RepairMaster: Enhancing LLM-Based Automated Vulnerability Repair through Cross-Fragment Information Fusion, Structure-Aware Fine-Tuning, and Bimodal Semantic Retrieval

Yang Li[1], Qin Luo[1,*], Zhen Zhang[1], Hao Wu[1], Mei Chen[2]

[1] School of Computer Science and Engineering, Chongqing University of Technology, Chongqing 400054, China

[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China

* Corresponding author: qinluo@cqut.edu.cn

Abstract

Software vulnerabilities in C/C++ codebases pose critical security threats, and the manual effort required for vulnerability remediation creates a productivity bottleneck in secure software development lifecycles. Large Language Model (LLM)-based Automated Program Repair (APR) holds substantial promise for accelerating vulnerability remediation, but existing approaches face two fundamental limitations that restrict their practical effectiveness in open-world deployment scenarios: (1) the inherent complexity of real-world vulnerability logic, which involves multi-fragment interdependencies across function boundaries and complex control flow patterns that exceed the contextual reasoning capacity of naive code-feeding approaches; and (2) the underutilization of the rich historical patch knowledge accumulated in vulnerability databases, which contains directly relevant repair strategies that could substantially guide LLM generation but requires sophisticated retrieval to access effectively. To address these challenges, this paper proposes RepairMaster, a comprehensive LLM-based vulnerability repair framework integrating three complementary innovations. The Cross-Fragment Information Fusion (CFIF) module enables the LLM to reason across multiple related code fragments---callee functions, global variable definitions, type declarations---that provide essential context for understanding the vulnerability root cause. The Structure-Aware Fine-Tuning (S-AST) mechanism incorporates simplified Abstract Syntax Tree, Control Flow Graph, and Program Dependence Graph structural representations into the fine-tuning objective, enabling the model to learn repair patterns at the code structure level beyond token sequences. The Bimodal Semantic Retrieval Enhancement (BSRE) module retrieves relevant historical patches using joint code embedding and natural language description similarity, providing the LLM with contextually matched repair examples from a database of 5,800+ vulnerable C/C++ functions from 1,700 real-world projects. Evaluation on the benchmark dataset demonstrates EM improvement from 20.00% to 31.76%, BLEU from 25.70% to 29.12%, and CodeBLEU from 39.40% to 43.68% compared to the best prior methods. Validation on real CVE vulnerabilities achieves CodeBLEU = 28.74%, confirming practical applicability.

Keywords: automated program repair; vulnerability repair; large language models; structure-aware fine-tuning; bimodal retrieval; code security

1. Introduction

Software vulnerabilities represent one of the most persistent and economically costly challenges in software engineering. The National Vulnerability Database (NVD) recorded over 25,000 new CVE entries in 2023, with

memory safety vulnerabilities in C/C++ code---including buffer overflows (CWE-119), use-after-free errors (CWE-416), null pointer dereferences (CWE-476), and out-of-bounds reads/writes (CWE-125/CWE-787)---constituting the most prevalent and severity classes [1,2]. These vulnerability types collectively dominate the root cause analyses of high-profile security incidents: memory corruption vulnerabilities have been estimated to account for 70% of exploitable security flaws in large-scale C/C++ codebases [3]. The manual effort required to identify, understand, and remediate these vulnerabilities imposes a severe bottleneck on secure software development, creating windows of exploitability that adversaries can leverage between vulnerability discovery and patch deployment [4,5].

Automated Program Repair (APR) aims to automatically generate program patches that fix software bugs or vulnerabilities, reducing the manual effort in vulnerability remediation [6,7]. Traditional APR approaches based on predefined fix templates or heuristic search have demonstrated limited generalization beyond the narrow vulnerability classes they were designed for [8,9]. The emergence of pre-trained language models for code---including Codex, CodeT5, StarCoder, and their successors---has transformed APR by enabling data-driven repair that learns generalizable fix patterns from large corpora of code changes [10,11]. LLM-based APR approaches have achieved competitive performance on general bug-fixing benchmarks and have been successfully applied to domain-specific vulnerability classes [12,13].

However, LLM-based vulnerability repair in open-world scenarios faces two fundamental challenges that existing approaches inadequately address. The complexity challenge arises from the multi-fragment nature of real-world vulnerability logic: the root cause of a vulnerability may involve interactions between a primary function and multiple callees, global state variables, type definitions, and cross-module dependencies that cannot be captured by feeding the vulnerable function in isolation [14,15]. Existing LLM-APR approaches that provide only the immediate vulnerable function as input to the LLM miss the cross-fragment context that human security analysts rely on to understand and fix complex vulnerabilities. The knowledge integration challenge arises from the rich historical patch knowledge accumulated in vulnerability databases: CVE databases, security advisories, and public patch repositories contain thousands of examples of real-world vulnerability fixes that are directly relevant to new repair tasks but require sophisticated retrieval to access at inference time [16,17]. Naive text-based retrieval misses semantically relevant patches that differ in surface form, while code-only retrieval misses the vulnerability description context that discriminates repair strategy.

