# Evaluating Large Language Models for Real-World Vulnerability Repair in C/C++ Code

Lan Zhang
Northern Arizona University
AZ, USA

Qingtian Zou
Penn State University
PA, USA
University of Texas Southwestern Medical Center
TX, USA

Anoop Singhal
National Institute of Standards and Technology
MD, USA

Xiaoyan Sun
Worcester Polytechnic Institute
MA, USA

Peng Liu
Penn State University
PA, USA

## ABSTRACT

The advent of Large Language Models (LLMs) has enabled advancement in automated code generation, translation, and summarization. Despite their promise, evaluating the use of LLMs in repairing real-world code vulnerabilities remains underexplored. In this study, we address this gap by evaluating the capability of advanced LLMs, such as ChatGPT-4 and Claude, in fixing memory corruption vulnerabilities in real-world C/C++ code. We meticulously curated 223 real-world C/C++ code snippets encompassing a spectrum of memory corruption vulnerabilities, ranging from straightforward memory leaks to intricate buffer errors. Our findings demonstrate the proficiency of LLMs in rectifying simple memory errors like leaks, where fixes are confined to localized code segments. However, their effectiveness diminishes when addressing complicated vulnerabilities necessitating reasoning about cross-cutting concerns and deeper program semantics. Furthermore, we explore techniques for augmenting LLM performance by incorporating additional knowledge. Our results shed light on both the strengths and limitations of LLMs in automated program repair on genuine code, underscoring the need for advancements in reasoning abilities for handling complex code repair tasks.

## KEYWORDS

Large Language Models, Program Repair, Deep Learning

**ACM Reference Format:**

Lan Zhang, Qingtian Zou, Anoop Singhal, Xiaoyan Sun, and Peng Liu. 2024. Evaluating Large Language Models for Real-World Vulnerability Repair in C/C++ Code. In *Proceedings of* . ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Addressing code vulnerabilities is a significant challenge faced by developers, encompassing flaws or defects within software that, if exploited, can jeopardize system and application security. These vulnerabilities pose severe risks to data integrity, user privacy, and the overall stability of systems. However, manually rectifying these vulnerabilities is a demanding and error-prone process, requiring skilled developers to meticulously analyze the code, identify the root causes of issues, and apply precise patches. This approach often proves time-consuming. To streamline this critical task, recent efforts have turned to employing Large Language Models (LLMs) to automate code vulnerability repairs [2, 8].

Despite the increasing prevalence of LLMs in various code-related tasks, a significant gap remains in the literature regarding their evaluation, particularly in repairing real-world vulnerabilities. This gap primarily arises due to the inherent complexity in assessing LLMs when applied to actual code, which commonly presents intricate structures, interdependencies, and extensive contextual nuances. For instance, vulnerabilities related to memory corruption often involve intricate interactions among functions, variables, or structures, demanding a deep understanding of code logic, posing a stringent challenge for LLMs [22]. Moreover, the absence of standardized benchmarks and comprehensive datasets further complicates the evaluation process, hindering definitive assessment of LLMs' capabilities in code repair. Existing evaluations primarily focus on self-contained issues that can be resolved within a few lines of code, leaving LLMs' applicability in handling complex, real-world scenarios largely unexplored [2, 3, 6, 8, 11, 21]. While these evaluations offer insights into LLMs' strengths, substantial gaps persist in understanding their limitations, especially in addressing intricate and multifaceted code repair tasks [7, 16, 23].

This paper extensively evaluates LLMs for repairing memory corruption vulnerabilities, specifically focusing on widely used C/C++ programming languages. The prominence of C/C++ in industrial applications and the prevalence of memory corruption vulnerabilities make these languages an ideal domain for examination. Our meticulous curation and assessment of 223 real-world C/C++ code snippets aimed to evaluate the repair capabilities of advanced LLMs like ChatGPT-4 and Claude. These snippets cover a wide spectrum of memory corruption vulnerabilities, from simple cases like memory leaks to intricate scenarios involving multifaceted code structures.

Our results revealed that existing LLMs exhibit proficiency in repairing simple memory errors like memory leaks, particularly in localized code segments. However, their efficacy diminishes when handling complex vulnerabilities requiring in-depth analysis of cross-cutting concerns and intricate program semantics. Furthermore, we discuss strategies to enhance LLM performance on intricate vulnerabilities by incorporating additional knowledge, such as context of variables, external functions, data structures, or variables, and logic of the vulnerable code. Our findings offer essential insights into both the capabilities and limitations of LLMs in automated program repair on real code. They serve as a motivator for future research aimed at expanding the reasoning abilities of LLMs for handling complex code repair tasks.

## 2 RELATED WORK

Automatic program repair (APR) refers to the process of automatically identifying and fixing software bugs, errors, or vulnerabilities without human intervention. Various approaches such as search based, symbolic execution based, and deep learning based methods are used to generate these patches [1, 4, 5]. Search based methods utilize evolutionary algorithms, heuristic search, or optimization techniques to explore patch spaces and derive effective solutions [13]. Symbolic execution based methods delve into program paths symbolically to derive constraints guiding the patch generation process [14]. These methods analyze exploit code or test cases, applying repair strategies to generate patches. However, their efficacy might be limited for intricate bugs where the patch space is complex. Deep learning based methods leverage extensive datasets of faulty and corrected programs to train neural networks. These models can then predict patches directly for new bugs [17]. These advancements indicate the potential of LLMs such as GPT-4, capable of understanding and generating code, becoming pivotal in APR tasks, showcasing promising abilities in automated patch generation. Their capacity to analyze code structures and relationships might offer valuable contributions to enhancing the efficiency and accuracy of automated patching systems.

Existing research utilizing LLMs for code repair involves diverse methodologies. Some methods employ LLMs to generate initial code snippets containing vulnerabilities and subsequently repair the generated code using the same models [2, 9, 11]. However, this approach assumes that LLMs accurately understand the vulnerabilities they generate, which may not always hold true. Another approach involves employing LLMs to analyze and repair code problems from platforms like LeetCode [3]. Additionally, the success of LLM based code repair heavily relies on the model's ability to precisely analyze textual descriptions found in reports such as CWE or LeetCode problem statements. Researchers are exploring the combination of LLMs with static analysis techniques to rectify vulnerable code. Static tools are utilized to prompt program errors, which are then examined by LLMs. Upon analyzing these errors, LLMs generate relevant code snippets and offer real-time feedback and suggestions, leveraging the information provided by the queries [19, 21].

