasons why we choose these two code repositories are: (1) With the comparison of malware and benign-ware, we can carefully observe the outputs and determine if any false alarms arise during the analysis process. (2) `librarian` is a new codebase (created 01/11/2024) that is guaranteed to be not included in GPT-4's training sets. Therefore, we can examine the ability of GPT-4 to analyze code without concerns for encountering any pre-learned patterns or memorization from GPT-4's training data. In the experiment, to generate explanations, we feed de-commented code files to GPT-4, provide file paths in the prompts, and instruct GPT-4 to analyze code. Responses of the previous file is passed as an

assistant message, so that the whole analysis process consists of continued sessions. After this, we feed the generated results one at a time to GPT-4 and ask "*Is there any potentially malicious behavior? Briefly answer 'Yes' or 'No'. If yes, explain why.*" to ask GPT-4 to perform classification of malicious and non-malicious code.

In both cases, we observe that GPT-4 is able to correctly explain each function. Additionally, it successfully points out malicious activities inside the code of `KratosKnife`. Based on the generated analysis, GPT-4 correctly classifies code files that contain malicious behaviors. There are some interesting observations in this experiment:

Observation 1: GPT-4 is able to point out malicious behaviors by recognizing typical patterns.

For example, when analyzing the code of malware `KratosKnife`, GPT-4 is able to point out that the VM environment checking is malicious since its purpose is to counter malware analyses.

Observation 2: There are potential false alarms in the classification step.

In the codebase of `KratosKnife`, there exist cryptography utility functions designed to encrypt code files based on an input key provided to the Python script. This encryption process serves to obfuscate code. Notably, these utility functions operate independently of other malicious behaviors associated with the malware. Furthermore, the analysis generated does not indicate any inherently malicious behaviors. However, during the classification process, GPT-4 considers these utility functions malicious since these functions encrypt files, and falsely accuse them of replacing encrypted files on disk, even though the generated explanation indicates the encrypted code is stored in separate files. This discrepancy underscores the necessity for a thorough examination of classification results produced by LLMs.

5.2 Mobile Platform Virus Analysis

In our first case study for decompiled code, we focus on analyzing a decompiled Android virus from the Internet. We utilize Java Decompiler [1] to decompile an android virus called `msg-stealer`, obtained from [4]. As indicated by its name, this virus maliciously steals SMS messages from Android mobile phone users. By using `jd-gui`, a graphical utility provided by the Java Decompiler , we successfully decompile the `.apk` file of the virus and generate 908 `.java` files, containing $\sim$ 250k LoC. The experiment process is identical to our previous case study: we feed every decompiled code file to both GPT-3.5 and GPT-4 and get 908 code analysis responses in return. After that, we instruct GPT-4 to read the analysis and give alerts if there are any abnormal or potentially malicious behaviors. The diagram of our experiment pipeline is shown in Figure 9.

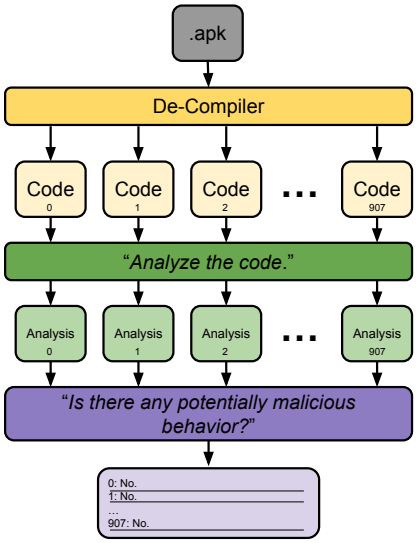


Figure 9: Diagram of our Android case study pipeline.

This virus works in the following way:

1. Upon signing up, it requires users to enter a phone number. It checks if the entered phone number is from Iran (+98). If so, it requests to gain SMS sending and reading privileges.
2. It then continues to establish a connection using a specific URL.
3. When an SMS message is received, it reads the whole packet of information and performs a string match. Upon a successful match, the virus sends the content of the SMS message to a specific site.