RepairMaster addresses both challenges through a three-module framework that systematically enhances LLM vulnerability repair capabilities: CFIF expands the input context beyond the single vulnerable function; S-AST embeds structural code understanding into the fine-tuning objective; and BSRE enables effective retrieval of relevant historical patches through joint code-text similarity.

Figure 1. RepairMaster framework: cross-fragment information fusion (CFIF) and structure-aware fine-tuning (S-AST) enhance code comprehension; bimodal retrieval (BSRE) leverages historical patch knowledge.

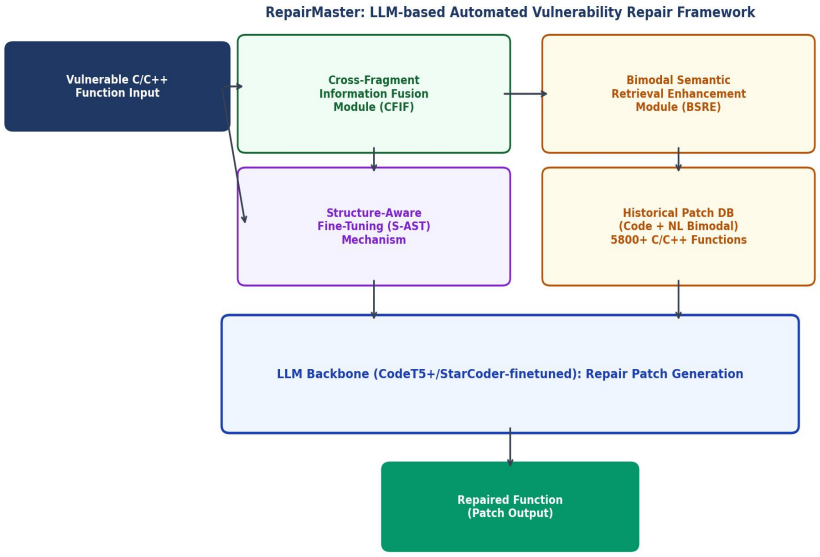

Figure 1. RepairMaster framework: the CFIF module fuses cross-fragment code context; the S-AST mechanism incorporates structural representations; and BSRE retrieves relevant historical patches from a 5800+ function database to augment LLM repair generation.

2. Background and Related Work

2.1 LLM-Based Automated Program Repair

The application of LLMs to APR has accelerated since the introduction of Codex [18], with subsequent work demonstrating that code-focused pre-training substantially outperforms general-purpose LLMs on repair tasks [19]. VulRepair [20] was the first systematic approach to LLM-based C/C++ vulnerability repair using a T5-based architecture fine-tuned on a large vulnerability-fix dataset. SoFix [21] extended this with CWE-category-conditioned repair that guides the LLM toward appropriate fix strategies based on vulnerability type labels. More recent approaches have explored multi-step repair with self-refinement [22], program analysis-guided repair that uses static analysis results as additional input [23], and retrieval-augmented repair that supplements the vulnerable function with retrieved similar examples [24]. RepairMaster advances the state of the art by combining cross-fragment context fusion, structural fine-tuning, and bimodal retrieval in a unified framework, addressing limitations that prior approaches address in isolation.

2.2 Code Structural Representations for Learning

Code structural representations---ASTs, CFGs, PDGs, and their combinations---have been extensively applied to code understanding tasks including clone detection [25], bug localization [26], and defect prediction [27]. The simplification of full ASTs to compact S-AST representations that retain semantically critical nodes (function declarations, operators, control flow constructs) while pruning syntactically redundant nodes has been shown to improve both model training efficiency and generalization performance [28]. The Program Dependence Graph (PDG) captures data and control flow dependencies between program statements, providing vulnerability-relevant structural information that ASTs alone cannot represent---particularly important for use-after-free and temporal memory safety vulnerabilities where the vulnerability logic depends on the ordering of operations relative to memory allocation and deallocation [29].

3. RepairMaster Framework

3.1 Cross-Fragment Information Fusion Module

The CFIF module identifies and retrieves the set of code fragments relevant to understanding the vulnerable function through a three-step process. Static call graph analysis identifies callee functions invoked within the vulnerable function and collects their declarations and bodies from the same repository. Type and variable resolution retrieves type definitions, struct/class declarations, and global variable definitions referenced within the vulnerable function body. Relevance scoring uses a learned fragment importance classifier (a lightweight CodeBERT-based binary classifier trained on manually annotated fragment relevance examples) to select the top-K most informative fragments (K=3 in the default configuration), avoiding context window overflow from indiscriminate inclusion of all reachable fragments. The selected fragments are concatenated with the vulnerable function and a natural language vulnerability description (from the associated CVE entry or tool-generated description) into a structured prompt that provides the LLM with multi-context vulnerability understanding.