The prevailing state-of-the-art in LLM based program repair predominantly concentrates on particular languages such as Java or Python and are often restricted to smaller-scale test programs, which imposes certain constraints [3, 9, 19, 20]. Exsiting work has discussed that existing LLMs and APR models fix very few Java vulnerabilities [20]. However, directly porting the results of a Java or Python program to C++ can present challenges. The languages differ in syntax and core features. For instance, as depicted in Example 1, Java lacks explicit pointers like those in C++. Java manages references to objects, but these references are not directly manipulatable as pointers in C++. Conversely, C++ utilizes explicit pointers for memory management and direct memory access. Moreover, Java employs automatic memory management through garbage collection, demonstrated in Example 2, where 'str' is automatically deallocated when it's no longer referenced or goes out of scope. In contrast, C++ necessitates manual memory management using operations such as 'new' for allocation and 'delete' for deallocation. Failure to explicitly deallocate memory in C++ can lead to memory leaks.

EXAMPLE 1: Syntax difference in Java and C++

**Java:**
```
String str = "Hello";
```

**C++:**
```
std::string* ptrToStr = nullptr;
std::string str = "Hello";
ptrToStr = &str;
```

EXAMPLE 2: Memory management in Java and C++

**Java:**
```
List<Integer> ptr = new ArrayList<>();
```

**C++:**
```
int* ptr = new int(5);
std::cout << "Value: " << *ptr << std::endl;
delete ptr;
```

Another limitation involves the transition from simplified test programs to real-world applications. Real-world software systems are complex, interconnected, and subject to various constraints and dependencies that may not be adequately addressed by current repair methods [2, 9, 11]. While LLMs excel in generating initial code snippets or providing feedback based on provided queries, they often face challenges dealing with intricate vulnerabilities such as memory corruption. Memory corruption spans across the entirety of code context, involving intricate data structures and memory management techniques. This study aims to assess the capabilities of LLMs in automatically generating patches for real-world C/C++ programs, focusing particularly on addressing memory corruption vulnerabilities. These vulnerabilities, encompassing various scenarios like buffer overflows or double-free errors, are critical security risks in software systems. By delving into the LLMs' proficiency in analyzing the memory handling intricacies within C/C++ codebases, the study seeks to understand the extent to which LLMs can effectively identify and rectify these vulnerabilities.

# 3 BACKGROUND

Memory corruption vulnerabilities represent a critical category of security flaws in software, capable of causing severe consequences when exploited by attackers. These vulnerabilities typically encompass manipulations or corruptions within memory areas, potentially resulting in unauthorized access, data leaks, system crashes, or even enabling remote code execution.

In this paper, we discuss the following memory corruption vulnerabilities and their corresponding Common Weakness Enumeration (CWE) identifiers [15]:

- **Buffer Error (CWE-119):** It occurs when data is read from or written to a buffer that is not in the boundary of the buffer. This can lead to memory corruption and potentially code execution.
- **Buffer Copy without Checking Size of Input (CWE-120):** This vulnerability relates to copying data to a buffer without verifying the input's size, which can result in buffer overflows.
- **Integer Overflow (CWE-190):** Integer overflow vulnerabilities happen when an integer value exceeds its maximum representable value, causing data corruption and unexpected program behavior.
- **Double Free (CWE-415):** Double-free vulnerabilities occur when a program tries to free the same memory location more than once. This can lead to memory corruption or crashes.
- **Use After Free (CWE-416):** Use-after-free vulnerabilities happen when a program tries to access memory that has already been deallocated. This can lead to memory corruption or crashes.
- **Memory Leaks (CWE-401):** Memory leaks occur when a program allocates memory but forgets to deallocate it. Although not a direct memory corruption issue, it can lead to inefficient memory usage and eventual exhaustion.
- **NULL Pointer Dereference (CWE-476):** Dereferencing a null pointer can lead to crashes or unpredictable behavior, which is a form of memory corruption.

The vulnerabilities chosen for examination in this paper span a spectrum, including relatively straightforward instances like memory leaks (CWE-401) and more intricate cases such as Buffer errors (CWE-119). Memory leaks are generally considered less complex as they involve the accidental failure to deallocate memory, while Buffer errors, on the other hand, encompass a range of issues like buffer overflows or underflows, often requiring intricate understanding of memory allocation, data structure, and code logic. Understanding and addressing these vulnerabilities are critical in ensuring the development of secure software and applications. These weaknesses, if exploited, can significantly compromise data integrity, breach confidentiality, and undermine the reliability of software systems. Our investigation aims to explore the extent to which LLMs can effectively handle these diverse cases of vulnerabilities.

# 4 CODE PATCH GENERATION

This paper studies the following research questions:

- "Can LLMs effectively generate patches for vulnerable code when supplied with the vulnerability's definition and bug location?" This question serves as an evaluation of LLMs' proficiency in autonomously devising secure code modifications when presented with vulnerability information. It explores their capacity to contribute to automated patch generation.
- "Under which conditions and scenarios do LLMs encounter challenges in producing proficient code patches for vulnerabilities?" This research question seeks to elucidate the contextual constraints and scenarios in which LLMs may falter in generating effective patches for code vulnerabilities.

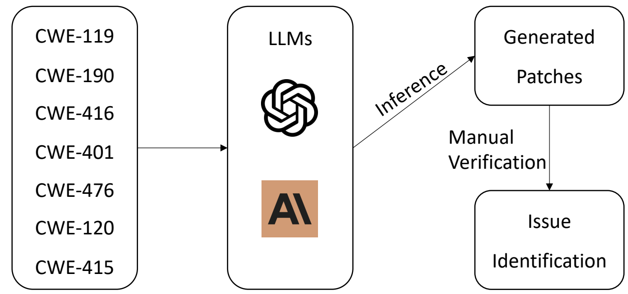


**Figure 1: Overview of our study**

Our assessment of LLMs' performance regarding real-world code vulnerabilities commences by retrieving code snippets from GitHub. These snippets originate from reported instances of memory corruption vulnerabilities in the National Vulnerability Database (NVD) [15]. Our goal in this process is to pinpoint both the functions containing vulnerabilities and the exact locations within the code necessitating patching.

