# VulBinLLM: LLM-powered Vulnerability Detection for Stripped Binaries

Nasir Hussain
University of California, Los Angeles
nasirhm@ucla.edu

Haohan Chen
University of California, Los Angeles
heyohan@ucla.edu

Chanh Tran
University of California, Los Angeles
ctran0014@ucla.edu

Philip Huang
University of California, Los Angeles
philiph930@g.ucla.edu

Zhuohao Li
University of California, Los Angeles
zhuohaol@g.ucla.edu

Pravir Chugh
University of California, Los Angeles
pravirchugh@ucla.edu

William Chen
University of California, Los Angeles
billchen314@g.ucla.edu

Ashish Kundu
Cisco Research
ashkundu@cisco.com

Yuan Tian
University of California, Los Angeles
yuant@ucla.edu



## Abstract

Recognizing vulnerabilities in stripped binary files presents a significant challenge in software security. Although some progress has been made in generating human-readable information from decompiled binary files with Large Language Models (LLMs), effectively and scalably detecting vulnerabilities within these binary files is still an open problem. This paper explores the novel application of LLMs to detect vulnerabilities within these binary files. We demonstrate the feasibility of identifying vulnerable programs through a combined approach of decompilation optimization to make the vulnerabilities more prominent and long-term memory for a larger context window, achieving state-of-the-art performance in binary vulnerability analysis. Our findings highlight the potential for LLMs to overcome the limitations of traditional analysis methods and advance the field of binary vulnerability detection, paving the way for more secure software systems.

In this paper, we present `Vul-BinLLM` , an LLM-based framework for binary vulnerability detection that mirrors traditional binary analysis workflows with fine-grained optimizations in decompilation and vulnerability reasoning with an extended context. In the decompilation phase, `Vul-BinLLM` adds vulnerability and weakness comments without altering the code structure or functionality, providing more contextual information for vulnerability reasoning later. Then for vulnerability reasoning, `Vul-BinLLM` combines in-context learning and chain-of-thought prompting along with a memory management agent to enhance accuracy. Our evaluations encompass the commonly used synthetic dataset Juliet to evaluate the potential feasibility for analysis and vulnerability detection in C/C++ binaries. Our evaluations show that `Vul-BinLLM` is highly effective in detecting vulnerabilities on the compiled Juliet dataset.

## 1 Introduction

Binary analysis is pivotal for program analysis and provides a deep understanding of how executable files operate. In many cases, the source code is not available for analysis [79, 88], especially with proprietary or legacy software, which is common in commercial software (e.g. Microsoft Windows, Adobe Acrobat), firmware in IoT devices, and third-party libraries [116] where vendors often withhold source code to protect intellectual property. Binary analysis allows security engineers to directly analyze the compiled binary, enabling them to understand how the program functions and identify potential vulnerabilities and weaknesses. Traditional binary analysis tools, such as Ghidra [5] IDA Pro [6], and Angr [101] analyze binary code by translating it into assembly language and providing higher-level representations to facilitate understanding. This is accomplished through two main processes: disassembly and decompilation. Disassembly converts machine code into assembly language, while decompilation translates this code into a high-level programming language. However, this process inherently leads to information loss, for reasons such as loss of high-level constructs, compiler optimizations, and symbol and debug information removal. Thus, this leads to lots of ambiguity in the decompiled code. Additionally, learning and mastering reverse engineering require significant effort and expertise. The raw binary format and the complexity of the analysis process make it challenging and time-consuming for security analysts to detect vulnerabilities in the binaries. Therefore, automating the analysis process is essential to efficiently detect vulnerabilities in binaries, reduce manual efforts, and improve the speed of vulnerability identification.

Recent developments in large language models (LLMs) [1, 15, 19, 77, 97] present a promising avenue for addressing the challenges of binary analysis. LLMs have demonstrated capabilities in code summarization and generation [32, 50, 61, 70–72, 87, 90, 107]. LLM agents [93], which integrate LLMs as a component in the workflow, enable interactions that closely mimic expert-level intelligence. This emergence and rapid adoption of LLMs in code analysis and code copilots raises a critical research question: *Can large language models, with fine-grained prompt engineering and specialized optimizations in system design, assist security and software engineers to effectively reason about vulnerabilities in binary code?*

*Objective of this work:* We aim to provide an LLM-powered approach to enable efficient, and scalable vulnerability analysis for the binary code. However, applying LLMs within binary analysis retains several challenges. First, compilation and decompilation causes a loss of contextual information for the code, which makes it challenging to identify vulnerabilities on the binary level. While source code vulnerabilities are typically classified using standards like Common Vulnerabilities and Exposures (CVEs) [64] and Common Weakness Enumeration (CWEs) [65], translating these definitions to the binary level is challenging. Many reported CVEs involve complex

projects with custom-built toolchains. These projects include build configurations and compilation settings that optimize the binary for various use cases, resulting in an even higher loss of information during the compilation phase. Thus, it is difficult to map the binary representation to its source code. Complex compilation processes add a layer of difficulty in accurately tracking and detecting vulnerable code within binary files. Second, binary code is much more abstract than natural language. Since LLMs rely on probabilities to predict in an auto-regressive manner, they often have difficulty extracting meaningful information from low-level representations directly [94]. Reverse engineering tools are able to help extract useful information, but the output usually lacks semantic information like variable names and comments. Third, LLMs might suffer from hallucinations, especially with longer input. This issue is orders of magnitude worse in binary analysis, where the lack of context and semantic clues makes it harder for LLMs to interpret the actual behavior and vulnerabilities of the code reliably. Finally, binary files are usually very large, posing challenges for the limited context window of LLMs. As a result, as far as we know, state-of-the-art LLM-assisted binary analysis solutions are usually evaluated with small synthetic datasets, and cannot handle real-world binaries.

To address these challenges, we propose `Vul-BinLLM` , an end-to-end LLM-powered binary vulnerability analyzer. As far as we know, `Vul-BinLLM` is a LLM-assisted binary vulnerability analysis tool that detects vulnerabilities from compiled binaries. We achieve this goal by effectively allowing the LLM's to analyze a binary file that far exceeds its context window and attaching a function analysis queue that reduces hallucinations in the memory management agent to ensure complete analysis of the binary file.

`Vul-BinLLM` employs a structured workflow to assess and optimize the binary analysis process iteratively, emulating the approach of a security expert. It features an optimized decompiler that enhances decompiled code by appending vulnerability-specific comments, as well as prompt engineering to enhance vulnerability reasoning. By decomposing the traditional binary analysis workflow, `Vul-BinLLM` integrates insights from the extensive decompiled source code, offering an effective memory management approach to improve LLM analysis. In order to reduce hallucinations from processing of the code within the limited context window, we attach a function queue with the LLM's extended memory.

We evaluate `Vul-BinLLM` on binaries on commonly used synthetic vulnerability datasets i.e Juliet. `Vul-BinLLM` is able to beat the state-of-art tools and frameworks for stripped synthetic data binaries. VulBinLLM can detect CWEs for Juliet with appropriate information to justify the presence of such vulnerability in code.

We summarize our contributions for building a LLM-Powered binary analysis framework as follows:

- We build an LLM-assisted binary vulnerability analysis tool tailored for CWE detection on decompiled files by integrating a memory management system and a function analysis queue, enabling it to analyze complex binaries.
- With our analysis, we provide insights into how LLMs can analyze vulnerabilities on the binary level without much contextual knowledge and be able to reconstruct them for vulnerability detection.
- With our evaluations, we achieved approximately 10% increased accuracy in detecting stripped synthetic code vulnerabilities.