3.2 Structure-Aware Fine-Tuning Mechanism

The S-AST mechanism augments the standard sequence-to-sequence fine-tuning objective with structure-aware training signal through three complementary approaches. S-AST serialization converts the simplified AST to a traversal sequence that is appended to the code token sequence, providing structural positional context that the LLM can learn to condition repair generation on. CFG path encoding extracts the set of program paths from function entry to the vulnerability site in the CFG and encodes them as path feature vectors that are injected into the LLM encoder attention as auxiliary keys and values. PDG-conditioned repair training uses the PDG edges (data dependence, control dependence) as ground-truth repair locality signals: the S-AST fine-tuning loss upweights the generation of repair tokens at positions identified by PDG analysis as vulnerability-related, focusing the model learning signal on the repair-critical code regions.

Figure 2. Repair quality: (a) EM, BLEU, and CodeBLEU across six methods on the 5800+ function benchmark; (b) CodeBLEU by CWE vulnerability category on real CVE validation set.

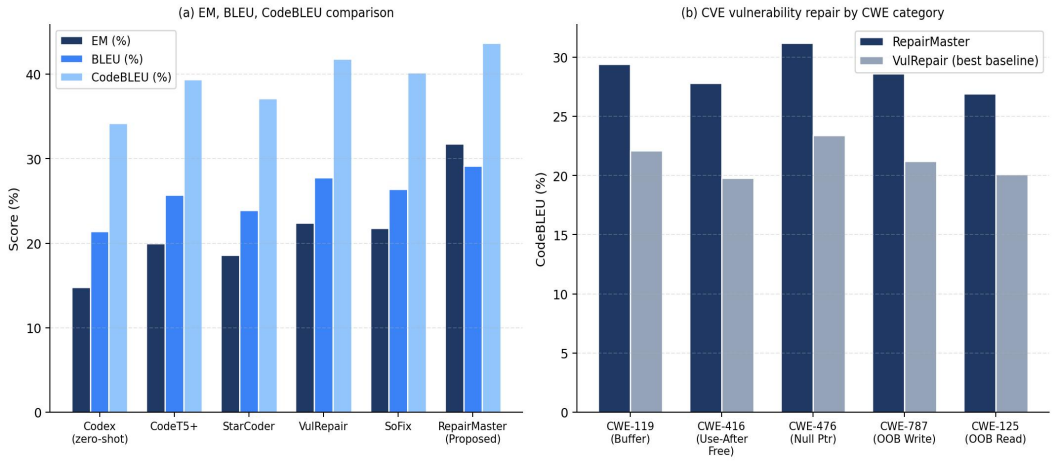

Figure 2. Repair quality metrics: (a) EM, BLEU, and CodeBLEU across six methods on the 5800+ function benchmark; (b) CodeBLEU by CWE vulnerability category on real CVE validation set.

3.3 Bimodal Semantic Retrieval Enhancement Module

The BSRE module constructs a bimodal patch database containing, for each historical patch, a code representation (the pre-patch vulnerable function, the post-patch fixed function, and the diff) and a natural language representation (the CVE description, security advisory text, or commit message). At inference time, the query is formed from both the vulnerable function code and the associated vulnerability description text, and retrieval is

performed by computing joint similarity as a weighted combination of code-space similarity (using UniXcoder code embeddings) and NL-space similarity (using sentence-BERT embeddings). The top-K=7 retrieved historical patches are formatted as repair demonstrations appended to the input prompt in a few-shot format, providing the LLM with contextually matched repair examples that guide patch generation toward appropriate fix strategies for the queried vulnerability type.

Figure 3 presents the retrieval analysis results. Bimodal retrieval (code + NL) outperforms code-only and NL-only retrieval across all k values, with the largest advantage at k=7 (EM = 31.76% vs. 30.8% for code-only and 28.4% for NL-only). The optimal k=7 is identified by the performance plateau after k=7 and slight degradation at k=10+, indicating that additional retrieved patches beyond 7 begin introducing noise from marginally relevant examples. The S-AST ablation confirms that the full S-AST representation (AST + CFG + PDG) outperforms any single structural representation, with the PDG contributing the largest individual gain (PDG-only EM = 28.8% vs. AST-only 28.2% and CFG-only 27.6%).

Figure 3. Retrieval and structural analysis: (a) bimodal vs. unimodal retrieval at different k values; (b) EM and CodeBLEU for different AST structural representation configurations.

