

# Enhance Language Model-based Repair for Memory-related Vulnerabilities via Knowledge- and Semantic-guided Analysis

Hao Shen<sup>†</sup>, Ming Hu<sup>†</sup>, Yanxin Yang<sup>†</sup>, Xiaofei Xie<sup>‡</sup>, Mingsong Chen<sup>†</sup>

<sup>†</sup>MoE Engineering Research Center of SW/HW Co-Design Technology and Application, East China Normal University, China

<sup>‡</sup>School of Computing and Information Systems, Singapore Management University, Singapore

**Abstract**—Memory-related vulnerabilities often result in system crashes and performance drops, imposing significant risks for embedded systems. Despite the potential of Language Models (LMs) in program repair, existing LM-based approaches struggle with these vulnerabilities due to two primary limitations: i) LMs do not possess adequate domain knowledge concerning program analysis and the characteristics of memory-related vulnerabilities, and ii) LMs face constraints in managing contexts as the size of programs increases. To address this issue, we introduce MVRepair, a novel lightweight Language Model ( $\ell$ LM)-driven framework built upon a domain-specific knowledge library that is developed through the examination of 7,935 real-world memory-related vulnerabilities. By using our proposed knowledge-based analysis strategy and semantic-guided segmentation mechanism, MVRepair can substantially enhance the LM’s ability to repair programs with memory-related vulnerabilities. Comprehensive experimental results on 8,118 real-world memory-related vulnerabilities demonstrate that, compared with state-of-the-art LM-based approaches, MVRepair yields improvements of a minimum of 23.8% in EM, 31.9% in BLEU-4, and 16.7% in CodeBLEU.

## I. INTRODUCTION

The growing complexity of modern embedded systems has resulted in increased vulnerabilities, particularly related to memory [1], [2]. Accounting for approximately 40% of the entries in the CVE database [3], these vulnerabilities pose serious threats to system reliability and security [4]–[7], which can easily lead to performance degradation or even system crashes. Due to intricate code dependencies, addressing these vulnerabilities is challenging, often requiring in-depth manual examination and expert knowledge. Thus, investigating automated and precise solutions is increasingly vital for enhancing the efficiency of embedded software development [8]–[10].

The advent of Deep Learning (DL) models in comprehending natural language, particularly through pre-trained models, has led to the development of numerous DL-driven techniques [11]–[13] aimed at program repair. Nonetheless, these methods struggle when dealing with vulnerabilities related to memory due to the following two primary limitations: ❶ Language Models (LMs) lack the capability to perform program analysis and possess insufficient knowledge of memory-related vulnerabilities; and ❷ since segmented code resulting from token limitations [12] often compromises the semantic integrity of vulnerabilities related to memory, the arbitrary segmentation inevitably disrupts the structure and semantics of the target program code, thus misleading LMs in comprehending code. Clearly, *enhancing the capabilities of language models with domain-specific insights about memory-related vulnerabilities*

*and improving their ability to understand the intricate contexts of such vulnerable code are emerging as pivotal challenges in the automated remediation of these vulnerabilities.*

To address the above challenge, this paper proposes MVRepair, a novel lightweight Language Model ( $\ell$ LM)-based vulnerability repair framework. Instead of segmenting vulnerable program code at the token level, MVRepair strives to enhance the ability of LMs to wisely understand the real context of vulnerable code at the grammar level. Firstly, we analyze 7,935 real-world cases to build a comprehensive knowledge library that reflects various characteristics of memory-related vulnerabilities. Built on top of this constructed knowledge library, MVRepair enables effective repair of memory-related vulnerabilities based on ❶ a knowledge-based static analysis approach that utilizes domain-specific knowledge about memory-related vulnerabilities to enhance the LM-based repair process and ❷ a semantic-guided segmentation mechanism that boosts the ability of LMs to reason through complex contexts. Specifically, the knowledge-based static analysis strategy extracts potential characteristics of memory-related vulnerabilities from a target program using the predefined knowledge library, which are then used to construct tailored prompts. Moreover, this approach entails the development of a Semantic Sampling Tree (SST) for the target program. Based on this SST, we carry out Adaptive code SEGmentation (ASEG) while adhering to the tag length imposed by the underlying LM. In this way, MVRepair maintains the semantic integrity of the analyzed target program, thereby improving the LM’s ability to understand the vulnerable program code and perform repairs with higher quality.