Figure 1 provides an overview of our study. First, we constructed a novel dataset encompassing 223 C++ code snippets, each intricately associated with specific memory corruption vulnerabilities. These vulnerabilities were categorized as follows: roughly 30% were classified under CWE-119, 20% under CWE-190, 17% under CWE-416, and 18% under CWE-476, with the remaining samples falling under CWE-401, CWE-120, and CWE-415 categories. Then, we evaluated the effectiveness of LLMs ChatGPT4 and Claude in addressing these vulnerabilities. For each code snippet, we provided the vulnerability definition and identified the sections requiring modification within the code. Finally, the patches were meticulously scrutinized through manual verification to ascertain their accuracy and identify the issues within incorrect patches. Both the resulting code snippets and their repaired versions have been made publicly available on GitHub [10].

## 4.1 Results of Automated Patching

To prompt LLMs for code patch generation in response to each code snippet, we adopt a structured approach. Initially, we provide a concise introduction outlining the specific vulnerabilities associated with the code. Following this, we instruct the LLMs by specifying that, within the vulnerable code, the "start-bug" marker designates the initiation point of lines requiring modification, while the "end-bug" marker indicates the conclusion of the lines requiring changes. We explicitly emphasize that any code beyond the boundaries of "start-bug" and "end-bug" should remain unaltered. This clear delineation ensures that LLMs understand the precise scope of the required patching within the vulnerable code. Subsequently, we present the vulnerable code and explicitly ask the LLMs to repair the vulnerable code by modifying code between “start-bug" and “end-bug" markers. This systematic process ensures that LLMs receive the scope of the required patching.

After we collect the patched code from LLMs, we conduct a comparative analysis between these generated code snippets and the provided patch codes. This analysis involves calculating the edit distance between the generated code snippets and the correct patch codes, followed by a manual verification process to ascertain the reasons for the generated code snippets’ inaccuracies. This comparative examination enables us to assess the efficacy of LLMs in addressing the identified vulnerabilities and in producing code modifications consistent with the provided patches.

The results, as depicted in Table 1, reveal success rates for both ChatGPT-4 and Claude in effectively patching code snippets. ChatGPT-4 successfully patched approximately 16.5% of the analyzed code snippets, while Claude exhibited a slightly lower patching rate of 11.6%. Within the subset of correctly patched code, a significant proportion, around 43% and 53% respectively, originally has the CWE-401 vulnerability, a prevalent issue where memory allocation occurs but fails to be deallocated.

The ease in patching CWE-401 vulnerabilities is primarily due to the code’s reliance solely on local variables. In the context of C/C++, memory allocation is governed by pointers, denoted by an asterisk (*) preceding the variable name. LLMs demonstrate a notable proficiency in identifying pointers, which notably streamlines the modification process for CWE-401 vulnerabilities. Once these pointer objects are accurately identified, LLMs adeptly proceed to manage the process by releasing the associated memory through the suitable deallocation functions.

Take Example 3 for illustration, the code snippet showcases a function, *rtl8xxxu_submit_int_urb*, responsible for submitting an interrupt *urb* in a device driver. Within this function, an object *urb* is allocated using *usb_alloc_urb*. However, if an error occurs during the *usb_submit_urb* function call, the code jumps to the error label, where cleanup operations are supposed to take place. The ChatGPT response correctly identifies the issue within the code. It recognizes that after the *usb_unanchor_urburb* call, there is a failure to release the memory allocated for the *urb*. To rectify this vulnerability, the proposed fix involves inserting the *usb_free_urburb* function call between the “start-bug" and “end-bug" markers. This action explicitly frees the allocated *urb* memory, addressing the CWE-401 memory leak issue and ensuring proper memory management within the code. Notably, even in scenarios where specific information about the deallocation function is absent, LLMs can reliably deduce the appropriate function, such as *usb_free_urb*, for freeing the allocated memory."

EXAMPLE 3: Handling simple vulnerabilities

*I will provide a code that has bug: CWE-401. The product does not sufficiently track and release allocated memory after it has been used, which slowly consumes remaining memory.*
*In the vulnerable code, the “start-bug" marks the beginning of lines that should be modified, and “end-bug" is the end of lines to be modified. The code that is not within “start-bug" and “end-bug" should remain unchanged.*
*Please only modify the code inside “start-bug" and “end-bug".*

*Code:*

```
static int rtl8xxxu_submit_int_urb(struct ieee80211_hw *hw)
  {
    struct rtl8xxxu_priv *priv = hw->priv;
    struct urb *urb;
    u32 val32;
    int ret;
    urb = usb_alloc_urb(0, GFP_KERNEL);
    if (!urb) return -ENOMEM;
    usb_fill_int_urb(urb, priv->udev, priv->pipe_interrupt,
      priv->int_buf, USB_INTR_CONTENT_LENGTH,
      rtl8xxxu_int_complete, priv, 1);
    usb_anchor_urb(urb, &priv->int_anchor);
    ret = usb_submit_urb(urb, GFP_KERNEL);
    if (ret) {
        usb_unanchor_urb(urb);
        /**<start-bug>**/
        /**<end-bug>**/
        goto error;
    }
    val32 = rtl8xxxu_read32(priv, REG_USB_HIMR);
```

**Table 1: Patch results of ChatGPT-4 and Claude**

| CWE NUM | LLMs | Incorrect | Correct | Total |
|---|---|---|---|---|
| CWE-119 | ChatGPT-4 | 65 | 1 | 66 |
| | Claude | 65 | 1 | |
| CWE-190 | ChatGPT-4 | 42 | 4 | 46 |
| | Claude | 44 | 2 | |
| CWE-416 | ChatGPT-4 | 33 | 6 | 39 |
| | Claude | 37 | 2 | |
| CWE-401 | ChatGPT-4 | 1 | 16 | 17 |
| | Claude | 3 | 14 | |
| CWE-476 | ChatGPT-4 | 32 | 9 | 41 |
| | Claude | 35 | 6 | |
| CWE-120 | ChatGPT-4 | 8 | 0 | 8 |
| | Claude | 8 | 0 | |
| CWE-415 | ChatGPT4 | 6 | 1 | 7 |
| | Claude | 6 | 1 | |