## 2 Background

### 2.1 Reserve Engineering for Vulnerability Analysis

Reverse engineering plays a critical role in vulnerability analysis, malware detection, and overall system security. In system security, reverse engineering refers to the process by which an analyst examines a binary executable to recover the design and implementation details necessary for understanding the program's functionality. This process can be applied in various contexts, such as malware analysis, vulnerability discovery, or firmware analysis, with the specific output varying according to the context. However, the core goal remains the same: reconstructing the program logic and identifying the conditions required to reach specific code locations, which could reveal bugs or suspicious behaviors, especially in the case of malicious binaries.

While automation has significantly advanced in areas like host-based and network-based attack detection, malware classification, and phishing detection, binary reverse engineering is still primarily driven by highly skilled human experts. Although tools for unpacking, disassembly, emulation, and binary similarity comparison assist in the process, the task of fully understanding the code remains a predominantly human effort. This manual work demands a deep level of expertise and is both time-consuming and labor-intensive. Unfortunately, the shortage of expert reverse engineers is a bottleneck, especially considering the increasing volume of software that requires analysis.

Initiatives like the DARPA Cyber Grand Challenge (CGC) have pushed forward the development of autonomous systems capable of analyzing binaries and identifying vulnerabilities. Despite these advancements, current automated solutions still struggle to match human-expert level depth of reasoning and intuition in reverse engineering tasks. As such, reverse engineering continues to be a crucial component of software vulnerability discovery, relying heavily on a combination of human skill and machine assistance.

### 2.2 Binary Analysis and Program Analysis

Binary analysis can be broadly categorized into dynamic and static binary analysis. Dynamic analysis examines program behavior during execution, while static analysis involves examining binary code without executing it. In this paper, unless otherwise noted, we focus exclusively on static binary analysis.

Traditional static binary analysis begins with binary acquisition and format inspection, followed by systematic layers of examination, including disassembly, control flow analysis, and data flow analysis. Disassembly and decompilation converts binary code into human-readable assembly instructions or source code. Control flow analysis maps the potential execution paths by identifying decision points and jumps. Data flow analysis tracks data movement and transformations to uncover vulnerabilities such as buffer overflows or memory leaks. Additional techniques like call graph analysis, help visualize relationships between functions and identify entry and exit points. Symbolic execution explores all possible execution

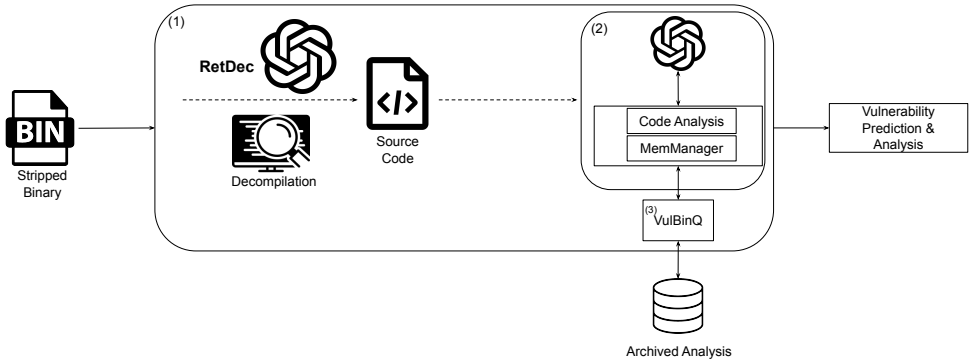


**Figure 1: Workflow of `Vul-BinLLM` : (1) Binary files are decompiled into source code, where an LLM-assisted decompiler enriches the code with contextual information for vulnerability detection. (2) The decompiled source code is then analyzed by `Vul-BinLLM` for vulnerability analysis which has an archival storage to store analysis, The analyzer then provides a comprehensive vulnerability detection result (3) VulBinQ: It features an additional queue that manages the functions that are to be analyzed using `Vul-BinLLM` and serves as middleware between the Archived Analysis and `Vul-BinLLM` .**

paths by using symbolic rather than concrete values, uncovering potential vulnerabilities missed by traditional methods. Static analysis tools also commonly incorporate pattern matching to detect known vulnerabilities, which can accelerate the analysis process. However, this comprehensive analysis usually requires significant expertise and extensive training.

```
// Function to multiply two matrices
// with intentional overflow
void buffer_overflow_matrixmul(long m1[R1][C1],
                    long m2[R2][C2],
                    long result[R1][C2])
{
    for (int i = 0; i < R1; i++) {
        for (int j = 0; j < C2; j++) {
            result[i][j] = 0;
// Initialize the result matrix element
            for (int k = 0; k <= C1; k++) {
// Intentional off-by-one error causing overflow
                result[i][j] += m1[i][k] * m2[k][j];
// This will read and write out of bounds when k == C1
            }
        }
    }
}
```

**Figure 2: An example of a buffer overflow vulnerability: the detection of this vulnerability relies on human expertise in security review. Human experts are able to detect the above vulnerability, but the subtlety of the vulnerability may lead to difficulty in detection by an LLM.**

## 2.3 CVE and CWE

Existing vulnerability classification systems, such as Common Vulnerabilities and Exposures (CVE) [64] and Common Weakness Enumeration (CWE) [65], provide standardized frameworks for categorizing and managing security issues. CVE is a system used to describe and report specific vulnerabilities in real-world applications (e.g. Google Chrome, Android), with each reported vulnerability assigned a unique identifier (CVE ID). A single CVE ID may correspond to multiple distinct code snippets representing different instances of the same vulnerability. CWE serves as a classification system that organizes common software and hardware security weaknesses. Each weakness type is assigned a unique identifier (CWE ID), providing a broad taxonomy of security flaws. Code behaviors within the same CWE category can vary significantly. For instance, CWE-416 (Use After Free) [69] denotes improper use of memory after it has been freed, which may result from issues such as race condition mismanagement (e.g., CVE-2023-30772 [68]) or incorrect reference counting, leading to premature object destruction (e.g., CVE-2023-3609 [67]). While CWE offers a higher-level understanding of weaknesses, CVE provides specific instances of vulnerabilities in real-world systems. CWE and CVE are widely used in the standardized approach for categorizing and managing security issues.

| Datasets | # of CWEs | # of Test Cases |
|---|---|---|
| Juliet C/C++ [7] | 118 | 90,000+ |
| SARD [11] | 200+ | 100,000+ |
| Big-Vul [3] | 10+ critical CWEs | 6,800 |
| Devign [4] | 5+ memory-related CWEs | 7,000+ |
| REVEAL [10] | 10+ memory safety CWEs | 3,000+ |
| MVD [8] | 10+ critical CWEs | 1,000+ |
| Vul4J [12] | 10+ Java-specific CWEs | 500+ |

**Table 1: Overview of widely-used CWE datasets in vulnerability detection. Juliet C/C++ and SARD each contain over 100 CWEs for in-depth testing in C/C++ and Java. NVD includes over 150 CWEs linked to public CVEs. Big-Vul offers critical vulnerabilities from open-source projects. Devign and REVEAL focus on memory-related weaknesses in C/C++. MVD and Vul4J cater to critical and Java-specific vulnerabilities.**

# 3 Method

## 3.1 Overview

Vul-BinLLM breaks the problem down into two steps: 1) the reconstruction of information optimized for vulnerability detection from binary files, 2) the analysis of decompiled code to infer vulnerabilities in terms of CWEs. In particular, the second step poses a unique challenge when the decompiled code exceeds the context window for the LLM, leading to an inability for vulnerability analysis of a binary file. To further enhance the problem statement we present the following Research Questions (RQ) to analyze: **RQ1:** Does LLM-powered human-readable code restoration for functionality description allow for vulnerability detection using LLMs?