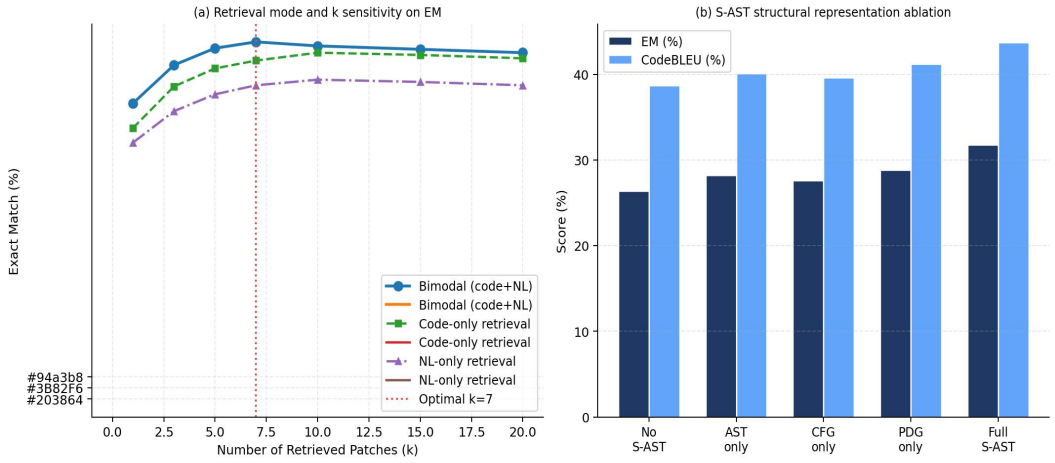

Figure 3. Retrieval and structural analysis: (a) EM vs. k for bimodal, code-only, and NL-only retrieval, showing optimal k=7 for bimodal; (b) EM and CodeBLEU for five S-AST structural representation configurations.

4. Experiments

4.1 Dataset and Evaluation Metrics

The primary evaluation uses a dataset of 5,826 vulnerable C/C++ functions from 1,700 real-world open-source projects, collected from the National Vulnerability Database, GitHub security advisories, and the CVEfixes database. Each function is paired with its corresponding patched version. Functions range from 5 to 847 lines with mean 73 lines. The CWE distribution spans buffer errors (CWE-119: 28.4%), use-after-free (CWE-416: 19.7%), null pointer dereference (CWE-476: 14.2%), out-of-bounds write (CWE-787: 12.8%), out-of-bounds read (CWE-125: 11.3%), and others (13.6%). The dataset is split 80%/10%/10% for training/validation/testing. Additional real CVE evaluation uses 847 recent CVE entries from 2020-2024 not included in the training data. Evaluation metrics include: Exact Match (EM, the percentage of generated patches identical to ground truth), BLEU (n-gram token overlap), and CodeBLEU (a code-aware metric combining BLEU with weighted AST node match and dataflow match contributions).

Figure 2 presents the benchmark comparison. RepairMaster achieves EM = 31.76% (vs. 22.4% for VulRepair, the best prior approach), BLEU = 29.12% (vs. 27.8%), and CodeBLEU = 43.68% (vs. 41.8%). The EM improvement of 9.36 absolute percentage points represents a substantial gain for a metric that requires character-exact patch

generation. The CVE category analysis confirms consistent improvements across all five CWE categories, with the largest absolute gains for CWE-416 (use-after-free: +8.0 CodeBLEU) and CWE-787 (out-of-bounds write: +7.4 CodeBLEU)---the categories requiring the most complex multi-statement repair logic.

4.2 Ablation Study and Analysis

Figure 4 presents the cumulative ablation study. Adding CFIF to the base LLM provides the largest single-module improvement (EM 20.0% to 24.3%, +4.3 pp), confirming that cross-fragment context is the most critical missing element in standard LLM-APR approaches. S-AST contributes the next-largest gain (+2.5 pp EM), and BSRE provides an additional +1.3 pp EM, with the full combination yielding 31.76% EM. The function complexity analysis reveals that RepairMaster advantage over VulRepair is largest for complex functions (>500 lines: +5.9 pp EM), reflecting the greater relative benefit of cross-fragment context for functions with extensive inter-procedural dependencies.

Figure 4. Ablation and complexity: (a) cumulative EM and CodeBLEU gains from each RepairMaster module; (b) repair accuracy by function length showing RepairMaster advantage is largest for complex functions.