**Table 2: Incorrect Cases of ChatGPT-4 and Claude**

| CWE NUM | LLMs | Incorrect Cases | | | |
|---|---|---|---|---|---|
| | | C1 | C2 | C3 | C4 |
| CWE-119 | ChatGPT-4 | 53 | 0 | 8 | 4 |
| | Claude | 45 | 0 | 16 | 4 |
| CWE-190 | ChatGPT-4 | 35 | 1 | 2 | 4 |
| | Claude | 31 | 0 | 11 | 2 |
| CWE-416 | ChatGPT-4 | 23 | 0 | 7 | 3 |
| | Claude | 22 | 0 | 14 | 1 |
| CWE-401 | ChatGPT-4 | 0 | 0 | 1 | 0 |
| | Claude | 1 | 0 | 2 | 0 |
| CWE-476 | ChatGPT-4 | 29 | 0 | 1 | 2 |
| | Claude | 23 | 0 | 8 | 4 |
| CWE-120 | ChatGPT-4 | 7 | 0 | 0 | 1 |
| | Claude | 5 | 0 | 1 | 2 |
| CWE-415 | ChatGPT4 | 4 | 0 | 2 | 0 |
| | Claude | 3 | 0 | 3 | 0 |

```
    val32 |= USB_HIMR_CPWM;
    rtl8xxxu_write32(priv, REG_USB_HIMR, val32);
error:
    return ret;
}
```

**Reply from ChatGPT (shorten to save space):** In the given code, once the urb is unanchored using 'usb_unanchor_urb(urb)', the memory allocated for urb is not being freed, which is causing memory leak. To fix this bug, we should call 'usb_free_urb(urb)' to free the memory allocated for urb. Updated code:

```
...
        /**<start-bug>**/
        usb_free_urb(urb);
        /**<end-bug>**/
...
```

## 4.2 Issues in Patching the Code

Table 2 illustrates a substantial proportion of incorrectly patched samples, accounting for 83.5% in ChatGPT-4 and 88.4% in Claude. These inaccuracies in patching can be segmented into four distinct categories:

- **C1: Missing Patched Code context**: This category encompasses vulnerabilities that lack related information. Challenges arise due to insufficient context within the code snippet itself, where essential information like function and structure definitions exists beyond the snippet's scope.
- **C2: New Issues in the Patched Code**: Within this category, the patched code resolves the initially identified vulnerability but inadvertently introduces new issues. This highlights the potential for unintended consequences when addressing vulnerabilities. Careful consideration of potential side effects is crucial during the patching process.
- **C3: Error in Identifying the Vulnerable Code**: Vulnerabilities in this category result from inaccuracies in pinpointing the vulnerable section within the code snippet. The primary issue lies in accurately understanding the root cause of the vulnerability. Consequently, the patches provided fail to effectively resolve the underlying issues as they don't accurately target the source of the problem.
- **C4: Error in Understanding the Code**: This category encompasses vulnerabilities originating from misconceptions or misinterpretations regarding the code's logic or functionality. As a result, patches generated based on these misconceptions may prove ineffective in addressing the actual issues within the code.

#### 4.2.1 *C1: Missing Patched Code context.*

This category comprises vulnerabilities that miss important information to patch the code. Patched code relies on variables, structures, or functions absent in the provided snippet, resulting in incomplete or inaccurate patches. To effectively address vulnerabilities, LLMs must access all pertinent information within the code snippet.

In the given example 4, the code snippet exhibits a vulnerability classified under CWE-416. The snippet involves a function *destroy_server_connect* responsible for handling server connections. Within this function, memory allocated for the *ircconn* pointer is freed for *usermode* and *alternate_nick* pointers. The function *g_free_not_null* designed to free memory only if the pointer is not NULL, preventing potential crashes or errors that might occur from trying to free an already deallocated memory or an uninitialized pointer. ChatGPT attempted to propose a solution to address CWE-416 vulnerability by suggesting setting *usermode* and *alternate_nick* as NULL after freeing them. The intention behind this solution was to prevent potential issues related to accessing freed memory. However, ChatGPT's solution deviated from the correct patch required to resolve this vulnerability. The accurate fix should involve freeing the pointers *sasl_username* and *sasl_password* within the *ircconn* structure. The reason for ChatGPT's inability to provide the correct patch lies in its limited understanding of the code's internal structure (*ircconn*). This limitation stems from ChatGPT's reliance on provided information within the code snippet and its lack of broader context or knowledge about the specific structures involved.

EXAMPLE 4: Handing code independence

*I will provide a code that has bug: CWE-416. Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code. ...(shorten to save space)*

```
static void destroy_server_connect(SERVER_CONNECT_REC *conn)
      {
    IRC_SERVER_CONNECT_REC *ircconn;
    ircconn = IRC_SERVER_CONNECT(conn);
    if (ircconn == NULL)
        return;
    g_free_not_null(ircconn->usermode);
    g_free_not_null(ircconn->alternate_nick);
    /**<start-bug>**/
    /**<end-bug>**/
}
```

**Reply from ChatGPT (shorten to save space):** The pointer *ircconn* is freed but it is not set to NULL, and can therefore risk being used after it has been freed. In order to avoid this issue, we should set the pointer to NULL after freeing it.

```
...
    /**<start-bug>**/
    ircconn->usermode = NULL;
    ircconn->alternate_nick = NULL;
    /**<end-bug>**/
...
```

**Correct patch:**

```
...
    /**<start-bug>**/
    g_free_not_null(ircconn->sasl_username) ;
    g_free_not_null(ircconn->sasl_password) ;
    /**<end-bug>**/
...
```

#### 4.2.2 *C2: New Issues in the Patched Code.*

This category involves scenarios where the patched code initially succeeds in resolving the identified vulnerability. However, the application of the patch inadvertently introduces a new issue into the codebase. In essence, while one problem is successfully fixed, another issue arises, underscoring the importance of exercising caution and thorough consideration when applying code patches to anticipate potential side effects.