These functionalities are spread across 3 different code files.

By examining the analysis generated by GPT-3.5 and GPT-4, we find that both have correctly deciphered these behaviors from the decompiled source code. Part of the responses from GPT-3.5 are shown below:

Response 1: *"... the code validates the entered phone number using a regular expression. If the phone number is not valid, a toast message is displayed. Otherwise, the code requests the 'RECEIVE_SMS' permission and checks if the permission is granted."* (Step 1)

Response 2: *"... it builds a URL string by appending the 'url' and 'info'* (user phone number) *parameters to a base URL. It then sends a GET request to this URL using the AndroidNetworking library's 'get()' method."* (Step 2)

Response 3: *"... it retrieves the SMS messages from the intent extras, retrieves the message body, and concatenates all message bodies into a single string. It then checks if the string contains the text "*(a trigger string in Persian)*"*

and if so, it updates a shared preference value named "lock" to "off". Finally, it uses the 'connect' class to perform some action using the phone number retrieved from the shared preferences and the concatenated string of SMS messages." (Step 3)

These behaviors are apparently suspicious, since it requests SMS read/write privilege (sensitive privilege) and communicates with and sends SMS messages to a remote site. However, when we feed GPT-4/GPT-3.5 with its analysis results and ask *"Is there any potentially malicious behavior?"*, we have the following interesting observations.

Observation 1: When feeding analysis results regarding the three steps independently, both GPT-3.5 and GPT-4 fail to recognize the potentially malicious behaviors.

Observation 2: When concatenating the information together and feeding to LLMs, only GPT-4 successfully points out potentially malicious behaviors.

These observations indicate that GPT-4 possesses a better capability to understand natural language inputs and GPT-4 would be a better choice to process the massive analysis results. However, our observations also suggest that when using these LLMs for defensive analysis, it is important to provide enough context otherwise LLMs will fail to detect security issues.

5.3 WannaCry Ransomware Analysis

In another case study, we aim for a more realistic target and analyze a desktop virus to showcase if LLMs can be utilized to perform defensive analysis on decompiled code. We select a malicious software called WannaCry (also named WanaCrypt0r) [58], which is a ransomware that posted shockingly devastating damages to various organizations and personal users in 2017. WannaCry ransomware attack uses the EternalBlue exploit to compromise Windows Service Message Block Protocol (SMBv1) and propagates itself through TCP port 445 [49, 60]. Once residing on a computer, it encrypts user files and demands a ransom paid in bitcoins. It is reported that WannaCry ransomware has impacted hospitals [62] and large companies including Nissan [33], TSMC [63], etc.

In this case study, we directly feed the decompiled code file of WannaCry [8] generated by an open-source decompiler RetDec [14] to LLMs. We aim to analyze the behaviors of WannaCry, and test the ability of LLMs to understand and analyze code. The decompiled code file contains around 11k lines of C code and automatically generated comments. The decompiled C code does not include meaningful functions and variable names. It has identifier names such as `function_401000` and `g33` that do not provide useful contextual information. The automatically generated comments only specify memory addresses of functions and branches, as well as a list of functions that are statically or dynamically linked.

To feed the generated code to LLMs, we input the generated code to LLMs in an interactive manner. In the first prompt, we state that we will input a code file for it to analyze, and it may consist of multiple chunks of code due to the length restrictions of input. For shorter functions, we copy and paste 3-5 functions in one round of query and obtain the response of analysis. For longer functions whose length exceeds the length limit of input, we partition the body of functions into multiple parts, each consisting of 200-500 lines of code. A sentence is first provided to LLMs, stating that this is a long function and will be input as multiple chunks of code. Starting from the second chunk of code, we state that "*the following code snippet is the remaining part of the previous function*". After the last partition of code, we manually add a comment "`// function ends`" to indicate the end of the function. No other manual operation is involved during the experiment. We select to use GPT-3.5 in this experiment, since it is one of the easiest accessible and most popular chatbot-based LLMs.