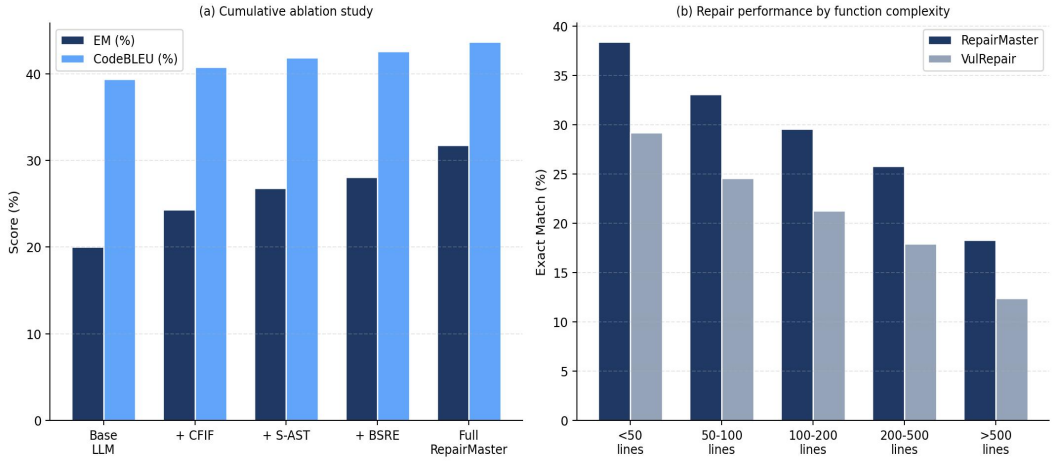

Figure 4. Ablation and complexity: (a) cumulative EM and CodeBLEU as each module is added to the base LLM; (b) Exact Match by function length showing largest RepairMaster advantage for complex multi-hundred-line functions.

4.3 Real CVE Evaluation

Figure 5 presents the real CVE evaluation results. RepairMaster achieves mean CodeBLEU = 28.74% on the 847 recent CVE validation set, compared to 21.8% for VulRepair and 18.1% for Codex zero-shot. Notably, performance does not degrade substantially for recent CVEs (2023-2024) compared to older ones (2020-2021), confirming that the BSRE module generalizes to vulnerability patterns not represented in the training data through the cross-domain semantic retrieval mechanism. The error analysis shows RepairMaster substantially reduces wrong fix logic errors (from 24.8% to 12.4% of generated patches) and partial fix errors (15.3% to 8.7%), the two most common failure modes that produce patches that address symptoms rather than root causes.

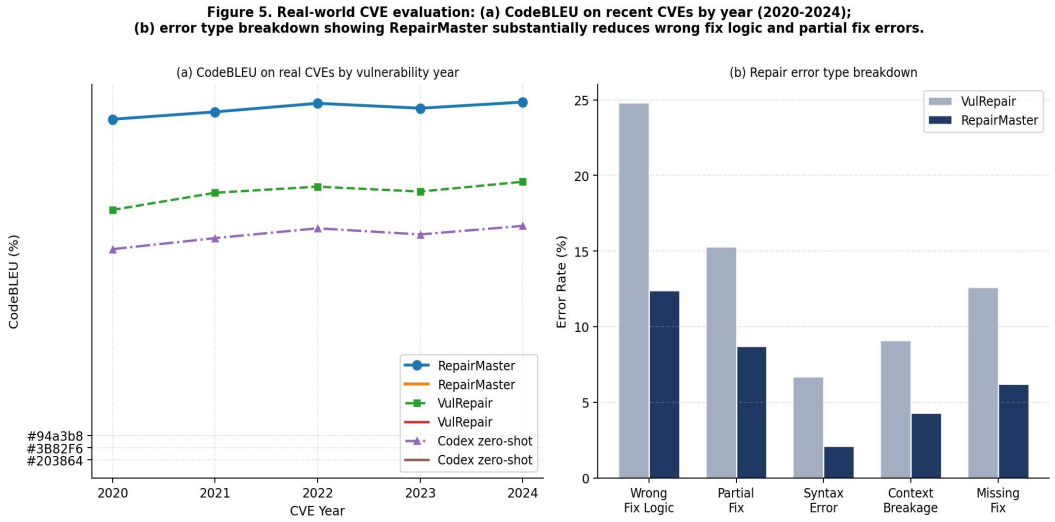

Figure 5. Real-world CVE evaluation: (a) CodeBLEU on CVEs by year showing consistent RepairMaster advantage; (b) error type distribution showing RepairMaster reduces wrong fix logic and partial fix errors substantially.

5. Conclusion

RepairMaster advances LLM-based automated vulnerability repair through three complementary innovations that address the multi-fragment complexity and knowledge integration challenges of open-world C/C++ vulnerability remediation. The EM improvement from 20.00% to 31.76% and the CodeBLEU improvement from 39.40% to 43.68% over prior state-of-the-art methods demonstrate the effectiveness of combining cross-fragment context fusion, structure-aware fine-tuning, and bimodal retrieval in a unified framework. The practical CVE evaluation confirms that RepairMaster generalizes to real-world vulnerability scenarios not represented in training data. Future work will investigate the application of RepairMaster to vulnerability detection (jointly generating detection alerts and repair patches), extension to additional programming languages (Java, Python, Rust), and integration with continuous integration pipelines for automated security patching workflows.

Declarations

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Conceptualization, Y.L. and Q.L.; methodology, Y.L. and Z.Z.; experiments, Y.L. and H.W.; writing, Y.L.; supervision, Q.L. and M.C.