**RQ2:** Can we optimize the restoration to make the vulnerability features more prominent such that the LLM's ability to detect vulnerabilities is improved?

To address these RQs, we propose Vul-BinLLM , a LLM-powered binary analysis framework. Vul-BinLLM is the first framework that focuses on recovering syntactic information to highlight vulnerable features using LLMs.

In this section, we present the methodologies behind Vul-BinLLM . It employs a structured workflow to first generate syntactic information from the binary code, making the vulnerable features more prominent within the source code, to allow the LLM to detect vulnerabilities using this syntactic information. Then, it iteratively assesses the source code with an extended memory structure to analyze each function in accordance to the control flow, emulating the approach of a security expert. By decomposing the traditional binary analysis workflow, Vul-BinLLM integrates insights from the decompiled source code by embedding appropriate information during the decompilation stage and then utilizing it for vulnerability detection, offering an integrated approach to improve vulnerability analysis. Detailed discussions are provided in Section 3.2.

First, we summarize the challenges of applying LLMs to binary analysis and describe how Vul-BinLLM addresses them:

- **Feature Definition Code Generation:** Binary code, especially in stripped binaries, lacks the high-level abstractions and semantic information present in source code. This makes it difficult for LLMs, which are primarily trained on text and source code, to understand the underlying logic and identify potential vulnerabilities.
- **Data Scarcity:** Training effective LLMs for binary vulnerability detection requires large datasets of labeled binary code with known vulnerabilities. However, such datasets are relatively scarce compared to source code datasets, which can hinder the development and evaluation of LLM-based approaches.
- **Limited Contextual Awareness:** LLMs may struggle to understand the broader context in which a particular code segment operates. This can lead to inaccurate vulnerability assessments, as the model may not fully grasp the implications of a specific code pattern within the larger program.

## 3.2 Design of Vul-BinLLM

The overview of Vul-BinLLM is illustrated in Figure 4. The key insight behind Vul-BinLLM is to emulate the traditional binary analysis workflow while optimizing each step with LLM-powered enhancements. In conventional binary analysis, binaries are first disassembled and decompiled into source code or intermediate representations (IR) to then be analyzed using various static analysis techniques to detect vulnerabilities in the binary as the source code and IR represent the same functionality but are not easily understandable by a human. We integrate an LLM-powered workflow to assist in tradition binary analysis vulnerability detection workflow which allows for a better analysis of vulnerable binaries Vul-BinLLM approaches these challenges using neural decompilation to recover high-level, vulnerability-related syntactic information from binary code, enabling LLMs to better understand program logic and identify potential vulnerabilities. We incorporate LLM models that analyze the decompiled code in a context-sensitive manner, considering the relationships between different code segments and the overall program structure, by utilizing an extended context window which serves as an archival storage for the analysis of vulnerabilities in functions within the source code. The archival storage is a SQL Database which allows the LLM Agent to store and fetch the summarized analysis of different functions within the source code file. We also emulate a queue within Vul-BinLLM to ensure the coverage of all functions within a binary.

*3.2.1* Vul-BinLLM - *Vulnerability Prominence.* Neural decompilation acts as a bridge between the low-level world of binary code and the high-level understanding of LLMs. By recovering source code from stripped binaries, it unlocks the potential for LLMs to analyze and comprehend the program's logic, paving the way for effective vulnerability detection. This recovered source code, however, is not merely a verbatim translation of the binary. It is optimized specifically for vulnerability detection, with embedding the key features and potential security flaws highlighted for LLMs to focus on.3 This optimization process involves identifying and emphasizing code patterns that are commonly associated with vulnerabilities. These patterns can include dangerous function calls, such as those known to be susceptible to buffer overflows or injection attacks, as well as code constructs that often lead to memory corruption or logic errors. By highlighting these vulnerability-related features, the recovered source code becomes a more informative and targeted input for LLMs, guiding their analysis towards potential security flaws. Furthermore, the optimization process can involve incorporating contextual information into the recovered source code. This can include information about overall program structure, the relationships between different code segments, and the intended functionality of the program. By providing LLMs with this broader context, they can better understand the implications of specific code patterns and make more accurate vulnerability assessments. LLMs, with their ability to learn complex patterns and representations from data, can then analyze the code and identify potential vulnerabilities based on the highlighted features and the overall code structure. This approach combines the strengths of both neural decompilation and LLMs, enabling the detection of vulnerabilities

[1]For the buffer overflow weakness, we included the vulnerability by not setting the bound for the array

| Models | Creator | # Parameters | Modality | Max Tokens | Function | Vul-1 [1] | Vul-2 |
|---|---|---|---|---|---|---|---|
| GPT-4 [2] | OpenAI | 1.7T | text | 32K | ✓ | ✓ | ✗ |
| ChatGPT [77] | OpenAI | 175B | text | 16K | ✓ | ✓ | ✗ |
| Claude 3.5 [19] | Anthropic | 175B | text | 32K | ✓ | ✓ | ✗ |
| Gemini [96] | Google | 500B | text | 32K | ✓ | ✗ | ✗ |
| CodeLlaMA [1] | Meta AI | 13B | text & code | 100K | ✓ | ✓ | ✗ |

**Table 2: Our study on LLMs understanding of decompiled binary vulnerable code: Vul-1 is a buffer overflow weakness in our defined `matrixmul` function, Vul-2 is a standard CWE-78 (OS Command Injection) in Juliet dataset. We compiled both of them to binary ELF files and analyze the decompiled code through LLMs. No LLM can find OS Command Injection intuitively from binary level but some of them can understand simple code (e.g. `matrixmul`) with synthetic weakness. NOTE: The 100k context window for CodeLlaMA requires intense computational power to process it.**

in stripped binary files that were previously difficult to analyze with traditional methods. In essence, vulnerability feature optimized source code recovery transforms binary code into a format that is both understandable and actionable for LLMs. It bridges the gap between the low-level representation of binary code and the high-level reasoning capabilities of LLMs, enabling the efficient and effective detection of vulnerabilities in stripped binary files. Custom data types and user-defined classes are generally outside the scope of these tools, leading to inaccurate or incomplete representations. For instance, the decompiled output of a simple *matrixMultiply()* function may display significant discrepancies in variable names and introduce overly complex or incorrect data structures. Furthermore, comments from the original source code are lost during decompilation since traditional reverse engineering tools cannot reconstruct them. Given our goal of using LLMs to analyze decompiled source code, preserving meaningful variable names, clear code structures, and informative comments is essential for enhancing the model's comprehension and processing capability. Several recent approaches, such as DeGPT [48] and ReSym [109], leverage LLMs to improve decompilation outputs. However, these methods are not tailored for vulnerability detection, limiting their effectiveness in security-focused analysis. In order to make the decompilation efficient to be analyzed by LLMs, we use an LLM to first do a syntactical recovery of comment, structure and variable information from the decompiled code from RetDec which is then utilized by the detection agent with an extended context window to analyze the program state with respect to the syntactically descriptive code. In this work, we specifically focus on optimizing decompilation for identifying vulnerabilities alongside extending the LLMs capability to evaluate binaries, addressing the unique challenges in this domain.