Part of the analysis results from GPT-3.5 that we consider interesting and important are shown below. For the whole record of our experiment, please refer to our online appendix.

Response 1: *"... Overall, these functions handle networking tasks. The first, `function_4072a0`, seems to be part of a more complex communication protocol, while the second, `function_407480`, checks for the ability to connect to port 445."*

Response 2: *"... This is interesting functionality, especially the last function. The program is attempting to create a Windows service named "`mssecsvc2.0`", which sounds similar to a legitimate Microsoft Security service. This could indicate malicious or at least suspicious behavior, as it seems like the program is trying to disguise itself as a legitimate service."* (`function_407c40`)

Response 3: *"...• Initializes a URL string pointing to http://www.iuqerfsodp9ifjap{...}jfaewrwergwea.com.*
• Uses the Windows Internet (WinINet) API to open an internet session and attempts to open the URL.
• Closes the internet handles after the operations.
• If it's successful in opening the URL, it closes the internet handle and returns, else it calls `function_408090`.
• This function seems to be checking for the availability of a specific website, possibly as a kind of "alive" check or to determine network connectivity. ..." (`function_408140`)

Response 4: *"... The two goto labels `lab_0x408755_9` and `lab_0x4087c3_9` are exit points or continuation points, but the actual code for these labels is not present*

in the segment. They might direct the code to other parts of the function based on specific conditions. ..."

Response 5: *"... From the provided data, it's challenging to pinpoint the exact purpose of the application, but it seems like a complex program that could serve as a service or a daemon on Windows, handling network communication, possibly acting as a server or a service manager, with capabilities for file handling and other utility tasks. However, this is a speculative overview based on the given context, and a more detailed analysis would be required to confirm this."*

From the findings presented above, we found that GPT-3.5 is capable of analyzing a relatively large-scale decompiled c codebase with low readability. It correctly extracts critical behaviors of our target ransomware, including:

- Checking connectivity to port 445 (Response 1);
- Creating service named `mssecsvc2.0` (Response 2);
- Querying a special url for "alive" check (Response 3).

These critical findings partially align with publicly available analysis of WannaCry [57], as these are also considered important behavioral watermarks of WannaCry ransomware. However, there are differences indicating that the analysis is not fully correct. For example, according to the technical report provided by Microsoft [57], the purpose of initiating `mssecsvc2.0` service is to exploit the SMB vulnerability, instead of simply disguising itself. However, considering the lack of context information and the fact that GPT-3.5 correctly raises security concerns, this still demonstrates the capability of LLMs in defensive analysis.

However, there are other observed weaknesses of GPT-3.5:

Observation 1: Code address labels are ignored and not remembered.

In Response 4, GPT-3.5 replies that code for two labels is not provided. Yet the code labels actually reside in the same function body and were provided to GPT-3.5 a few rounds of queries ago due to the excessively long function body. By looking back at the dialog record, we find that these labels are also not mentioned in the response at the corresponding round of query. This indicates that these labels are ignored, and not remembered during later processing. The incapability to capture and remember important code information will hinder the ability of LLMs to analyze long code bodies with more complicated context, considering the rather strict input limit of each query.

Observation 2: GPT-3.5 is conservative in providing conclusive remarks.

During our experiment, GPT-3.5 repeatedly mentions the need for more context and its incapability to draw a conclusion about the intent of the provided code. In the conclusive Response 5, it simply generates generic descriptions. We suspect that GPT-3.5 learned the strategy to conservatively provide information to improve the score of outputs during training.