This paper mainly makes the following three contributions:

- We establish a knowledge library for various memory-related vulnerabilities based on 7,935 real-world cases.
- We propose a novel vulnerability repair framework, MVRepair, which adopts our proposed knowledge-based static analysis strategy and semantic-guided segmentation mechanism to enhance the repair capability of LMs.
- We perform a comprehensive evaluation on a real-world dataset to demonstrate the effectiveness of MVRepair.

## II. A MOTIVATION EXAMPLE

Figure 1 illustrates how security engineers address an out-of-bounds write vulnerability (CVE-2021-46822) in *libjpeg-turbo*, highlighting the need for both knowledge-based static analysis and semantic-guided segmentation. In practice, engineers must first locate the vulnerable code region, which often

involves reasoning over literals, identifiers, and expressions, as summarized in Table I. For example, as shown on line 8 in Figure 1, the variable `maxval` is used without being validated against the `IS_EXT_RGB` condition, which is an issue that requires understanding the semantics of identifiers and their relationships within expressions. This manual process relies heavily on domain expertise to correctly trace program dependencies before applying the appropriate repair.

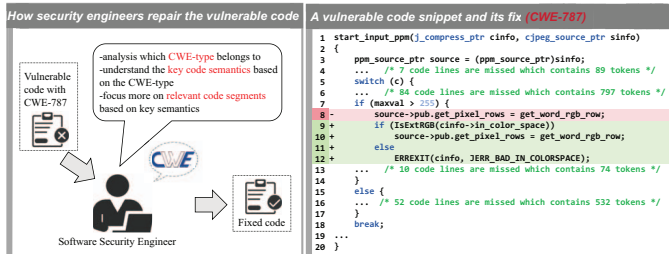


Fig. 1. A motivating example: the left part shows how a software security engineer fixes vulnerable code, while the right part presents a snippet of the repaired vulnerable code from `libjpeg-turbo`.

Automated approaches face additional challenges. Vulnerable functions are often long and context-dependent, with critical information scattered across hundreds of tokens. In our example, the vulnerable function contains more than 800 tokens before reaching the faulty statement. Existing approaches, such as VulMaster, segment code using a fixed token length (e.g., 512 tokens), which risks cutting across variable declarations or logical blocks. Such segmentation breaks semantic continuity, depriving models of essential context and ultimately degrading repair quality. This motivates our design of MVRepair, by combining a knowledge-based static analysis to extract vulnerability-relevant code elements with a semantic-guided segmentation mechanism, we preserve the structural and semantic integrity of the function, allowing the model to reason over the complete context needed for accurate repair.

### III. METHODOLOGY

Figure 2 illustrates the framework and workflow of our proposed MVRepair. As shown in the middle part of the figure, MVRepair consists of two parts, i.e., the knowledge-based static analysis strategy and the semantic-guided segmentation mechanism. Specifically, based on the pre-established knowledge library for various memory-related vulnerabilities, the former part determines an SST for each program using our static analysis engine. Then, by using our developed segmentation engine together with pre-defined prompt templates, the latter part properly segments vulnerable programs for prompt generation, without compromising their structural and semantic information for code repair. Note that, as shown in the left part, the fine-tuning of  $\ell$ LMs in MVRepair is also based on the results produced by these two engines. The following subsections will detail their implementation details.