**Prompt Engineering**. We utilized in-context learning and few-shot Chain-of-Thought (CoT) [105] and in-context-learning [31] prompting to enhance LLMs' capability in identifying potential vulnerabilities. In-context learning enables LLMs to understand complex reasoning tasks by exposing them to examples of similar vulnerability patterns. This approach is particularly beneficial for vulnerability detection, as it allows the model to adapt to specific security weaknesses through example-based guidance. Few-shot CoT further decomposes complex reasoning into smaller, logical steps, prompting the LLMs to analyze code snippets by examining functionality, root causes, and potential impacts. This structured approach mirrors the systematic methods employed by security experts, allowing the model to connect multiple dimensions of a vulnerability. CoT is especially effective in uncovering hidden risks by providing a multi-faceted analysis aligned with expert practices.

**Construct Knowledge Documents**. Several datasets, such as Devign [118], BigVul [46], and CodeSearchNet [34], are commonly used to benchmark vulnerability detection performance. PairVul [32] is a unique resource containing pairs of vulnerable and patched code samples. The memory management agent is connected to the VulBinQ, which is responsible for managing the functions that are to be analyzed by the LLM and various knowledge documents are created from the binary code to then be placed into the archived analysis data store allowing for a clearer representation of the binary code for the LLM.

## 4 Evaluation

In this section, we evaluate the performance of `Vul-BinLLM` with an existing state-of-the-art approach to analyze each part's effectiveness.

### 4.1 Implementation

The implementation of `Vul-BinLLM` is built upon the reverse engineering frameworks Ghidra [5] and [9]. The `Vul-BinLLM` decompilation utilizes the large language model GPT-4o to enhance vulnerability detection by carefully embedding the vulnerability information into the decompiled code. Below, we outline the implementation process. First, the binary is loaded and analyzed using Ghidra plugins. Ghidra facilitates the extraction of decompiled code, control flow graphs, import tables, stack frames, and other relevant information for subsequent analysis. Through the API of GPT-4.0 [77], `Vul-BinLLM` identifies weaknesses in the binary and augments the output by (1) appending comments about potential vulnerabilities and functionality, (2) simplifying code structures, and (3) renaming variables for clarity and to make the vulnerability features more prominent.

Following these enhancements, `Vul-BinLLM` performs binary vulnerability classification using GPT-4o [78], leveraging vulnerability descriptions from MITRE for CWEs and then utilizes a extended context window approach with a shared contextual memory that

**Source code**

```
void multiplyMatrix(long m1[R1][C1], long
m2[R2][C2])
{
 // C1 and C2 are dimension of matrix m1 and m2
    int result[C1][C2];
    for (int i = 0; i < R1; i++) {
        for (int j = 0; j < C2; j++) {
            result[i][j] = 0;
            for (int k = 0; k < R2; k++) {
                result[i][j] += m1[i][k] *
m2[k][j];
            }
        }
    }
}
```

**Ghidra**

```
void multiplyMatrix(long param_1, long param_2)
{
    local_10 = *(long *)(in_FS_OFFSET + 0x28);
    for (*(long *)local_34 = 0; *(long *)local_34 < R1; local_34++) {
        for (local_30 = 0; local_30 < C2; local_30++) {
            (auStack_28)[(long)local_34 * C2 + (long)local_30] = 0;
            for (local_2c = 0; local_2c < R2; local_2c++) {
                auStack_28[(long)local_34 * C2 + (long)local_30] +=
                    *(int *)((long)local_2c * 0xc + param_2 +
(long)local_30 * 4)*
                    *(int *)((long)local_34 * 8 + param_1 +
(long)local_2c * 4);
            }
        }
    }
}
```

**Ghidra+ GPT-4o**

```
void multiplyMatrix(long matrix1, long matrix2) {
    long saved_offset  = *(long *)(in_FS_OFFSET + 0x28); // Preserves frame pointer, if necessary
    int result[R1][C2] = {0};  // Initialize result matrix
                                                                Text
    // Iterate over rows and columns to perform matrix multiplication
    for (int i = 0; i < R1; i++) {
        for (int j = 0; j < C2; j++) {
            result[i][j] = 0;
            for (int k = 0; k < R2; k++) {
                // Calculate addresses in the input matrices and accumulate the result
                int value1 = *(int *)(matrix1 + i * 8 + k * 4);    // Access element from matrix1
                int value2 = *(int  *)(matrix2 + k * 12 + j * 4); // Access element from matrix2
                result[i][j] += value1 * value2;
            }
        }
    }
}
```

**VulBinLLM**

```
void multiplyMatrix(long input_matrix_1, long input_matrix_2)
{
    // Initialize a matrix to store the result of matrix multiplication
    int result_matrix[row_1][column_2] = {0};

    // Iterate over each row and column of the result matrix
    for (int current_row = 0; current_row < row_1; current_row++) {
        for (int current_col = 0; current_col < column_2; current_col++) {
            // Perform matrix multiplication by iterating over the shared dimension
            for (int shared_dim = 0; shared_dim < row_2; shared_dim++) {
                // Vulnerability: Potential pointer arithmetic issue if column_1 and row_2 are not properly allocated
                // Multiply corresponding elements from the input matrices and accumulate the result
                result_matrix[current_row][current_col] += *((int *)(input_matrix_2 + shared_dim * 12 + current_col * 4))
* *((int *)(input_matrix_1 + current_row * 8 + shared_dim * 4));
            }
      }
}
```

**Figure 3: An example of `Matrix Multiplication` decompilation output across different stages: the original source code, Ghidra, Ghidra enhanced with GPT-4o, and `Vul-BinLLM` . The Ghidra decompilation provides a low-level representation with generic variable names and lacks context, making the functionality and security aspects harder to interpret. Ghidra + GPT-4o improves readability with meaningful variable names and clarifying comments. `Vul-BinLLM` further augments the output by adding vulnerability-specific annotations, such as warnings about potential pointer arithmetic issues that could lead to buffer overflows or memory access vulnerabilities. This layered enhancement helps bridge the gap between decompilation and security analysis, making `Vul-BinLLM` particularly beneficial for identifying and understanding vulnerabilities in binary code.**

allows for agents to reason about vulnerabilities in the decompiled code. Utilizing this agentic unbounded context window along with prompt templates and advanced prompt engineering techniques alongside a function queue, `Vul-BinLLM` generates structured prompt sequences to guide GPT-4o in vulnerability inspection. GPT-4o processes code snippets in the context of these instructions,