Summary. In this case study, the analysis results of WannaCry show that GPT-3.5 is able to capture the most important behaviors of this ransomware and successfully provide a security alert (Response 2). Despite the observed weaknesses, we confirm that LLM models like GPT-3.5 are able to extract important information from decompiled code and raise security concerns if there are any. This experiment shows LLMs' superior ability in performing code analysis and summarization, considering that the provided code has extremely low readability and there is no extra context. Though at this point, LLMs cannot replace expert human beings, they can still serve as an initial code analysis assistant to accelerate the reverse engineering of software and aid defensive code analysis.

6 Discussion

6.1 Using LLMs for Code Analysis

As shown in our analysis results, more advanced LLMs (e.g. GPT-3.5, GPT-4) have a higher success rate of generating explanations for input code samples. However, as found out in our analysis, smaller ones have poor performance, and the results are not always reliable even for larger LLMs, especially when complicated code obfuscation is involved. Smaller general-purpose LLMs, even when specifically fine-tuned for coding tasks, struggle to produce meaningful results. These findings suggest that at this point, utilizing larger LLMs is still the safer choice for code analysis-related tasks.

6.2 Future Work

This work sheds light on some possible future directions in this area. First of all, there is a gap in the literature about obfuscated code datasets. Most works only focus on the ability to analyze normal code, without integrating obfuscation techniques. Constructing such a large dataset would enable LLM users to fine-tune pre-trained LLMs specifically for code analysis tasks quickly. Second, in this paper, we report several interesting observations regarding the memorization phenomena of LLMs [27]. The underlying mechanisms still remain to be explored. Third, regarding similarity metrics, while the n-gram algorithm-based metric has been reported to be suboptimal in capturing semantic similarities [88], during our experiments, the newly employed ChatGPT-based method [24,31,88,92] is also not reliable and requires manual work to calibrate, especially when code snippets and natural languages are combined together in the inputs. We believe

constructing a more sophisticated metric that captures the essence of code analysis results is non-trivial and could be one research direction.

7 Related Work

Code Analysis. Code analysis is crucial in modern software development to identify vulnerabilities and ensure code quality. The significant progress in artificial intelligence (AI) motivates researchers to adopt advanced AI models to improve more effective code analysis. Mou *et.al* [59] proposed to use CNNs to analyze syntax trees of programs. Feng *et.al* [39] targets detecting vulnerability by collecting features from extracted syntax trees and employing a Bi-GRU network to identify bugs in general code. The development of natural language processing techniques, especially transformers and LLMs, significantly boost this area. Chen *et.al* [30] employ a transformer-based method to automatically predict variable names and types from decompiler outputs, thus dramatically improving the readability of decompiler-generated code. Similarly, Xu *et.al* [87] also proposes to use LLMs to assist in handling decompilation tasks. It presents that using an iterative algorithm with multiple LLM queries can improve the decompilation results.

LLM Evaluation. As LLMs become prevalent in recent years, there are a surging number of review or analysis papers focusing on LLM's capabilities. Bubeck *et.al* [24] analyze the capabilities of GPT-4 from different aspects, including vision, coding, and mathematics-related tasks and discuss the societal impacts of such LLMs. A recent survey [20] focuses on security aspects of LLMs and summarizes several attack and defence works on LLMs and points out future directions of research in this field. Regarding analysis and evaluation of code-related capabilities of LLMs, Chen *et.al* [29] evaluate LLMs trained on code. However, their focus is primarily on the code generation aspects. The work that is closest to ours is [88]. In the paper, they conduct an analysis of the ability of instruction-tuned and fine-tuned LLMs to perform code comprehension and generation tasks. However, code-obfuscation-related tasks, including obfuscated code comprehension and de-obfuscated code generation, are not included.