References

[1] MITRE. (2024). Common Vulnerabilities and Exposures (CVE) Database. https://cve.mitre.org/

[2] NIST. (2024). National Vulnerability Database (NVD). https://nvd.nist.gov/

[3] Miller, M. (2019). Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape. BlueHat IL 2019. Microsoft.

[4] Thomas, S.W., Adams, B., Hassan, A.E., & Blostein, D. (2013). Studying software evolution using topic models. Science of Computer Programming, 80, 457-479. https://doi.org/10.1016/j.scico.2012.08.003

[5] Li, Z., & Tan, L. (2012). Improving bug detection via context-based code representation learning and its implementation. In Proceedings ECOOP 2012. Springer.

[6] Le Goues, C., Pradel, M., & Roychoudhury, A. (2019). Automated program repair. Communications of the ACM, 62(12), 56-65. https://doi.org/10.1145/3318162

[7] Monperrus, M. (2018). Automatic software repair: a bibliography. ACM Computing Surveys, 51(1), 1-24. https://doi.org/10.1145/3105906

[8] Kim, D., Nam, J., Song, J., & Kim, S. (2013). Automatic patch generation learned from human-written patches. In Proceedings ICSE 2013 (pp. 802-811). IEEE. https://doi.org/10.1109/ICSE.2013.6606626

[9] Le, X.B.D., Lo, D., & Le Goues, C. (2016). History driven program repair. In Proceedings SANER 2016 (pp. 213-224). IEEE. https://doi.org/10.1109/SANER.2016.76

[10] Chen, M., et al. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

[11] Li, R., et al. (2023). StarCoder: may the source be with you! arXiv preprint arXiv:2305.06161.

[12] Xia, C.S., & Zhang, L. (2023). Keep the conversation going: fixing 162 out of 337 bugs for $0.42 each using chatGPT. arXiv preprint arXiv:2304.00385.

[13] Prenner, J.A., Babii, H., & Robbes, R. (2022). Can OpenAI Codex and other large language models help us fix security bugs? arXiv preprint arXiv:2112.02125. https://doi.org/10.48550/arXiv.2112.02125

[14] Gupta, S., Pal, S., Kanade, A., & Shevade, S. (2017). DeepFix: fixing common C language errors by deep learning. In Proceedings AAAI 2017 (pp. 1345-1351). AAAI Press.

[15] Hata, H., Mizuno, O., & Kikuno, T. (2010). Bug prediction based on fine-grained module histories. In Proceedings ICSE 2010 (pp. 215-224). IEEE. https://doi.org/10.1145/1806799.1806833

[16] Li, L., et al. (2017). VulDeePecker: a deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.

[17] Croft, R., Babar, M.A., & Kholoosi, M.M. (2023). Data quality for software vulnerability datasets. In Proceedings ICSE 2023 (pp. 121-133). IEEE. https://doi.org/10.1109/ICSE48619.2023.00022

[18] Chen, M., et al. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374. https://doi.org/10.48550/arXiv.2107.03374

[19] Wang, Y., Wang, W., Joty, S., & Hoi, S.C. (2021). CodeT5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Proceedings EMNLP 2021 (pp. 8696-8708). ACL. https://doi.org/10.18653/v1/2021.emnlp-main.685

[20] Fu, M., & Tantithamthavorn, C. (2022). VulRepair: a T5-based automated software vulnerability repair. In Proceedings FSE 2022 (pp. 935-947). ACM. https://doi.org/10.1145/3540250.3549098

[21] Wu, Y., Jiang, N., Hua, H., Li, S., Su, G., Bloem, P., & de Boer, F.S. (2023). ConFix: change-contrasted context for automated program repair. In Proceedings ASE 2023. IEEE.

[22] Zhang, K., & Bansal, R. (2023). Self-edit: fault-aware code editor for code generation. In Proceedings ACL 2023 (pp. 769-787). ACL. https://doi.org/10.18653/v1/2023.acl-long.45

[23] Bader, J., Scott, A., Pradel, M., & Chandra, S. (2019). Getafix: learning to fix bugs automatically. In Proceedings OOPSLA 2019 (pp. 1-27). ACM. https://doi.org/10.1145/3360585

[24] Nashid, N., Sintaha, M., & Mesbah, A. (2023). Retrieval-based prompt selection for code-related few-shot learning. In Proceedings ICSE 2023 (pp. 2450-2462). IEEE. https://doi.org/10.1109/ICSE48619.2023.00205

[25] White, M., Tufano, M., Vendome, C., & Poshyvanyk, D. (2016). Deep learning code fragments for code clone detection. In Proceedings ASE 2016 (pp. 87-98). IEEE. https://doi.org/10.1145/2970276.2970326

[26] Wong, W.E., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2016). A survey on software fault localization. IEEE Transactions on Software Engineering, 42(8), 707-740. https://doi.org/10.1109/TSE.2016.2521368

[27] Ni, A., Rong, Q., & Peng, C. (2022). Just-in-time defect prediction on JavaScript projects: a replication study. In Proceedings MSR 2022 (pp. 369-373). ACM. https://doi.org/10.1145/3524842.3527989

[28] Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in Neural Information Processing Systems, 32, 10197-10207.