#### A. Knowledge-based Static Analysis Strategy

To systematically capture the characteristics of memory-related vulnerabilities, we analyzed 7,935 real-world cases from the MegaVul dataset, which aggregates vulnerabilities from 17,380 open-source repositories and documents 169 unique

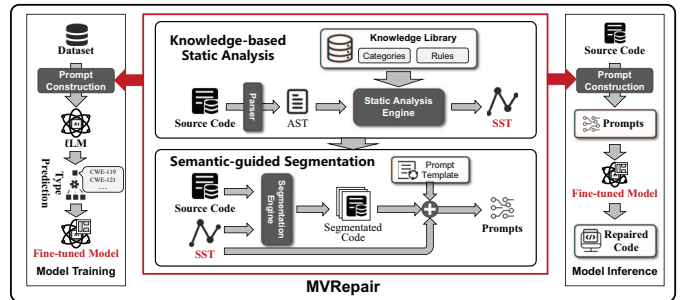


Fig. 2. The framework and workflow of MVRepair.

CWE types reported between January 2006 and October 2023. Our analysis focuses on code regions within the vulnerable functions that could introduce security issues. We observed that such regions consistently involve literals or identifiers, which appear in key syntactic constructs such as declarations and expressions. Declarations define program elements (e.g., variables, functions, classes), while expressions combine literals, identifiers, and operators to compute values. Vulnerable statements often reference earlier declarations or participate in later expressions, making them crucial for understanding vulnerability context. Based on these observations, we distill four key AST node categories. These four categories of nodes are literal values, identifiers, declarations, and expressions. We compile these four categories into a knowledge library, which serves as the foundation for our static analysis strategy. Table I outlines these node categories and their forms.

TABLE I  
KEY NODE INFORMATION FOR MEMORY-RELATED VULNERABILITIES.

Category	Specific Type	Symbolic Example
Literal	StringLiteral	"iceman"
	NumberLiteral	30
	BooleanLiteral	TRUE
	RegExpLiteral	/\d/
Identifier	Identifier	age
Declaration	VariableDeclaration	const listlen = 1
	FunctionDeclaration	function getInfo(info)
Expression	ArrayExpression	[1,2,3]
	AssignmentExpression	age = 1
	BinaryExpression	1 + 2
	ObjectExpression	var obj

To extract characteristics of memory vulnerabilities at the grammar level, MVRepair adopts the static analysis strategy based on the pre-established knowledge library. This strategy aims to utilize domain-specific knowledge about memory-related vulnerabilities to better understand the complex structural contexts within target programs, thus facilitating code segmentation and enhancing repair performance.

In MVRepair, we utilize the static analysis tool Tree-Sitter [14] to parse the source code and construct Abstract Syntax Trees (ASTs). Since vulnerabilities often occur in only a few lines of code, providing the entire program to the model can negatively impact its performance. Therefore, we employ program slicing [15], which involves performing both backward and forward slicing from a specific point of interest in the program. This technique helps reduce the noise caused by irrelevant statements. To ensure that the sliced code includes memory-related vulnerabilities, we use a set of node type rules summarized in Table I to create an SST. This process extracts



### C. Vulnerable Code Repair

To capture hidden vulnerability patterns, we incorporate prior knowledge of vulnerability types during training. By conditioning the generation process on the predicted vulnerability type, MVRepair can better recognize recurring vulnerability patterns and select repair strategies that are most effective for that type.

Similar to existing code-related tasks, MVRepair employs a multi-task learning framework to enhance its effectiveness. We incorporate vulnerability type prediction as an auxiliary task, which helps the model learn more complex features. The model is trained on two consecutive tasks: i) vulnerability type prediction, which identifies the specific type of vulnerability within the dataset, and ii) vulnerability repair, which generates the corrected code. This allows the model to utilize its prior knowledge of a vulnerability type to inform its repair strategy.

To efficiently adapt the repair generation module under resource constraints, MVRepair fine-tunes the underlying  $\ell$ LM using Low-Rank Adaptation (LoRA). Given the fused representation  $[h_c, h_{ct}]$  from the cross-attention layer, LoRA learns a low-rank update  $\Delta W = BA$  for the decoder weights:

$$W = W_O + \Delta W, \quad (5)$$

where  $W_O$  denotes the frozen pre-trained parameters and only  $\Delta W$  is updated during training. The forward propagation for a given input  $[h_c, h_{ct}]$  is then:

$$h = W_O[h_c, h_{ct}] + BA[h_c, h_{ct}], \quad (6)$$

where  $BA[h_c, h_{ct}]$  captures the task-specific knowledge learned during fine-tuning and augments the output of the frozen pre-trained parameters  $W_O[h_c, h_{ct}]$ . This design significantly reduces the number of trainable parameters and memory footprint, enabling MVRepair to perform vulnerability repair generation even on a single consumer-grade GPU. To ensure stable training,  $\Delta Wx$  is scaled by  $\alpha/\sqrt{r}$ , where  $\alpha$  is the rank of the low-rank matrices.