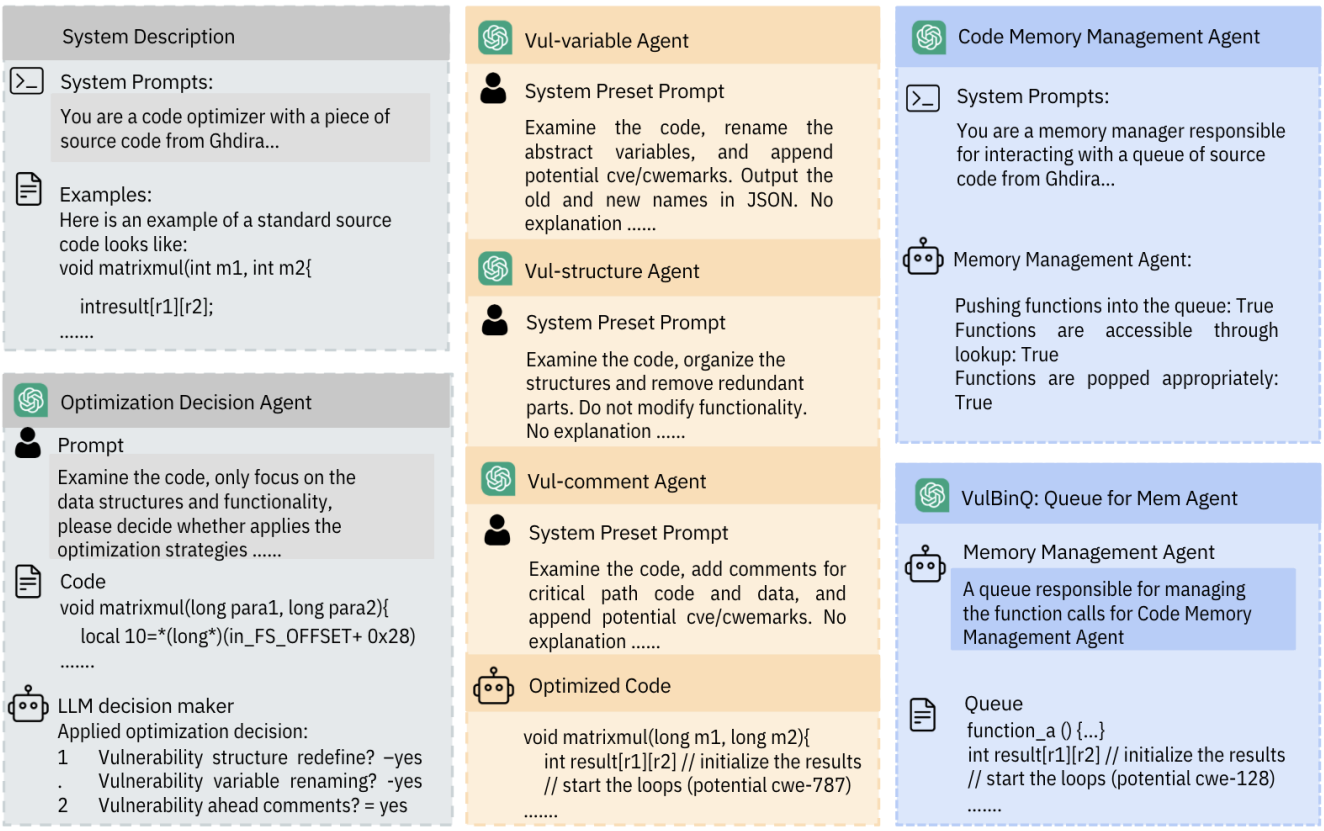


**Figure 4: `Vul-BinLLM` -Decompiler overview. `Vul-BinLLM` -decompiler includes an Optimization Decision Agent and three Action Agents (Vul-variable, Vul-struct, Vul-comment). After getting raw decompilation output from reverse engineering tool, `Vul-BinLLM` -decompiler will perform an initial check on the grammar, functionality, and structures and decide what optimization decision will be made. Be sending requests to Action Agents, `Vul-BinLLM` -decompiler will focus on renaming the variables and functions' names, reorganize the defined code structure, and critically, appending explanations on potentially vulnerability and functionalities attached the code. `Vul-BinLLM` -decompiler can also support to learn examples code by in-context learning**

performing detailed analysis to identify potential vulnerabilities in the binary.

## 4.2 Research questions and evaluation setup

In order to address the research questions described above in section 3, we evaluate `Vul-BinLLM` on binaries from the Juliet dataset, then compare the accuracy of the detected vulnerability on unstripped versions of these binaries. We also analyze how accuracy is affected with `Vul-BinLLM` 's extended context window approach for smaller binaries.

We evaluate the accuracy of `Vul-BinLLM` 's accuracy by the following methods:

- How accurate is LLMs in detecting vulnerabilities in the stripped and unstipped binaries?
- Can we extend the LLM's capability to analyze binaries that far exceed it's context window?
- Can we utilize LLMs to detect vulnerabilities in the stripped synthetic test suits and compare it with state-of-the-art LLM powered tools?

In order to formulate our results we take two approaches for binary vulnerability analysis where the LLMs are prone to hallucinations and might require additional analysis for verification. We benchmark our system on the decompiled code from Juliet Test Suite.

## 4.3 Results

We evaluated `Vul-BinLLM` on CWE classification on binary decompiled code. For the evaluation of `Vul-BinLLM` , we utilized stripped Juliet Test Suite binaries. We use the following **Dataset:**

- **Juliet Test Suite (v1.3) [66]:** This C/C++ vulnerability test suite is organized by CWE categories and includes vulnerabilities relevant to our study (e.g., CWE-78, CWE-134, CWE-190, CWE-606). For evaluation, we compiled the test cases into binaries, removing debug information and

symbol tables to simulate real-world scenarios. Test cases involving constant values were excluded, as LATTE targets vulnerabilities introduced by external inputs. Using GCC as our compiler, we generated over 20,000 binary samples. Table 2 summarizes the number of test cases used in our analysis.

**Evaluations on Synthetic Dataset:** For the evaluations on sythetic dataset (Juliet), we compare `Vul-BinLLM` against LATTE [57], the state-of-the-art approach in utilizing LLMs for binary taint analysis. LATTE enhances vulnerability detection precision by integrating flow analysis with prompt-engineered LLM responses, ensuring consistency and reliability across multiple vulnerability categories but is limited to a certain class of CWEs that exist in binary files which are subject to be detected by binary taint analysis. We provide our results in Table 3 where we compare `Vul-BinLLM` 's abilities with LATTE.

**CWE Classification in `Vul-BinLLM`** In `Vul-BinLLM` 's CWE classification task, we prompt GPT-4o to respond with the appropriate vulnerability (e.g., `CWE-78: OS Command Injection`) to determine the presence of a specific vulnerability in the optimized, neurally decompiled binary code. As shown in the Figure 5, we provide the model with a code snippet alongside a list of descriptions for potential weaknesses, such as OS Command Injection, Stack-based Buffer Overflow, and Out-of-bound Write. These descriptions are sourced directly from authoritative references like the MITRE CWE database, ensuring accuracy in the model's understanding of each vulnerability type.

To minimize the risk of memorization, where the model might rely on learned patterns rather than performing genuine analysis, we include multiple CWEs in each query. By appending multiple vulnerability types and directing GPT-4o to focus on particular aspects—like data flow in this example—we guide the model to discern the nuanced characteristics of each potential weakness. This approach leverages extended context solutions with a stack that contains the decompiled source code functions to then combine contextual information with targeted prompt engineering, enabling `Vul-BinLLM` to make informed vulnerability assessments within binary code.

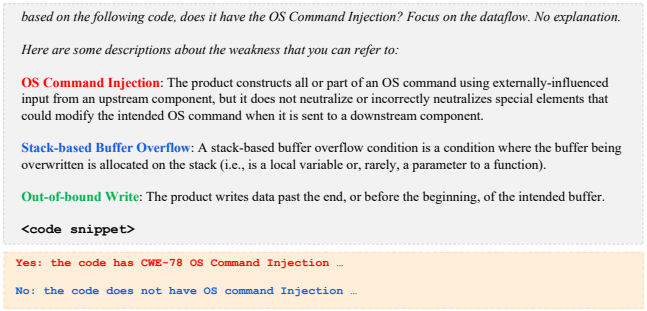


**Figure 5: An example of binary classification with CWE-78. The LLM is required to respond with `yes` or `no`, when asked if it is concentrating on the code flow rather than semantics. To avoid memorization of LLMs in our special case, we append multiple CWEs (CWE-121: Stack Buffer Overflow, CWE-787: Out-of-bound Write)**