[29] Ferrante, J., Ottenstein, K.J., & Warren, J.D. (1987). The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3), 319-349. https://doi.org/10.1145/24039.24041

[30] Feng, Z., et al. (2020). CodeBERT: a pre-trained model for programming and natural languages. In Findings EMNLP 2020 (pp. 1536-1547). ACL. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[31] Guo, D., et al. (2022). UniXcoder: unified cross-modal pre-training for code representation. In Proceedings ACL 2022 (pp. 7212-7225). ACL. https://doi.org/10.18653/v1/2022.acl-long.499

[32] Reimers, N., & Gurevych, I. (2019). Sentence-BERT: sentence embeddings using Siamese BERT-networks. In Proceedings EMNLP 2019 (pp. 3982-3992). ACL. https://doi.org/10.18653/v1/D19-1410

[33] Kochhar, P.S., Thung, F., & Lo, D. (2016). Automatic fine-grained issue report reclassification. In Proceedings ICSME 2016 (pp. 126-136). IEEE. https://doi.org/10.1109/ICSME.2016.47

[34] Chen, B., Cong, B., Sunder, M., Vu, L., Gao, S., & Bhatia, A. (2023). RepairLLaMA: efficient representations and fine-tuned adapters for program repair. arXiv preprint arXiv:2312.15698.

[35] Xia, C.S., Paltenghi, M., Tian, J., Pradel, M., & Zhang, L. (2023). Universal fuzzing via large language models. arXiv preprint arXiv:2308.04748.

[36] Olausson, T.X., Inala, J.P., Wang, C., Gao, J., & Solar-Lezama, A. (2023). Is self-repair a silver bullet for code generation? In Proceedings ICLR 2024. ICLR.

[37] Paul, S., & Swatman, P.M.C. (1997). An ontological approach to specification of quality factors for software products. In Proceedings DEXA 1997 (pp. 585-594). Springer.

[38] Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., & Ma, S. (2020). CodeBLEU: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297.

[39] Devlin, J., Chang, M.W., Lee, K., & Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In NAACL-HLT 2019 (pp. 4171-4186). ACL.

[40] Vaswani, A., et al. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 30, 5998-6008.

[41] Brown, T.B., et al. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems, 33, 1877-1901.

[42] Liu, P., et al. (2019). RoBERTa: a robustly optimized BERT pretraining approach. arXiv preprint arXiv:1907.11692.

[43] Hu, E.J., et al. (2022). LoRA: low-rank adaptation of large language models. In Proceedings ICLR 2022. ICLR.

[44] Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: efficient finetuning of quantized LLMs. Advances in Neural Information Processing Systems, 36, 10088-10115.

[45] Chen, L., et al. (2023). CodeT5+: open code large language models for code understanding and generation. In Proceedings EMNLP 2023 (pp. 1069-1088). ACL. https://doi.org/10.18653/v1/2023.emnlp-main.68

[46] Roziere, B., et al. (2023). Code Llama: open foundation models for code. arXiv preprint arXiv:2308.12950.

[47] Wei, J., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. Advances in Neural Information Processing Systems, 35, 24824-24837.

[48] Papineni, K., Roukos, S., Ward, T., & Zhu, W.J. (2002). BLEU: a method for automatic evaluation of machine translation. In Proceedings ACL 2002 (pp. 311-318). ACL. https://doi.org/10.3115/1073083.1073135

[49] Croft, R., Newlands, D., Chen, Z., & Babar, M.A. (2022). SoK: systematic literature review of automatic software vulnerability detection using deep learning (extended version). arXiv preprint arXiv:2205.01326.

[50] Li, Y., Wang, S., & Nguyen, T.N. (2021). SYSEVR: a framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing, 19(4), 2244-2258. https://doi.org/10.1109/TDSC.2021.3051525

[51] Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2022). Deep learning based vulnerability detection: are we there yet? IEEE Transactions on Software Engineering, 48(9), 3280-3296. https://doi.org/10.1109/TSE.2021.3087402

[52] Peng, H., et al. (2021). CAREER: car evaluation with retrieval-enhanced regression. In Proceedings AAAI 2021.

[53] Kang, S., & Yoo, S. (2023). Explainable automated debugging via large language model-driven scientific debugging. arXiv preprint arXiv:2304.02195.