In our approach, MVRepair is deployed locally without connecting to online LLMs. During the decoding process, we utilized the Greedy Search algorithm [18], [19], which consistently selects the most optimal option at each step, allowing for a quick tracking of the search process. When generating code, if there is prior knowledge about the structure and logic of the code, Greedy Search can leverage this information to guide the search process, thereby enhancing the performance of the generated code.

## IV. EXPERIMENTAL SETUP

**Dataset.** Since there is no dataset specifically designed for memory-related vulnerabilities, we constructed a vulnerability dataset based on MegaVul [20]. The new dataset contains 13 common memory-related vulnerabilities (including CWE-119, -120, -121, -122, -124, -125, -126, -401, -415, -416, -476, -787, and -824), which were reported between January 2006 and October 2023. Table II presents the statistics of the dataset.

We followed the same procedures as VulRepair [11] and VRepair [13] to preprocess the dataset samples, where each vulnerable function is surrounded by two special tokens, i.e.,

TABLE II  
DATASET STATISTICS.

Type	Train	Valid	Test
Count	5692	802	1624
Samples with >512 tokens	797	115	202
Max number of tokens in a sample	2686	2396	2496

$\langle StartLoc \rangle$  and  $\langle EndLoc \rangle$ , and the repaired parts of the function are surrounded by  $\langle ModStart \rangle$  and  $\langle ModEnd \rangle$ , respectively. These notations inform the model about vulnerable code lines and their corresponding fixes.

**Baselines.** We compared MVRepair with four categories of baselines. The first is the *pre-trained model-based methods*, which use pre-trained code models widely used in code-related downstream tasks. Specifically, we investigated three encoder-based models (i.e., CodeBERT, GraphCodeBERT, and Roberta), three codec-based models (i.e., CodeReviewer, CodeT5, and Bart). The second is the  *$\ell$ LM-based methods*, which adopt various  $\ell$ LM models, each with fewer than 10 million parameters. Here, we considered CodeLlama-7b and three codec-based models (i.e., CodeT5+, CodeGen, and DeepSeek-Coder). For CodeGen, we analyzed three versions of varying sizes (i.e., 350M, 2B, and 6B). Meanwhile, we considered four variants of CodeT5+ with different sizes (i.e., 220M, 770M, 2B, and 6B). We also investigated two DeepSeek-Coder models with two sizes (i.e., 1.3B and 6.7B). The third is the *LLM-based methods*, involving well-known ChatGPT models (i.e., GPT-3.5-turbo and GPT-4-turbo) and DeepSeek models (i.e., DeepSeek-V3 and DeepSeek-R1) with prompts as described in Section III-A. The fourth is the *task-specific methods*, including three automatic vulnerability repair approaches (i.e., VulRepair, VRepair, and VulMaster).

**Evaluation Metrics.** We assessed MVRepair’s performance using three metrics: i) *Exact Match (EM)*, which determines if the predicted answer directly corresponds with the true answer, evaluating the fraction of model-generated answers that exactly align with the correct answer; ii) *BLEU-4*, which measures the resemblance between the model’s generated repair suggestions and the reference repairs, by calculating the exact matches of n-grams of various lengths to assess the quality of the generated repairs; and iii) *CodeBLEU*, which evaluates the quality of the generated code by incorporating syntactic and semantic aspects via AST and dataflow analysis. Typically, the higher the metric values, the better the repair performance we can achieve.