## 5 Discussions

Despite the promising results demonstrated by `Vul-BinLLM` , certain limitations remain First, due to the scarcity of binary vulnerability datasets, we evaluated `Vul-BinLLM` on compiled data from the Juliet test suite. Although popular source code datasets such as BigVul, Devign, and REVEAL are often used in vulnerability research, they are challenging to apply in binary analysis due to inconsistent compilation environments and build dependencies. For example, Devign consists of complex projects requiring specific Makefiles for compilation, while BigVul includes diverse projects with inconsistent toolchains for building vulnerable code. There's an ongoing effort to improve the neural decompilation process for binaries and to optimize the decompiled code to recover syntactic details out of the semantic information that is created by the decompiler [94] [109]. Another approach that can be pursued to extend the LLM's capability to analyze a binary through LLMs without having to go through the decompilation process is to directly detect vulnerabilities on the assembly level. The critical issue with such a process it the high syntactic similarity between two binaries for a single architecture. In our evaluations we evaluated various similarity metrics including cosine similarity and Levenshtein distance between two different CWE examples in X86 assembly representation from the Juliet Test Suite, we detected the cosine similarity to be approximately 98% for most of the CWEs in there even though the semantic information is very different for each of those files. It occurs due to the similarity among the two binaries in their assembly format referring to the same registers and the same architecture-specific assembly instructions. Another important field of research is to extend the LLM's capability to reason about complex tasks, especially vulnerability reports. The justification in CVE reports is provided by either a clear definition of the vulnerability or an exploit method that constitutes as a piece of sufficient evidence to show the existence of a vulnerability in the source code. However, in a binary describing a vulnerability would require the availability of an attack method that is then run to ensure the availability of such vulnerability in a binary. We consider the explainability of the reasoning of the vulnerability to be out of scope for this paper and a future direction. We also lack a comprehensive classification model for the definition of vulnerabilities in the binary level. For source code vulnerability classification, we have CVE and CWE definitions which provide both a fine and coarse-grained classification for vulnerabilities which is not defined for architecture-specific binary vulnerabilities. Another future direction for `Vul-BinLLM` is to explore the areas of formal specification, probabilistic inference mechanisms, retrieval augmented generation, small language model alongside a descriptive analysis of architecture specific vulnerability definition as future directions to advance the binary analysis ecosystem and to introduce LLM-like solutions to assign security experts to potentially analyze binaries for vulnerabilities and malware.

| | CWE-78 (960/960)+ | | CWE-134 (1200/1200)+ | | CWE-190 (2860/2860)+ | | CWE-606 (240/240)+ | |
|---|---|---|---|---|---|---|---|---|
| | LATTE | Vul-BinLLM | LATTE | Vul-BinLLM | LATTE | Vul-BinLLM | LATTE | Vul-BinLLM |
| TP | 892 | 1055 | 1151 | 1345 | 1773 | 1725 | 210 | 1218 |
| FN | 68 | 0 | 49 | 0 | 1087 | 0 | 30 | 0 |
| TN | 960 | 4288 | 1102 | 3998 | 1779 | 3618 | 142 | 4125 |
| FP | 0 | 191 | 98 | 14 | 1081 | 35 | 98 | 416 |
| Accuracy | 96.46% | **96.55%** | 93.88% | **99.74%** | 62.1% | **99.34%** | 73.33% | **74.54%** |
| Precision | **100%** | 84.67% | 95.92% | **98.97%** | 52.04% | **98.01%** | 68.18% | **92.78%** |
| F1 Score | **96.33%** | 91.70% | 93.99% | **99.48%** | 62.05% | **98.99%** | 74.24% | **85.4%** |

**Table 3: Evaluation results of Vulnerability inspection based on Juliet Testsuite.**

+:LATTE evaluation is done by embedding additional information alongside the code of the stripped binary to detect the CWE in the test suite.
+:The numbers in parentheses indicate the number of test cases (bad/good) for this vulnerability type.
+:The values for Vul-BinLLM represents the number of files which contained various (bad/good) vulnerability types.
+:TN are all the files such that the vulnerability was in the prompt but was correctly undetected by the LLM.

# 6 Related Work

## 6.1 Semantic Recovery from Binary files

While LLMs have shown promise in various software engineering tasks, their application in binary analysis has primarily focused on syntactic regeneration of code. Previous research works [109] [48] has explored using LLMs to generate human-readable representations of binary code, such as recovering function names or providing code summaries. These approaches leverage the LLMs' ability to learn patterns and structures in code to reconstruct higher-level representations from the low-level binary code. However, these efforts primarily aim to improve code understandability for human analysts rather than directly detecting vulnerabilities. They focus on regenerating the syntactic structure of the code without necessarily delving into the identification of potential security flaws. This limitation stems from the inherent challenges of binary code analysis, where the absence of high-level abstractions and semantic information makes it difficult to extract vulnerability-related features directly from the binary.

## 6.2 Reverse Engineering and Program Analysis for Vulnerability Detection

Reverse engineering is the process of analyzing and understanding the design, structure, and functionality of a code snippet by working backward from its final form. It is a critical technique in software engineering particularly within binary analysis. Traditional binary analysis is involved in decompilation, which is a significant portion in reverse engineering. Reverse engineering has been adopted in many program analysis domains, e.g. vulnerability assessment [18, 26, 44, 51, 82, 99], malware analysis [24, 38, 103, 112], software repair [56, 80, 84, 98], and code optimization [62].

Due to the inherent difficulty and opacity of binary code, prior research has focused on the readability and maintainability. Similarity analysis [102, 110], memory analysis [81], assembly-to-code translation [13, 40], function identification [52]. Decompilation optimization is an active research area since it combines reverse engineering and machine learning. David et al. [28] leverage LSTM [47] to predict variable names in stripped binaries. TIE [53], Retyped [75], and OSPREY [115] focus on variable type recovery. Direct [74] and DIRTY [23] have exploited Transformer-based methods for recovering variable names from decompiled code. More recently, DeGPT [48] and Resym [109] leverage LLMs to identify, and recover code that is readable and similar to ground-truth original code. Vul-BinLLM , on the other hand, emphasizes on annotating potential vulnerability information while simplifying code structures and renaming variable names, while prior work [22, 45] employ ML-based approaches to predict debugging information from stripped binaries. Vul-BinLLM can generate more concrete vulnerability comments to provide appropriate information for the LLM to analyze it for vulnerability detection.

## 6.3 Learning-based Vulnerability Detection

Learning-based approaches have made substantial progress in vulnerability detection [21, 37, 43, 63, 89, 118], employing various methods such as graph representations and large language models to analyze code. Graph Neural Network (GNN)-based techniques [63, 95, 118] represent code snippets as graph-based structures, such as Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs). By utilizing GNN on these graph based structured representations, It allows for a better analysis for vulnerabilities in the representation of control flow graphs, call graphs and code property graphs. However, GNN-based methods face challenges in effectively distinguishing vulnerabilities within lexically-similar but semantically-different code pairs, as they are limited by their reliance on structural information alone, often lacking deeper semantic understanding required to identify subtle vulnerabilities. [32]

Other learning-based approaches have explored the use of transformer-based models, where Large Language Models (LLMs) have been employed to detect vulnerabilities by leveraging large-scale pre-training on code [59, 83]. Although LLMs have shown promise in generating syntactically correct and contextually relevant code, their effectiveness in identifying vulnerabilities is limited when they encounter complex low-level vulnerabilities or intricate

program flows, which require precise interpretation of code semantics and memory handling [117].

In addition, traditional static analysis tools, such as Cppcheck, offer rule-based analysis to detect code issues, including potential vulnerabilities. While these tools are efficient and interpretable, they often suffer from high false-positive rates and lack the flexibility to detect novel or context-dependent vulnerabilities, as they cannot generalize beyond predefined rules.