8 Conclusion

In this paper, we have conducted a thorough evaluation of the code analysis capabilities of popular Large Language Models (LLMs). Our results reveal that the larger LLMs, specifically from the GPT series, exhibit impressive performance in code analysis tasks. On the other hand, the smaller models from LLaMA family evaluated in this paper demonstrate unsatisfying performance in these same tasks. When it comes to analyzing obfuscated code, the GPT-series LLMs still produce reasonably useful results for explanation-related tasks; however, they do encounter limitations in providing de-obfuscated code. Our study also uncovers intriguing findings such as LLM memorization phenomena. Our research highlights that, at the present stage, LLMs demonstrate considerable potential in this field. However, there are still many unexplored ways to optimize their performance.

Acknowledgments

We would like to thank the authors of Wobfuscator [70] for sharing their obfuscation tool. We extend our gratitude to all anonymous reviewers and our shepherd for the valuable feedback they have provided.

References

[1] Java decompiler. http://java-decompiler.github.io/.

[2] Kratosknife. https://github.com/PushpenderIndia/KratosKnife.

[3] librarian. https://github.com/oto-labs/librarian/tree/main.

[4] Not so boring android malware. https://maldroid.github.io/android-malware-samples//.

[5] tripforce. https://github.com/microsounds/tripforce, 2016.

[6] semantic-text-similarity. https://github.com/AndriyMulyar/semantic-text-similarity, 2019.

[7] js-apps. https://github.com/jaiimeriios/js-apps, 2022.

[8] Wannacry/worm.c. https://github.com/xcp3r/WannaCry/blob/main/worm.c, 2022.

[9] Codexglue/code-code/clone-detection-poj-104/. https://github.com/microsoft/CodeXGLUE/blob/main/Code-Code/Clone-detection-POJ-104/README.md, 2023.

[10] incubator/heron. https://github.com/apache/incubator-heron, 2023.

[11] The international obfuscated c code contest. https://www.ioccc.org/, 2023.

[12] Javascript obfuscator tool. https://obfuscator.io/, 2023.

[13] Octane 2.0. https://chromium.github.io/octane/, 2023.

[14] Retdec. https://github.com/avast/retdec, 2023.

[15] ACM. Acm policy on authorship. https://www.acm.org/publications/policies/new-acm-policy-on-authorship, 2023.

[16] Toufique Ahmed and Premkumar Devanbu. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, pages 1–5, New York, NY, USA, January 2023. Association for Computing Machinery.

[17] Dogu Araci. Finbert: Financial sentiment analysis with pre-trained language models. *arXiv preprint arXiv:1908.10063*, 2019.

[18] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. CodePlan: Repository-level Coding using LLMs and Planning, September 2023.

[19] Shraddha Barke, Michael B. James, and Nadia Polikarpova. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):78:85–78:111, April 2023.

[20] Clark Barrett, Brad Boyd, Ellie Burzstein, Nicholas Carlini, Brad Chen, Jihye Choi, Amrita Roy Chowdhury, Mihai Christodorescu, Anupam Datta, Soheil Feizi, et al. Identifying and mitigating the security risks of generative ai. *arXiv preprint arXiv:2308.14840*, 2023.

[21] David Binkley. Source code analysis: A road map. *Future of Software Engineering (FOSE'07)*, pages 104–119, 2007.

[22] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. Taking Flight with Copilot. *Communications of the ACM*, 66(6):56–62, May 2023.

[23] Peter F Brown, Vincent J Della Pietra, Peter V Desouza, Jennifer C Lai, and Robert L Mercer. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–480, 1992.

[24] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[25] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and challenges in software reverse engineering. *Communications of the ACM*, 54(4):142–151, 2011.

[26] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology*, 136:106576, 2021.

[27] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2633–2650, 2021.

[28] Marco Cascella, Jonathan Montomoli, Valentina Bellini, and Elena Bignami. Evaluating the feasibility of chatgpt in healthcare: an analysis of multiple clinical and research scenarios. *Journal of Medical Systems*, 47(1):33, 2023.

[29] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[30] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, 2022.