**Implementation Details.** All experiments were conducted on an Ubuntu server featuring two Intel Xeon CPUs, 128 GB of memory, and eight NVIDIA GeForce RTX 4090 GPUs. For the pre-trained model-based methods, we utilized publicly available source code with their original hyperparameters. For the  $\ell$ LM-based methods, all models are downloaded from HuggingFace. We conducted experiments using OpenAI’s public APIs (i.e., “GPT-3.5-turbo” and “GPT-4-turbo.”) with initial parameters provided by OpenAI. To ensure fairness in the experiment, we applied the same data segmentation for all approaches. In this paper, we focused on addressing vulnerabilities in C/C++, having gathered instances of these vulnerabilities from the CWE website [3]. Note that our approach can be easily applied to other programming languages by simply replacing

the underlying datasets. Throughout the training, we used the Adam optimizer with a learning rate of  $1e-4$ . A weight decay rate of 0.01 was applied. We used a batch size of 1, and the training spanned 20 epochs. According to the work in [12], we set the maximum length of a function segment to 512.

## V. EXPERIMENTAL RESULTS

We compared MVRepair with state-of-the-art (SOTA) vulnerability repair methods, aiming to answer the following two Research Questions (RQs): i) what is the **superiority** of MVRepair compared with SOTA vulnerability repair methods? and ii) how is the **effectiveness** of the proposed key components on the performance of MVRepair?

### A. Performance Evaluation (RQ1)

RQ1 aims to evaluate the performance of MVRepair against all baselines on our dataset for memory-related vulnerabilities.

TABLE III  
MODEL PERFORMANCE OF BASELINES AND MVREPAIR.

Model	Model Size	EM	BLEU-4	CodeBLEU
CodeBERT	125M	3.1	2.9	9.4
GraphCodeBERT	173M	4.1	1.7	12.1
Roberta	125M	3.4	2.6	7.2
Bart	140M	3.9	13.3	27.4
Codereviewer	173M	5.1	9.3	21.7
CodeT5	223M	6.7	14.1	25.6
CodeT5+	220M	7.3	10.7	22.1
CodeT5+	770M	8.9	18.2	29.3
CodeT5+	2B	9.9	19.1	30.5
CodeT5+	6B	10.8	19.3	31.8
CodeGen	350M	7.2	14.9	24.3
CodeGen	2B	7.3	15.4	32.2
CodeGen	6B	10.7	27.2	34.3
DeepSeek-Coder	1.3B	8.6	12.5	27.8
DeepSeek-Coder	6.7B	11.0	17.9	31.8
CodeLLAMA	7B	11.2	20.8	31.8
GPT-3.5	NA	4.6	5.3	17.5
GPT-4	NA	7.8	10.0	21.2
DeepSeek-V3	NA	5.4	6.8	18.3
DeepSeek-R1	NA	8.1	9.4	20.7
VRRepair	NA	4.6	8.1	30.3
VulRepair	NA	6.7	14.1	25.6
VulMaster	NA	14.7	23.2	33.5
MVRepair	7B	<b>18.2</b>	<b>30.6</b>	<b>39.1</b>

Table III compares the performance between MVRepair and all baseline models, with the best results highlighted in bold. We can find that MVRepair outperforms all supervised baseline methods in terms of EM, BLEU-4, and CodeBLEU scores on the evaluation dataset. Compared to VulMaster, MVRepair improves the scores of EM, BLEU, and CodeBLEU by 23.8%, 31.9%, and 16.7%, respectively. We performed a t-test on CodeBLEU using the paired data between MVRepair and each of the baseline models, and all comparisons yielded p-values less than or equal to 0.0004, suggesting that the difference in performance between MVRepair and the baseline models is statistically significant. Furthermore, we can observe that language models like GPT-3.5 and GPT-4 struggle to effectively address memory-related software vulnerabilities, due to their insufficient accuracy and reliability in understanding and managing complex memory issues. Since GPT-3.5 and GPT-4 are designed for question-and-answer tasks, under-utilization of their training data will hinder their ability to update parameters and learn vulnerability patterns effectively. Note that GPT-4 performs significantly worse than our tool on all the metrics.

TABLE IV  
PERFECT PREDICTION FOR THE FULL RANGE OF CWES.