PLM-based Vulnerability Detection involves fine-tuning pre-trained language models (PLMs) on vulnerability detection datasets. In this approach, code snippets are tokenized and processed by PLMs (e.g., RoBERTa [58]), which act as encoders. The PLMs extract semantic features from the code, which are then used in binary classification to determine the presence of vulnerabilities. This method leverages the language understanding capabilities of PLMs to analyze code textually, making it suitable for detecting vulnerabilities that are evident from code semantics.

LLM-based Vulnerability Detection utilize LLMs through either prompt engineering or fine-tuning. Prompt engineering strategies, such as Chain-of-Thought (CoT) [105] reasoning and few-shot learning, enable LLMs to detect vulnerabilities more accurately without modifying the original model parameters. In contrast, the fine-tuning approach involves training LLMs on vulnerability detection datasets, allowing the models to learn specific features of vulnerable code by updating their parameters. This approach takes advantage of LLMs' strong contextual understanding, making it well-suited for identifying complex vulnerabilities across diverse codebases.

### 6.4 Code Large Language Models (CodeLLMs)

With the power of Transformer architecture [100], modern large language models (LLMs) show great potential in code related tasks. Specifically, for encoder-only models (e.g., BERT [30], RoBERTa [58]), Code-BERT [35] learns from massive source code and natural language descriptions to show promising results in code search, completion, and summarizations. GraphBert [114] can learn representations from graph-structured code to leverage structural information. There are also followed-up work based on them Code-BERT [35]. Furthermore, for decoder-only models (e.g., GPTs [20, 85]), GPT-4 with canvas [2], CodeLLaMA series [33, 86, 97], Claude Sonnet series [19], Mistral [17]& Codestral [15], DeepSeek Code [42], and Gemini [96] enhance the capabilities of understanding code related tasks specifically. Recently, more emerging models like Codestral Mamba [16] offer the advantage of linear time inference [27, 41] and the theoretical ability to model sequences of infinite length. However, those Code LLMs are focus on code generation, debugging, and summarization [14, 36, 42, 55, 73, 86, 104]. Our work focuses on code analysis especially in vulnerability or weakness.

There are prior works applying LLMs in code understanding [25, 61, 71, 76, 87], software fuzzing [29, 50, 113], natural language alignment [39, 70, 72, 108], vulnerability repair [32, 49, 90, 106, 107]. There are few works in applying LLMs in binary analysis. LATTE [57] is the most relevant works. However, it focuses on specific taint analysis. Thus, we treat is as a complement to our work. Recent work in binary-related work [54, 60, 91, 92, 111] are all based on traditional machine leaning methods rather than LLMs. For example, CodeArt [92] pretrains a BERT-like model on binary functions through explicit attention mechanisms. VulHawk leverages an intermediate-representation by RoBERTa with code reuse similarity. VulANalyzeR exploits Gragh Convolution and attention mechanisms to classify vulnerabilities from Control Flow Graphs. Note that DeBinVul is an orthogonal work to ours. They fine-tuned a Code LLM to detect vulnerabilities from decompiled code with self-built datasets. On the other hand, `Vul-BinLLM` is able to detect vulnerabilities without fine-tuning. Further, `Vul-BinLLM` is easy to scale to multiple programming languages and have a better generalization due to its flexibility.

## 7 Conclusions

This paper introduces Vul-BinLLM, an LLM-based framework for binary vulnerability detection that integrates decompilation optimization utilizing neural decompilation and extended context window and memory management capabilities to enable vulnerability analysis with binaries. The results demonstrate the potential of LLMs to address longstanding challenges in binary vulnerability detection, paving the way for more scalable and secure software systems.

## References


[1] 2023. *CodeLlaMA by Meta AI.* https://ai.meta.com/blog/code-llama-large-language-model-coding/
[2] 2023. *GPT-4 by OpenAI.* https://openai.com/research/gpt-4
[3] 2024. *Big-Vul Dataset.* https://huggingface.co/datasets/bstee615/bigvul
[4] 2024. *Devign Dataset.* https://github.com/epicosy/devign
[5] 2024. *Ghidra.* https://ghidra-sre.org/
[6] 2024. *IDA Pro.* https://hex-rays.com/ida-pro
[7] 2024. *Juliet Test Suite for C/C++ and Java.* https://samate.nist.gov/SARD/testsuite.php
[8] 2024. *Microsoft Vulnerability Dataset (MVD).* https://github.com/microsoft/MS-MVD
[9] 2024. *RetDec.* https://github.com/avast/retdec
[10] 2024. *REVEAL Dataset.* https://github.com/VulDetProject/ReVeal
[11] 2024. *Software Assurance Reference Dataset (SARD).* https://samate.nist.gov/SARD/
[12] 2024. *Vul4J Dataset.* https://github.com/tuhh-softsec/vul4j
[13] Iftakhar Ahmad and Lannan Luo. 2023. Unsupervised Binary Code Translation with Application to Code Clone Detection and Vulnerability Discovery. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 14581–14592.
[14] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).
[15] Mistral AI. 2024. *Codestral.* https://mistral.ai/news/codestral/
[16] Mistral AI. 2024. *Codestral Mamba.* https://mistral.ai/news/codestral-mamba/
[17] Mistral AI. 2024. *Mixture of Experts Models.* https://mistral.ai/news/mixtral-of-experts/
[18] Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele. 2023. {FirmSolo}: Enabling dynamic analysis of binary Linux-based {IoT} kernel modules. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5021–5038.
[19] anthropic. 2024. *Claude.* https://www.anthropic.com/claude
[20] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).
[21] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48, 9 (2021), 3280–3296.
[22] Ligeng Chen, Zhongling He, and Bing Mao. 2020. Cati: Context-assisted type inference from stripped binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 88–98.
[23] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*. 4327–4343.
[24] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. 2021. {SelectiveTaint}: Efficient Data Flow Tracking With Static Binary Rewriting. In *30th USENIX Security Symposium (USENIX Security 21)*. 1665–1682.

[25] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).

[26] Victor Cochard, Damian Pfammatter, Chi Thang Duong, and Mathias Humbert. 2022. Investigating graph embedding methods for cross-platform binary code similarity detection. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 60–73.

[27] Tri Dao and Albert Gu. 2024. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060* (2024).

[28] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

[29] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[30] Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[31] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

[32] Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2024. Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG. *arXiv preprint arXiv:2406.11147* (2024).

[33] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[34] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[35] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[36] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[37] Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 608–620.

[38] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 896–899.

[39] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2023. An empirical study on using large language models for multi-intent comment generation. *arXiv preprint arXiv:2304.11384* (2023).

[40] Redha Gouicem, Dennis Sprokholt, Jasper Ruehl, Rodrigo CO Rocha, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Risotto: a dynamic binary translator for weak memory model architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 107–122.

[41] Albert Gu and Tri Dao. 2023. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752* (2023).

[42] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[43] Hazim Hanif and Sergio Maffeis. 2022. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International joint conference on neural networks (IJCNN)*. IEEE, 1–8.

[44] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. 2024. Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection. In *33rd USENIX Security Symposium (USENIX Security 24), PHILADELPHIA, PA*.

[45] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1667–1680.

[46] Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1865–1879.

[47] S Hochreiter. 1997. Long Short-term Memory. *Neural Computation MIT-Press* (1997).

[48] Peiwei Hu, Ruigang Liang, and Kai Chen. 2024. DeGPT: Optimizing Decompiler Output with LLM. In *Proceedings 2024 Network and Distributed System Security Symposium (2024). https://api. semanticscholar. org/CorpusID*, Vol. 267622140.