In Example 5, the function *php_html_entities* processes strings *str* and performs HTML entity encoding using another function *php_escape_html_entities_ex*, converting special characters to their corresponding HTML entities. However, there is a vulnerability in this code related to CWE-190, indicating a potential integer overflow or wraparound when dealing with *new_len*. To address this vulnerability, ChatGPT attempts to check if the value of *new_len* exceeds the maximum allowed integer value ( $INT\_MAX$). If this condition is true, it triggers an error message using the *php_error_docref* function, which is retrieved from ChatGPT training dataset, and returns *FALSE* to signal an error state. This check aims to prevent any issues resulting from the integer overflow by flagging it as an error and avoiding potential vulnerabilities that might arise from data loss or unexpected behavior due to integer wraparound. However, the code lacks a crucial step: deallocating the *replaced* pointer before returning, which can potentially cause memory leaks. This additional step ensures that the fix doesn't inadvertently introduce new issues, highlighting the necessity for a meticulous approach to code patching to prevent unintended consequences.

EXAMPLE 5: Introducing new issues

*I will provide a code that has bug: CWE-190. The product performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control. ...(shorten to save space)*

```
static void php_html_entities(INTERNAL_FUNCTION_PARAMETERS,
    int all) {
    char *str, *hint_charset = NULL;
    int str_len, hint_charset_len = 0;
    size_t new_len;
    long flags = ENT_COMPAT;
    char *replaced;
    zend_bool double_encode = 1;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s|
      ls!b", &str, &str_len, &flags, &hint_charset, &
      hint_charset_len, &double_encode) == FAILURE) {
        return;
    }
    replaced = php_escape_html_entities_ex(str, str_len, &
      new_len, all, (int)flags, hint_charset, double_encode
      TSRMLS_CC);
    /**<start-bug>**/
    /**<end-bug>**/
    RETVAL_STRINGL(replaced, (int)new_len, 0);
}
```

**Reply from ChatGPT (shorten to save space):** Because size_t is an unsigned integer type and its width is not less than that of int, when a value of type size_t, which exceeds the maximum value of int, is assigned/casted to a variable of type int, only the lower part is taken and this results in apparent data loss.

```
...
    /* Check for integer overflow */
    if (new_len > INT_MAX) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING, "Integer
          overflow detected in new_len");
        RETURN_FALSE;
        return;
    }
...
```

**Correct patch:**

```
...
    /**<start-bug>**/
    if(new_len > INT_MAX ) {
        efree(replaced);
        RETURN_FALSE ;
    }
    /**<end-bug>**/
...
```

#### 4.2.3 *C3: Error in Identifying the Vulnerable Code.*

Vulnerabilities categorized under C3 arise due to inaccuracies in pinpointing the specific section of the code that is vulnerable. The primary challenge lies in accurately identifying the vulnerable segment within the given code snippet. As a consequence, the generated patches do not effectively address the underlying issues, as they fail to precisely target the actual source of the problem.

For instance, consider Example 6, the *gdImageWBMPPtr* function aims to extract data from a GD image. However, it contains vulnerabilities related to CWE-415, specifically a potential double-free error that might occur due to inappropriate handling of the *gdIOCtx* structure (*out*). If the *gdImageWBMPCtx* function fails to successfully obtain the *gdIOCtx* structure, resulting in the *out* variable being NULL. Subsequently, attempting to extract data using this out pointer when it is NULL would cause issues. ChatGPT faces challenges due to two primary reasons. Firstly, ChatGPT misinterprets the code markers (e.g., "start-bug" and "end-bug") as standard text elements instead of recognizing them as indicators for segments requiring modification, especially when there are multiple makers. Consequently, this misunderstanding prevents the model from grasping the importance of these markers in pinpointing the vulnerable sections for patching, particularly those related to potential NULL-pointer issues in handling the *out* pointer within the *gdImageWBMPPtr* function. Second, similar to C1 and C4 categories, the model struggles when analyzing the intricate context of the code, resulting in difficulties in precisely identifying the segments that contain vulnerabilities. This lack of accurate identification leads to generated code modifications that may not effectively align with the intended changes, reducing the overall accuracy of the patches provided.

EXAMPLE 6: Error in identifying modification markers

*I will provide a code that has bug: CWE-415. The product calls free twice on the same memory address, potentially leading to modification of unexpected memory locations. ...(shorten to save space)*

```
void *gdImageWBMPPtr(gdImagePtr im, int *size, int fg) {
    void *rv;
    gdIOCtx *out = gdNewDynamicCtx(2048, NULL);
    /**<start-bug>**/
    gdImageWBMPCtx(im, fg, out);
    /**<end-bug>**/
    rv = gdDPExtractData(out, size);
    /**<start-bug>**/
    /**<end-bug>**/
    out->gd_free(out);
    return rv;
}
```

**Reply from ChatGPT (shorten to save space):** The problem is that the code doesn't store a copy of the pointer *out* before it's freed. The fact 'out->gd_free(out);' might led to a double free error in later executions. To fix this bug, we can use a temporary pointer to free *out* so the original pointer *out* doesn't get modified. This way, we prevent a potential double freeing.

```
void *gdImageWBMPPtr(gdImagePtr im , int * size , int fg) {
    void *rv ;
    gdIOCtx *out = gdNewDynamicCtx(2048, NULL);
    gdIOCtx *tmpOut = out;
    /**<start-bug>**/
    gdImageWBMPCtx(im, fg, tmpOut);
    /**<end-bug>**/
    rv = gdDPExtractData(tmpOut, size);
    tmpOut->gd_free(tmpOut);
    return rv ;
}
```

**Correct patch:**

```
void *gdImageWBMPPtr(gdImagePtr im, int *size, int fg) {
    void *rv;
    gdIOCtx *out = gdNewDynamicCtx(2048, NULL);
    /**<start-bug>**/
    if (!_gdImageWBMPCtx(im, fg, out)) {
    /**<end-bug>**/
        rv = gdDPExtractData(out, size);
        /**<start-bug>**/
    } else {
        rv = NULL;
    }
    /**<end-bug>**/
    out->gd_free(out);
    return rv;
}
```

#### 4.2.4 *C4: Error in Understanding the Code.*

This category encompasses vulnerabilities resulting from a fundamental misinterpretation of the code. Such vulnerabilities arise from misconceived perceptions or misunderstandings regarding the code's inherent logic and functionality. As a consequence, patches generated under these misconceptions may fall short in addressing the actual issues within the code. Patching these vulnerabilities demands a deeper analysis of the code's internal operations.