Group	CWE-Type	VRepair	VulRepair	VulMaster	MVRepair	# of Samples
A	CWE-119	12	17	39	54	340
	CWE-120	4	7	17	20	86
	CWE-121	1	0	1	4	9
	CWE-122	0	0	5	6	23
B	CWE-125	16	9	55	62	253
C	CWE-126	0	0	0	0	1
	CWE-401	2	8	14	15	61
D	CWE-415	2	2	4	5	33
	CWE-416	10	14	29	33	197
	CWE-476	12	30	38	46	273
E	CWE-787	15	23	38	52	343
	CWE-824	0	0	0	0	5
Total		74 (4.6%)	110 (6.7%)	240 (14.7%)	297 (18.2%)	1624

As shown in Figure 4, we categorized the test set into five groups based on the types of memory operation errors and resource management issues associated with the vulnerabilities. The categories include A: Buffer overflows, B: Out-of-bounds writes, C: Out-of-bounds reads, D: Memory management errors, and E: Null pointers and post-destruction references. Figure 4 shows the EM performance for each group, where MVRepair achieves the best result. For example, MVRepair

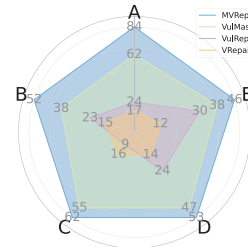


Fig. 4. Model performance on different groups.

fixes 84 vulnerabilities in Group A, which is a 35.5% improvement over VulMaster. Table IV presents the number of perfect predictions made by MVRepair for all CWE-types. Although MVRepair’s fix rate of 18.2% is not particularly high, the improvement over SOTA is significant. This indicates that our approach plays a crucial role in addressing memory-related software vulnerabilities. Additionally, we note that VulMaster performs worse in the out-of-bounds write category of vulnerabilities compared to other categories, because out-of-bounds write vulnerabilities typically involve more complex memory management and code logic challenges. However, MVRepair demonstrates comparable performance across all groups of vulnerabilities, showcasing the effectiveness of our approach to memory-related vulnerabilities.

### B. Ablation Study (RQ2)

RQ2 aims to examine the effects of the components of MVRepair, including the SST building module, the ASEG module, and the CWE-type knowledge acquisition module. We conducted two ablation studies using different models, input component design, and DL model design. In each ablation study, we removed one component at a time to analyze its individual contribution to the key design. Specifically, we considered three variants: i) *MVRepair w/o ASEG*, where only the first 512 tokens are intercepted, removing the subsequent parts of the code; ii) *MVRepair w/o SST*, which eliminates the semantic sampling tree token information of the code; and

iii) *MVRepair w/o CWE-type prediction*, which eliminates the CWE-type prediction module, ignoring prior knowledge of the vulnerability type and directly fixing vulnerable code.

TABLE V  
ABLATION STUDY RESULTS FOR MVREPAIR.

Type	Variant	EM	BLEU-4	CodeBLEU
Full Model	MVRepair	<b>18.2</b>	<b>30.6</b>	<b>39.1</b>
Input Components	w/o ASEG	14.1	27.5	35.4
	w/o SST	14.7	25.4	34.4
Model Designs	w/o CWE-type Prediction	14.5	25.2	34.8

From Table V, we can observe that each design component enhances the performance of MVRepair. Specifically, the removal of the SST component results in decreases of 25.5%, 8.0%, and 7.3% in the EM, BLEU-4, and CodeBLEU scores, respectively. Additionally, eliminating the ASEG component leads to a 16.9% drop in the CodeBLEU score, which negatively affects the semantic fluency of the generated repair code. CWE-type prediction is crucial in the *tool* repair process. Without obtaining prior knowledge of CWE types, there can be a significant decrease in EM metrics, by as much as 22.1%, which severely impacts the accuracy of the repair code. Additionally, it is noteworthy that the SST module is the most effective component in this context. In this context, EM refers to the percentage of generated code that matches the token sequence of the base fact. This alignment is particularly important for the code repair task, which is why we prioritize the EM metric as our main evaluation criterion. We observed that integrating a specific SST with a memory-relevant repair corpus led to a substantial enhancement in the performance of our tool.