[49] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1430–1442.

[50] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[51] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2022. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1661–1682.

[52] Soomin Kim, Hyungseok Kim, and Sang Kil Cha. 2023. Funprobe: Probing functions from binary code through probabilistic analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1419–1430.

[53] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).

[54] Litao Li, Steven HH Ding, Yuan Tian, Benjamin CM Fung, Philippe Charland, Weihan Ou, Leo Song, and Congwei Chen. 2023. VulANalyzeR: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution. *ACM Transactions on Privacy and Security* 26, 3 (2023), 1–25.

[55] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[56] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. Cctest: Testing and repairing code completion systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1238–1250.

[57] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhi Li, and Limin Sun. 2023. Harnessing the power of llm to support binary taint analysis. *arXiv preprint arXiv:2310.08275* (2023).

[58] Yinhan Liu. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* 364 (2019).

[59] Zhongxin Liu, Zhijie Tang, Junwei Zhang, Xin Xia, and Xiaohu Yang. 2024. Pre-training by Predicting Program Dependencies for Vulnerability Analysis Tasks. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[60] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search.. In *NDSS*.

[61] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems* 36 (2024).

[62] Niru Maheswaranathan, David Sussillo, Luke Metz, Ruoxi Sun, and Jascha Sohl-Dickstein. 2021. Reverse engineering learned optimizers reveals known and novel mechanisms. *Advances in Neural Information Processing Systems* 34 (2021), 19910–19922.

[63] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. 2023. {VulChecker}: Graph-based Vulnerability Localization in Source Code. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6557–6574.

[64] MITRE. 2024. *Common Vulnerabilities and Exposures.* https://cve.mitre.org/

[65] MITRE. 2024. *Common Weakness Enumeration.* https://cwe.mitre.org/

[66] MITRE. 2024. *Juliet C/C++ 1.3 v1.3.* https://samate.nist.gov/SARD/test-suites/112

[67] MITRE. 2024. *The webiste of cve-2023-3699.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-3609

[68] MITRE. 2024. *The website of cve-2023-30772.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-30772

[69] MITRE. 2024. *The website of cwe-416.* https://cwe.mitre.org/data/definitions/416.html

[70] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124* (2023).

[71] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[72] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005* (2022).

[73] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[74] Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. 2021. Direct: A transformer-based model for decompiled identifier renaming. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. 48–57.

[75] Matt Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 27–41.

[76] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying gpt self-repair for code generation. *arXiv preprint arXiv:2306.09896* (2023).

[77] OpenAI. 2024. *ChatGPT*. https://openai.com/blog/chatgpt

[78] OpenAI. 2024. *GPT-4o*. https://openai.com/index/hello-gpt-4o/

[79] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE symposium on security and privacy (SP)*. IEEE, 833–851.

[80] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

[81] Kexin Pei, Dongdong She, Michael Wang, Scott Geng, Zhou Xuan, Yaniv David, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. NeuDep: neural binary memory dependence analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 747–759.

[82] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. Learning approximate execution semantics from traces for binary function similarity. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2776–2790.

[83] Tao Peng, Shixu Chen, Fei Zhu, Junwei Tang, Junping Liu, and Xinrong Hu. 2023. PTLVD: Program Slicing and Transformer-based Line-level Vulnerability Detection System. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 162–173.

[84] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael Lyu. 2024. Domain knowledge matters: Improving prompts with fix templates for repairing python type errors. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[85] Alec Radford. 2018. Improving language understanding by generative pre-training. (2018).

[86] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[87] Jérémy Scheurer, Jon Ander Campos, Tomasz Korbak, Jun Shern Chan, Angelica Chen, Kyunghyun Cho, and Ethan Perez. 2023. Training language models with language feedback at scale. *arXiv preprint arXiv:2303.16755* (2023).

[88] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 138–157.

[89] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[90] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2237–2248.

[91] Zian Su, Xiangzhe Xu, Ziyang Huang, Kaiyuan Zhang, and Xiangyu Zhang. 2024. Source Code Foundation Models are Transferable Binary Analysis Knowledge Bases. *arXiv preprint arXiv:2405.19581* (2024).

[92] Zian Su, Xiangzhe Xu, Ziyang Huang, Zhuo Zhang, Yapeng Ye, Jianjun Huang, and Xiangyu Zhang. 2024. Codeart: Better code models by attention regularization when symbols are lacking. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 562–585.

[93] Yashar Talebirad and Amirhossein Nadiri. 2023. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *arXiv preprint arXiv:2306.03314* (2023).

[94] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling Binary Code with Large Language Models. *arXiv preprint arXiv:2403.05286* (2024).

[95] Wei Tang, Mingwei Tang, Minchao Ban, Ziguo Zhao, and Mingjun Feng. 2023. CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. *Journal of Systems and Software* 199 (2023), 111623.

[96] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[97] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[98] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *IEEE Symposium on Security and Privacy*.

[99] Jayakrishna Vadayath, Moritz Eckert, Kyle Zeng, Nicolaas Weideman, Gokulkrishna Praveen Menon, Yanick Fratantonio, Davide Balzarotti, Adam Doupé, Tiffany Bao, Ruoyu Wang, et al. 2022. Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs. In *31st USENIX Security Symposium (USENIX Security 22)*. 413–430.

[100] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[101] Fish Wang and Yan Shoshitaishvili. 2017. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 8–9.

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

[103] Junzhe Wang, Matthew Sharp, Chuxiong Wu, Qiang Zeng, and Lannan Luo. 2023. Can a Deep Learning Model for One Architecture Be Used for Others?{Retargeted-Architecture} Binary Code Analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7339–7356.

[104] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[105] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[106] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1282–1294.

[107] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[108] Danning Xie, Byungwoo Yoo, Nan Jiang, Mijung Kim, Lin Tan, Xiangyu Zhang, and Judy S Lee. 2023. Impact of large language models on generating software specifications. *arXiv preprint arXiv:2306.03324* (2023).

[109] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2024. ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries. (2024).

[110] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving binary code similarity transformer models by semantics-driven instruction deemphasis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1106–1118.

[111] Shouguo Yang, Chaopeng Dong, Yang Xiao, Yiran Cheng, Zhiqiang Shi, Zhi Li, and Limin Sun. 2023. Asteria-Pro: Enhancing Deep Learning-based Binary Code Similarity Detection by Incorporating Domain Knowledge. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–40.

[112] Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. 2020. Pmp: Cost-effective forced execution with probabilistic memory pre-planning. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1121–1138.

[113] Jiahao Yu, Xingwei Lin, Zheng Yu, and Xinyu Xing. 2024. {LLM-Fuzzer}: Scaling Assessment of Large Language Model Jailbreaks. In *33rd USENIX Security Symposium (USENIX Security 24)*. 4657–4674.

[114] Jiawei Zhang, Haopeng Zhang, Congying Xia, and Li Sun. 2020. Graph-bert: Only attention is needed for learning graph representations. *arXiv preprint arXiv:2001.05140* (2020).

[115] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 813–832.

[116] Binbin Zhao, Shouling Ji, Xuhong Zhang, Yuan Tian, Qinying Wang, Yuwen Pu, Chenyang Lyu, and Raheem Beyah. 2023. {UVSCAN}: Detecting {Third-Party} Component Usage Violations in {IoT} Firmware. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3421–3438.

[117] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large Language Model for Vulnerability Detection and Repair: Literature Review and Roadmap.

*arXiv preprint arXiv:2404.02525* (2024).

[118] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).