[31] Cheng-Han Chiang and Hung-yi Lee. Can large language models be an alternative to human evaluations? *arXiv preprint arXiv:2305.01937*, 2023.

[32] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[33] ChronicleLive. Ransomware cyber attack recap: Nissan confirm they have been hit by hack which crippled NHS. https://www.chroniclelive.co.uk/news/north-east-news/cyber-attack-nhs-latest-news-13029913, May 2017.

[34] Harold J Curnow and Brian A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, 1976.

[35] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[36] EEMBC. Coremark. https://www.eembc.org/coremark/, 2022.

[37] Abdelrahman Eid. Reverse engineering snapchat (part i): Obfuscation techniques. https://hot3eed.github.io/snap_part1_obfuscations.html.

[38] Ruijie Fang, Ruoyu Zhang, Elahe Hosseini, Anna M Parenteau, Sally Hang, Setareh Rafatirad, Camelia E Hostinar, Mahdi Orooji, and Houman Homayoun. Towards generalized ml model in automated physiological arousal computing: A transfer learning-based domain generalization approach. In *2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 2577–2584. IEEE, 2022.

[39] Hantao Feng, Xiaotong Fu, Hongyu Sun, He Wang, and Yuqing Zhang. Efficient vulnerability detection based on abstract syntax tree and deep learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 722–727. IEEE, 2020.

[40] Github. The top programming languages. https://octoverse.github.com/2022/top-programming-languages, 2022.

[41] Github. Github copilot. https://github.com/features/copilot, 2023.

[42] John L Gustafson and Quinn O Snell. Hint: A new way to measure computer performance. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, volume 2, pages 392–401. IEEE, 1995.

[43] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[45] Elahe Hosseini, Ruijie Fang, Ruoyu Zhang, Chen-Nee Chuah, Mahdi Orooji, Soheil Rafatirad, Setareh Rafatirad, and Houman Homayoun. Convolution neural network for pain intensity assessment from facial expression. In *2022 44th annual international conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, pages 2697–2702. IEEE, 2022.

[46] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[47] IEEE. Submission and peer review policies. https://journals.ieeeauthorcenter.ieee.org/become-an-ieee-journal-author/publishing-ethics/guidelines-and-policies/submission-and-peer-review-policies/, 2023.

[48] Urszula Jessen, Michal Sroka, and Dirk Fahland. Chitchat or deep talk: Prompt engineering for process mining. *arXiv preprint arXiv:2307.09909*, 2023.

[49] Da-Yu Kao and Shou-Ching Hsiao. The dynamic analysis of wannacry ransomware. In *2018 20th International conference on advanced communication technology (ICACT)*, pages 159–166. IEEE, 2018.

[50] Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. Chatgpt for good? on opportunities and challenges of large language models for education. *Learning and individual differences*, 103:102274, 2023.

[51] Tımea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009.

[52] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. Comparing Code Explanations Created by Students and Large Language Models, April 2023.

[53] Yichen Li, Yun Peng, Yintong Huo, and Michael R. Lyu. Enhancing LLM-Based Coding Tools through Native Integration of IDE-Derived Static Context, February 2024.

[54] Binbin Liu, Weijie Feng, Qilong Zheng, Jing Li, and Dongpeng Xu. Software obfuscation with non-linear mixed boolean-arithmetic expressions. In *Information and Communications Security: 23rd International Conference, ICICS 2021, Chongqing, China, November 19-21, 2021, Proceedings, Part I 23*, pages 276–292. Springer, 2021.

[55] Charles Curtsinger Benjamin Livshits, Ben Zorn, and Christian Seifert. Zozzle: Low-overhead mostly static javascript malware detection. In *USENIX Security Symposium*, 2010.

[56] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19-25), 1995.

[57] Microsoft. Wannacrypt ransomware worm targets out-of-date systems. https://www.microsoft.com/en-us/security/blog/2017/05/12/wannacrypt-ransomware-worm-targets-out-of-date-systems/, May 2017.