The original code in Example 7, showcases a vulnerability related to CWE-119, where the function tokenadd operates on a memory buffer ($p->tokenbuf$). This function aims to append a character ($c$) to the buffer and handles its expansion when required. The code begins with an assertion to ensure that the current position within the buffer ($p->tokenpos$) equals the allocated length ($p->tokenlen$). If the condition holds true, the code doubles the allocated size of the buffer and reallocates memory using the *jv_mem_realloc* function. While ChatGPT correctly identifies the need for memory allocation when $tokenpos$ reaches $tokenlen$, it fails to precisely understand the buffer's maximum index range. The issue arises from the assumption that the maximum index for $tokenpos$ should be $tokenlen$, which is incorrect. Instead, the buffer $tokenbuf$ should add up if $tokenpos$ is greater than or equal to $tokenlen - 1$.

Example 7: Error in understanding the code snippet

*I will provide a code that has bug: CWE-119. The product performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer. ...(shorten to save space)*

```
static void tokenadd(struct jv_parser *p, char c) {
    assert(p->tokenpos <= p->tokenlen);
    /**<start-bug>**/
    if (p->tokenpos == p->tokenlen) {
        /**<end-bug>**/
        p->tokenlen = p->tokenlen * 2 + 256;
        p->tokenbuf = jv_mem_realloc(p->tokenbuf, p->tokenlen);
    }
    assert(p->tokenpos < p->tokenlen);
    p->tokenbuf[p->tokenpos++] = c;
}
```

**Reply from ChatGPT (shorten to save space):** The issue here is that there's a potential to write outside of the allocated memory for $tokenbuf$ in the last line of code. If $p->tokenpos$ is equal to $p->tokenlen$ then it allocates new memory but it doesn't check then if *tokenpos* was updated to a value larger than *tokenlen* in multithreaded scenarios, or if *jv_mem_realloc* didn't allocate memory correctly (failure case).

```
...
    /**<start-bug>**/
    if(p->tokenpos >= p->tokenlen){
        p->tokenlen = p->tokenlen*2+256 ;
        char* newbuf = jv_mem_realloc(p -> tokenbuf, p -> tokenlen);
        if(newbuf == NULL) {
            fprintf(stderr, "Memory allocation failed\n");
            return;
        }
        p -> tokenbuf = newbuf;
    }
    /**<end-bug>**/
    assert ( p -> tokenpos < p -> tokenlen ) ;
...
```

**Correct patch:**

```
...
    /**<start-bug>**/
    if(p->tokenpos >= (p ->tokenlen - 1)){
    /**<end-bug>**/
...
```

## 5 TECHNIQUES TO IMPROVE VULNERABILITY REPAIR

The analysis of code vulnerabilities by LLMs can be limited by several factors. These constraints often arise from the intricate nature of code context, the precision of the prompt description, and the potential vagueness within textual depictions of vulnerabilities. LLMs heavily depend on the details in the prompt to create code. If the prompt is not comprehensive, LLMs might struggle to grasp the precise nature of the vulnerability [18]. This limitation could result in generating patches that address the symptoms rather than the underlying root cause, potentially leading to incomplete or ineffective solutions.

Improving the efficacy of LLMs in addressing code vulnerabilities relies on supplementing them with additional contextual knowledge. The study conducted by Zhang et al. delves into the significance of code context by examining the consequences of removing surrounding correct lines in a vulnerable code snippet [23]. However, solely providing the lines vulnerable to a security flaw and asking

LLMs to repair the code is not sufficient. For instance, consider the code snippet in Example 7, where presenting only the vulnerable line $if\, p->tokenpos == p->tokenlen$ without context may lead LLMs to struggle in analyzing the cause of the vulnerability and thus impede their ability to resolve the issue effectively.

In our paper, we focus on augmenting LLMs with additional knowledge that encompasses various elements such as the context of variables, understanding of external functions, data structures or variables, and the underlying logic of the vulnerable code. This augmented information aims to provide LLMs with a deeper understanding of the code's intricacies, enabling them to discern the root cause of vulnerabilities and devise more accurate and contextually appropriate patches. The following sections discuss different kind of additional knowledge.

## 5.1 Context of Variables

Understanding the contextual significance of variables proves pivotal in LLMs' adeptness at patching code vulnerabilities. This information involves grasping the variable's scope, its potential values, and its usage patterns within the code snippet.

For example, in the given case study 1, aimed at addressing the vulnerabilities in the Example 5, we explicitly explain the importance of the *replaced* variable in managing memory. By elucidating that *replaced* can trigger memory corruption, LLMs can generate patches that handle memory overflow issues more effectively. This contextual information significantly enhances the LLM's ability to generate more accurate and contextually aware code patches, thereby addressing vulnerabilities more effectively while maintaining the code's intended functionality.

CASE STUDY 1: Adding explanation of variables

**Additional knowledge:** *The allocated memory replaced should be freed if an error is detected.*

**Reply from ChatGPT (shorten to save space):**

```
...
    /**<start-bug>**/
    if(new_len > INT_MAX ) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING, "Integer
      overflow detected in new_len");
        efree(replaced);
        RETURN_FALSE ;
    }
    /**<end-bug>**/
...
```

## 5.2 External Functions, Data Structures or Variables

Code vulnerabilities can also manifest due to improper handling of external functions, data structures or variables. Equipping LLMs with knowledge about these external information, including return values, data types, and access patterns, allows them to identify potential memory corruption vulnerabilities and generate repair patches that adhere to safe memory management practices.