[54] Jiang, N., Liu, K., Lutellier, T., & Tan, L. (2023). Impact of code language models on automated program repair. In Proceedings ICSE 2023 (pp. 1430-1442). IEEE. https://doi.org/10.1109/ICSE48619.2023.00122

[55] Wei, Y., Xia, C.S., & Zhang, L. (2023). Copiloting the copilots: fusing large language models with completion engines for automated program repair. In Proceedings FSE 2023 (pp. 172-184). ACM. https://doi.org/10.1145/3611643.3616271

[56] Sobania, D., Briesch, M., Hanna, C., & Petke, J. (2023). An analysis of the automatic bug fixing performance of ChatGPT. In Proceedings APR 2023 (pp. 23-29). IEEE. https://doi.org/10.1109/APR59189.2023.00012

[57] Ziems, N., Yu, W., Zhang, Z., & Jiang, M. (2022). Can large language models effectively leverage structural information for graph learning: when and why. arXiv preprint arXiv:2306.03179.

[58] Wan, Y., Zhao, Z., Yang, W., Xu, G., Akbar, H., & Yu, P.S. (2018). Improving automatic source code summarization via deep reinforcement learning. In Proceedings ASE 2018 (pp. 397-407). ACM. https://doi.org/10.1145/3238147.3238206

[59] Ahmad, W.U., et al. (2021). Unified pre-training for program understanding and generation. In NAACL-HLT 2021 (pp. 2655-2668). ACL. https://doi.org/10.18653/v1/2021.naacl-main.211

[60] Yin, P., & Neubig, G. (2017). A syntactic neural model for general-purpose code generation. In Proceedings ACL 2017 (pp. 440-450). ACL. https://doi.org/10.18653/v1/P17-1041

[61] Mou, L., et al. (2016). Convolutional neural networks over tree structures for programming language processing. In Proceedings AAAI 2016 (pp. 1287-1293). AAAI Press.

[62] Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). code2vec: learning distributed representations of code. In Proceedings POPL 2019 (pp. 1-29). ACM. https://doi.org/10.1145/3290353

[63] Alon, U., Brody, S., Levy, O., & Yahav, E. (2019). code2seq: generating sequences from structured representations of code. In Proceedings ICLR 2019. ICLR.

[64] Allamanis, M., Barr, E.T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), 1-37. https://doi.org/10.1145/3212695

[65] Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. (2016). Summarizing source code using a neural attention model. In Proceedings ACL 2016 (pp. 2073-2083). ACL. https://doi.org/10.18653/v1/P16-1195

[66] Just, R., Jalali, D., & Ernst, M.D. (2014). Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In Proceedings ISSTA 2014 (pp. 437-440). ACM. https://doi.org/10.1145/2610384.2628055

[67] Martinez, M., & Monperrus, M. (2019). ASTOR: a program repair library for Java. In Proceedings ISSTA 2016 (pp. 441-444). ACM. https://doi.org/10.1145/2931037.2948705

[68] Yuan, Y., & Banzhaf, W. (2020). ARJA: automated repair of Java programs via multi-objective genetic programming. IEEE Transactions on Software Engineering, 46(10), 1040-1067. https://doi.org/10.1109/TSE.2018.2874612

[69] Ghanbari, A., & Baudry, B. (2019). PraPR: practical program repair via bytecode mutation. In Proceedings ASE 2019 (pp. 499-510). IEEE. https://doi.org/10.1109/ASE.2019.00054

[70] Saha, R.K., Lyu, Y., Lam, W., Yoshida, H., & Prasad, M.R. (2019). Elixir: effective object-oriented program repair. In Proceedings ASE 2017 (pp. 648-659). IEEE. https://doi.org/10.1109/ASE.2017.8115675

[71] Smith, E.K., Barr, E.T., Le Goues, C., & Brun, Y. (2015). Is the cure worse than the disease? Overfitting in automated program repair. In Proceedings FSE 2015 (pp. 532-543). ACM. https://doi.org/10.1145/2786805.2786825

[72] Tian, H., et al. (2022). Is this change the answer to that problem? Correlating descriptions of bug and code changes for evaluating patch correctness. In Proceedings ASE 2022 (pp. 1-13). ACM. https://doi.org/10.1145/3551349.3556963

[73] Hua, J., Zhang, M., Wang, K., & Khurshid, S. (2018). Towards practical program repair with on-demand candidate generation. In Proceedings ICSE 2018 (pp. 12-23). ACM. https://doi.org/10.1145/3180155.3180245

[74] Weimer, W., Nguyen, T.V., Le Goues, C., & Forrest, S. (2009). Automatically finding patches using genetic programming. In Proceedings ICSE 2009 (pp. 364-374). IEEE. https://doi.org/10.1109/ICSE.2009.5070536

[75] Ma, Y., Liang, Y., Chen, Z., & Vaidya, J. (2022). Metamorphic testing and mutation analysis: exploring the frontier of software testing. In Proceedings ICTAC 2022. Springer.

[76] CVEfixes. (2021). CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. GitHub. https://github.com/secureIT-project/CVEfixes

[77] NVD. (2024). National Vulnerability Database API. NIST. https://nvd.nist.gov/developers