### C. Case Study

Figure 5 presents a CWE-416 vulnerability case and the corresponding fixes generated by VulMaster [12] (lines 11) and MVRepair (lines 10), respectively. In the original code (lines 8–9), the function `init_new_context` calls `init_new_context_ldt` but does not handle its return value. If `init_new_context_ldt` fails and returns a non-zero value, the caller of `init_new_context` cannot detect this failure, which may introduce a security vulnerability.

```

CWE-416: Function return values are not handled correctly
1 static inline int init_new_context(struct task_struct *tsk,
2 struct mm_struct *mm)
3 {
4     if (cpu_feature_enabled(X86_FEATURE_OSPKE)) {
5         mm->context.pkey_allocation_map = 0x1;
6         mm->context.execute_only_pkey = -1;
7     }
8 -   init_new_context_ldt(tsk, mm);
9 -   return 0;
10 +  return init_new_context_ldt(tsk, mm);
11 +  return init_new_context_ldt(mm);
12 }

```

Fig. 5. Example of repairs by VulMaster and MVRepair.

MVRepair correctly generates the repair in line 10 by returning the result of `init_new_context_ldt`, ensuring that the success or failure of the inner function is visible to the caller. This demonstrates MVRepair’s ability to capture the semantic dependency between function outcomes and propagate error-handling logic. In contrast, VulMaster generates an incorrect patch (line 11) that omits the `tsk` parameter, since it was

defined but unused elsewhere in the function. This highlights a limitation of purely token-based segmentation approaches: by losing contextual cues about parameter significance, they may overlook the correct propagation of return values. From this case, we can find that MVRepair utilizes knowledge-guided analysis to preserve critical semantics and generate a patch that is not only syntactically correct but also functionally aligned with proper error-handling practices.

## VI. RELATED WORK

**Static Analysis-Based.** Static analysis-based methods aim to select the most appropriate algorithm depending on the intrinsic static characteristics of a vulnerability’s root cause, allowing for identification and resolution of the root code without executing the program. Shaw et al. [21] discovered that buffer overflow vulnerabilities typically stem from using unsafe functions. Thus, they developed a pointer analysis technique to evaluate buffer sizes and substitute insecure functions with secure counterparts. Ma et al. [22] introduced a static slicing-based algorithm to optimize the balance between analysis precision and efficiency. Nonetheless, the intricate programming logic found in actual projects complicates static identification and diminishes traditional methods’ effectiveness.

**Learning-Based.** Various deep learning-based methods have been investigated to autonomously identify explicit or implicit vulnerability features from known vulnerabilities, aiming to broaden the scope of learning-based program repair. For example, Fu et al. [23], however, noticed an absence of a mechanism to guide their model to focus more on vulnerable code areas during repair. To address this, they introduced the Vulnerability Query (VQ) for locating vulnerable code sections and employed a cross-attention mechanism for cross-matching between the VQ and the relevant vulnerable code region. Zhou et al. [12] introduced VulMaster, which fixes vulnerabilities by thoroughly comprehending the entire vulnerable code, regardless of its size. Our method distinguishes itself from current learning-based vulnerability remediation approaches by utilizing fully functional code data to aid in the remediation of memory-related vulnerabilities through innovative code SSTs.

## VII. CONCLUSION

This paper presents a novel LLM-based vulnerability repair method that effectively addresses the low repair performance limitations posed by existing LMs’ code segmenting methods. Rather than arbitrarily segmenting the target program code at the token level, this paper proposes to segment the code at the grammar level based on memory-related code structural information and CWE-type a priori knowledge. Comprehensive experiments on a real-world dataset show the superiority of MVRepair against SOTA methods.

## ACKNOWLEDGEMENT

This work was supported by Natural Science Foundation of China (62272170) and Shanghai Trusted Industry Internet Software Collaborative Innovation Center. Ming Hu (hu.ming.work@gmail.com) and Mingsong Chen (mschen@sei.ecnu.edu.cn) are the corresponding authors.