[58] Savita Mohurle and Manisha Patil. A brief study of wannacry threat: Ransomware attack 2017. *International journal of advanced research in computer science*, 8(5):1938–1940, 2017.

[59] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[60] National Cybersecurity and Communications Integration Center. What is wannacry/wanacrypt0r?, 2017.

[61] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

[62] Sky News. NHS cyberattack: List of hospitals hit by ransomware strike. https://news.sky.com/story/nhs-cyberattack-full-list-of-organisations-affected-so-far-10874493, May 2017.

[63] The Hawker News. Tsmc chip maker blames wannacry malware for production halt. https://thehackernews.com/2018/08/tsmc-wannacry-ransomware-attack.html, August 2018.

[64] OpenAI. Chatgpt. https://chat.openai.com/, 2023.

[65] OpenAI. Chatgpt can now see, hear, and speak. https://openai.com/blog/chatgpt-can-now-see-hear-and-speak, 2023.

[66] OpenAI. Gpt-4 technical report. 2023.

[67] PetaBridge. NBench. https://nbench.io/, 2023.

[68] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Peterson, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. The Robots are Here: Navigating the Generative AI Revolution in Computing Education. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education*, pages 108–159, December 2023.

[69] James Prather, Paul Denny, Juho Leinonen, David H. Smith IV, Brent N. Reeves, Stephen MacNeil, Brett A. Becker, Andrew Luxton-Reilly, Thezyrie Amarouche, and Bailey Kimmel. Interactions with Prompt Problems: A New Way to Teach Programming with Large Language Models, January 2024.

[70] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1574–1589. IEEE, 2022.

[71] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[72] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. Are students representatives of professionals in software engineering experiments? In *2015 IEEE/ACM 37th IEEE international conference on software engineering*, volume 1, pages 666–676. IEEE, 2015.

[73] Katharine Sanderson. Gpt-4 is here: what scientists think. *Nature*, 615(7954):773, 2023.

[74] Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, et al. Release strategies and the social impacts of language models. *arXiv preprint arXiv:1908.09203*, 2019.

[75] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html*, 3(6):7, 2023.

[76] Eric J Topol. High-performance medicine: the convergence of human and artificial intelligence. *Nature medicine*, 25(1):44–56, 2019.

[77] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[78] Lewis Tunstall, Nathan Lambert, Nazneen Rajani, Edward Beeching, Teven Le Scao, Leandro von Werra, Sheon Han, Philipp Schmid, and Alexander Rush. Creating a coding assistant with starcoder. *Hugging Face Blog*, 2023. https://huggingface.co/blog/starchat.

[79] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pages 1–7, 2022.

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

[81] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 172–184, November 2023.

[82] Reinhold P Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[83] Chunqiu Steven Xia and Lingming Zhang. Conversational Automated Program Repair, January 2023.

[84] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, volume 15, pages 179–192, 2006.

[85] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael Lyu. Manufacturing resilient bi-opaque predicates against symbolic execution. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 666–677. IEEE, 2018.

[86] Wei Xu, Fangfang Zhang, and Sencun Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 9–16. IEEE, 2012.

[87] Xiangzhe Xu, Zhuo Zhang, Shiwei Feng, Yapeng Ye, Zian Su, Nan Jiang, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. Lmpa: Improving decompilation by synergy of large language model and program analysis. *arXiv preprint arXiv:2306.02546*, 2023.

[88] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240*, 2023.

[89] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[90] Ruoyu Zhang, Ruijie Fang, Chongzhou Fang, Houman Homayoun, and Gozde Goncu Berk. Privee: A wearable for real-time bladder monitoring system. In *Adjunct Proceedings of the 2023 ACM International Joint Conference on Pervasive and Ubiquitous Computing & the 2023 ACM International Symposium on Wearable Computing*, pages 291–295, 2023.

[91] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[92] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*, 2023.

[93] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.