As shown in the case study 2 for the example 4, adding detailed information about the *username* and *password* variables within the *SERVER_CONNECT_REC* structure proves invaluable. By emphasizing that these variables should be freed to ensure complete memory management, we equip the LLM with a broader context of the vulnerability.

CASE STUDY 2: Adding explanation of data structures

**Additional knowledge:** *username and password in SERVER_CONNECT_REC structure should also be freed.*

**Reply from ChatGPT (shorten to save space):**

```
...
    /**<start-bug>**/
    g_free_not_null(ircconn->username);
    g_free_not_null(ircconn->password);
    ircconn->usermode = NULL;
    ircconn->alternate_nick = NULL;
    ircconn->username = NULL;
    ircconn->password = NULL;
    /**<end-bug>**/
...
```

## 5.3 Logic of the Vulnerable Code

Understanding the logic and functionality of the vulnerable code is essential for generating meaningful patches. LLMs should be aware of the expected behavior of the code, the purpose of different code segments, and the intended flow of the function.

For instance, in case study 3, which corresponds to the example 6, explanations of external functions such as *gdImageWBMPCtx*, *_gdImageWBMPCtx*, and *gdDPExtractData* is essential. By highlighting that the double-free issue originates from not checking the success of writing the WBMP file into the output stream, and suggesting the utilization of *_gdImageWBMPCtx* as an indicator of successful file writing, LLMs can craft more informed patches. Consequently, the LLMs can leverage this information to generate repair patches that check for successful writing of the WBMP file before proceeding further, ensuring safer and more robust code practices.

Similarly, in case study 4 associated with example 7, providing comprehensive information about the vulnerable code's logic and the relationship between the variables $p->tokenpos$ and $p->tokenlen$ is crucial to aid LLMs in generating effective patches. This additional knowledge should involve explaining the fundamental logic behind the vulnerability, emphasizing the potential disparity between $p->tokenpos$ and $p->tokenlen$, and clarifying the proper boundary conditions. Specifically, elucidating that $p->tokenbuf$ is capable of holding $p->tokenlen$ elements, making the maximum index $p->tokenpos$ should be $p->tokenlen-1$.

CASE STUDY 3: Adding explanation of external functions and code logic

**Additional knowledge:** *The double free is caused by not checking if image has successfully been written into out. If image has been successfully written into out, _gdImageWBMPCtx will return 0. Therefore, we could use _gdImageWBMPCtx instead of gdImageWBMPCtx to check if writing the WBMP file is successful. If it is successful, in gdDPExtractData, it* *cleans up the data block and returns to rv. Otherwise, rv should be set to NULL. Here is the definition of gdImageWBMPCtx, _gdImageWBMPCtx, and gdDPExtractData. (shorten to save space)*

```
/* Function: gdImageWBMPCtx
Write the image as a wbmp file
Parameters:
image - gd image structure
fg    - the index of the foreground color. any other value
        will be considered as background and will not be
        written
out   - the stream where to write
*/
BGD_DECLARE(void) gdImageWBMPCtx(gdImagePtr image, int fg,
        gdIOCtx *out)
{
  _gdImageWBMPCtx(image, fg, out);
}
/* Function: _gdImageWBMPCtx
Returns 0 on success, 1 on failure
*/
static int _gdImageWBMPCtx(gdImagePtr image, int fg, gdIOCtx
         *out)
{
  int x, y, pos;
  Wbmp *wbmp;
  /* create the WBMP */
  ...
  /* fill up the WBMP structure */
  ...
  /* write the WBMP to a gd file descriptor */
  if(writewbmp(wbmp, &gd_putout, out)) {
    freewbmp(wbmp);
    gd_error("Could not save WBMP\n");
    return 1;
  }
  freewbmp(wbmp);
  return 0;
}
BGD_DECLARE(void *) gdDPExtractData(gdIOCtxPtr ctx, int *
        size)
{
  dynamicPtr *dp;
  dpIOCtx *dctx;
  void *data;
  dctx = (dpIOCtx *)ctx;
  dp = dctx->dp;
  /* clean up the data block and return it */
  if(dp->dataGood) {
    trimDynamic(dp);
    *size = dp->logicalSize;
    data = dp->data;
  } else {
    *size = 0;
    data = NULL;
    if((dp->data != NULL) && (dp->freeOK)) {
      gdFree(dp->data);
    }
  }
  dp->data = NULL;
  dp->realSize = 0;
  dp->logicalSize = 0;
  return data;
}
```

**Reply from ChatGPT (shorten to save space):**

```
...
    if (_gdImageWBMPCtx(im, fg, out) == 0) {
        rv = gdDPExtractData ( out , size ) ;
    } else {
        rv = NULL;  // Indicates failure, rv should be set
        as NULL
    }
...
```

CASE STUDY 4: Adding explanation of code logic

**Additional knowledge:** *Off-by-one error in the tokenadd function allows remote attackers to cause a denial of service (crash), which triggers a heap-based buffer overflow. If $p->tokenpos$ is more than or equal to the end of buffer, resize the buffer. The end of buffer should be $p->tokenlen-1$ not $p->tokenlen$.*

**Reply from ChatGPT (shorten to save space):**

```
...
    /**<start-bug>**/
    if(p -> tokenpos >= p -> tokenlen - 1){
        p -> tokenlen = p -> tokenlen * 2 + 256 ;
        char* newbuf = jv_mem_realloc(p->tokenbuf , p->
      tokenlen);
        if(newbuf == NULL) {
            fprintf(stderr, "Memory allocation failed\n");
            return;
        }
        p->tokenbuf = newbuf;
    }
    /**<end-bug>**/
...
```

# 6 DISCUSSION

While this study presents valuable insights into leveraging LLM based approaches for automatic vulnerability repair, the focus of future research involves refining the approach by generalizing prompt templates to enhance vulnerability repair. When encountering similar issues across different codes, consistent prompts can effectively enhance the quality of the generated patched code even when different variables are involved. Clustering prompts to create templates for vulnerability repairs offers a structured way to address common vulnerabilities. In a case study like 1, where ChatGPT faced challenges related to freeing objects, this acquired knowledge could help create specialized templates, such as "The allocated memory variable name should be freed if an error is detected." However, certain cases, such as case study 3, cannot be generalized into templates. A deeper understanding of the vulnerable code's logic is required for substantial improvement in the patched code.