## REFERENCES

- [1] Jian Gao, Yiwen Xu, Yu Jiang, Zhen Liu, Wanli Chang, Xun Jiao, and Jiaguang Sun. Em-fuzz: Augmented firmware fuzzing via memory checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3420–3432, 2020.
- [2] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, Xiaoxue Wu, Chuanqi Tao, Tao Zhang, and Wei Liu. Learning to detect memory-related vulnerabilities. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–35, 2023.
- [3] CWE. <https://cwe.mitre.org/>, 2023.
- [4] Adib Nahiyani, Kan Xiao, Kun Yang, Yeir Jin, Domenic Forte, and Mark Tehranipoor. Avfsm: A framework for identifying and mitigating vulnerabilities in fsms. In *Proceedings of the Annual Design Automation Conference (DAC)*, pages 1–6, 2016.
- [5] Gustavo K Contreras, Adib Nahiyani, Swarup Bhunia, Domenic Forte, and Mark Tehranipoor. Security vulnerability analysis of design-for-test exploits for asset protection in socs. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 617–622, 2017.
- [6] Chathura Rajapaksha, Leila Delshadtehrani, Richard Muri, Manuel Egele, and Ajay Joshi. IOMMU deferred invalidation vulnerability: Exploit and defense. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2024.
- [7] Muhammad Monir Hossain, Arash Vafaei, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. Socfuzzer: SoC vulnerability detection using cost function enabled fuzz testing. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023.
- [8] Sree Ranjani Rajendran, Farimah Farahmandi, and Mark Tehranipoor. Cad tools pathway in hardware security. In *Proceedings of VLSI Design and Embedded Systems (VLSID)*, pages 342–347, 2024.
- [9] Linus Feiten, Matthias Sauer, Tobias Schubert, Victor Tomashevich, Iliia Polian, and Bernd Becker. Formal vulnerability analysis of security components. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1358–1369, 2015.
- [10] Feilong Zuo, Zhengxiong Luo, Junze Yu, Ting Chen, Zichen Xu, Aiguo Cui, and Yu Jiang. Vulnerability detection of ics protocols via cross-state fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4457–4468, 2022.
- [11] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 935–947, 2022.
- [12] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the international conference on software engineering (ICSE)*, pages 1–13, 2024.
- [13] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1):147–165, 2022.
- [14] Tree-sitter. In <https://github.com/tree-sitter>.
- [15] Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N. Nguyen. A learning-based approach to static program slicing. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, volume 8, pages 83–109, 2024.
- [16] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- [17] Jienan Chen, Jun Tu, Hao Wang, Jie Zheng, Hao Liu, Shenglong Bai, Xiantuo He, and Weikang Qian. GPCB routing: Generative pretrained transformers-based printed circuit board routing method. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(4):1420–1433, 2025.
- [18] Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. Developer-intent driven code comment generation. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 768–780, 2023.
- [19] Hanyang Guo, Xiangping Chen, Yuan Huang, Yanlin Wang, Xi Ding, Zibin Zheng, Xiaocong Zhou, and Hong-Ning Dai. Snippet comment generation based on code context expansion. *ACM Transactions on Software Engineering and Methodology*, 33(1):1–30, 2023.
- [20] Chao Ni, Liyu Shen, Xiaohu Yang, Yan Zhu, and Shaohua Wang. Megavul: Ac/c++ vulnerability dataset with comprehensive code representations. In *Proceedings of International Conference on Mining Software Repositories (MSR)*, pages 738–742, 2024.
- [21] Alex Shaw, Dusten Doggett, and Munawar Hafiz. Automatically fixing c buffer overflows using program transformations. In *Proceedings of Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 124–135, 2014.
- [22] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. Cdrep: Automatic repair of cryptographic misuses in Android applications. In *Proceedings of the ACM on Asia conference on computer and communications security (AsiaCCS)*, pages 711–722, 2016.
- [23] Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. Vision transformer inspired automated vulnerability repair. *ACM Transactions on Software Engineering and Methodology*, 33(3):1–29, 2024.