Another promising avenue for future exploration involves fine-tuning LLMs using datasets specifically curated for code repair tasks. Training LLMs on meticulously crafted datasets tailored for these tasks has the potential to significantly enhance their analysis of programming structures, error patterns, and appropriate remedial actions. However, it's critical to note that fine-tuning the model solely with vulnerable code snippets lacking contextual information is impractical. Consider Case Study 2—without the context of the external data structure in the training dataset, patching the code becomes an insurmountable challenge. Hence, comprehensive datasets encompassing complete contexts and pertinent libraries are imperative for effective model refinement. These datasets play a crucial role in empowering LLMs with the essential information required to identify intricate vulnerabilities and generate accurate patches. Since LLMs are black-box models, it is challenging to interpret which features are learned by the models and why different LLMs produce different outputs for the same input. If a fine-tuned model is available, interpreting its output using techniques like attention visualization or concept activation vectors could aid in understanding the results of LLMs and identifying potential biases or limitations.

Finally, in our experiments, we provided the incorrect lines in the prompts and tasked LLMs to repair these lines following the methodology of existing work [2, 9, 11, 12]. While this approach is useful for evaluating the LLM's ability to generate patches for known vulnerabilities, it may not fully capture the LLM's potential for vulnerability detection and localization. It would be worthwhile to test whether LLMs have the ability to detect vulnerabilities and locate the bug lines independently, without being provided with the specific vulnerable lines. This could involve providing the LLM with the complete source code and tasking it to analyze the code, identify and highlight potentially vulnerable lines or functions, and optionally suggest repairs. Such an approach would more closely resemble real-world scenarios where the location of vulnerabilities is not known a priori and would require the LLM to leverage its understanding of programming constructs, security best practices, and potential attack vectors.

# 7 CONCLUSION

In conclusion, this study presents a extensive evaluation of LLMs for automated code repair in the context of memory corruption vulnerabilities. Our analysis, encompassing a diverse range of real-world code snippets associated with various Common Weakness Enumeration (CWE) categories, sheds light on both the capabilities and limitations of LLMs in addressing code vulnerabilities.

The results shows while LLMs have showcased promising potential in handling simpler vulnerabilities like memory leaks, their efficacy diminishes when addressing more complex issues involving external dependencies, nuanced contextual information, and intricate code structures. The categorization of challenges faced by LLMs, such as issues with missing information, introduction of new issues, misidentification of vulnerable code, and misunderstanding of code logic, provides valuable insights into the areas where further advancements are needed. Moreover, the exploration of additional knowledge incorporation, including contextual variables, external functions, data structures, and logic explanation, demonstrates the potential for enhancing LLM performance in code repair tasks.

This work contributes by highlighting the need for more sophisticated methodologies and additional contextual information to enable LLMs to address multifaceted code vulnerabilities effectively. Future research directions could focus on refining LLM training methods, devising strategies for better contextual understanding, and developing standardized benchmarks for extensive evaluations in the domain of automated code repair.

# 8 DISCLAIMER

Commercial products are identified in order to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the identified products are necessarily the best available for the purpose.

# REFERENCES

[1] Bill Baloglu. 2016. How to find and fix software vulnerabilities with coverity static analysis. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 153–153.

[2] Jialun Cao, Meiziniu Li, Ming Wen, and Shing-chi Cheung. 2023. *A study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair*. Vol. 1. Association for Computing Machinery. arXiv:2304.08191 http://arxiv.org/abs/2304.08191

[3] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Automated Repair of Programs from Large Language Models. (2022). arXiv:2205.10583 http://arxiv.org/abs/2205.10583

[4] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. *arXiv preprint arXiv:2303.18184* (2023).

[5] Raoul Praful Jetley, Paul L Jones, and Paul Anderson. 2008. Static analysis of medical device software using CodeSonar. In *Proceedings of the 2008 workshop on Static analysis*. 22–29.

[6] Nan Jiang, Thibaud Lutellier, Jordan Davis, Lin Tan, and Petr Babkin. 2021. How E ective Are Neural Networks for Fixing Security Vulnerabilities. (2021).

[7] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? (2023), 1–46. arXiv:2310.06770 http://arxiv.org/abs/2310.06770

[8] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Ivan Radiček, and Gust Verbruggen. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. *Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI 2023* 37 (2023), 5131–5140. https://doi.org/10.1609/aaai.v37i4.25642 arXiv:2208.11640

[9] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2022. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. (2022). arXiv:2209.11515 http://arxiv.org/abs/2209.11515

[10] Lan. 2023. *(ChatGPT for Source Code)*. Technical Report. https://github.com/Moirai7/ChatGPTforSourceCode.git

[11] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. [n. d.]. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. *Icse* ([n. d.]), 1–13. https://github.com/microsoft/codamosa.

[12] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: a novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 511–523. https://doi.org/10.1145/3510003.3510177

[13] Ben Mehne, Hiroaki Yoshida, Mukul R Prasad, Koushik Sen, Divya Gopinath, and Sarfraz Khurshid. 2018. Accelerating search-based program repair. In *2018 IEEE 11th international conference on software testing, verification and validation (ICST)*. IEEE, 227–238.

[14] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[15] NIST. 2023. *(NATIONAL VULNERABILITY DATABASE)*. Technical Report. https://nvd.nist.gov/

[16] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. [n. d.]. Examining Zero-Shot Vulnerability Repair with Large Language Models. ([n. d.]). arXiv:2112.02125v3 https://popcon.debian.org/.

[17] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[18] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. 23–30. https://doi.org/10.1109/APR59189.2023.00012

[19] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2023. Frustrated with Code Quality Issues? LLMs can Help! *arXiv preprint arXiv:2309.12938* (2023).

[20] 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. *arXiv preprint arXiv:2305.18607* (2023).

[21] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[22] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. 2005. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the 12th ACM conference on Computer and communications security*. 223–234.

[23] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. 2023. Pre-Trained Model-Based Automated Software Vulnerability Repair: How Far are We? *IEEE Transactions on Dependable and Secure Computing* 14, 8 (2023), 1–18. arXiv:2308.12533