HEJ-Robust: A Robustness Benchmark for LLM-based Automated Program Repair

Anonymous Author(s)

Abstract

Recent Large Language Models (LLMs) have shown strong performance on automated program repair across standard benchmarks. However, these benchmarks evaluate models on a single canonical form of buggy code and do not reflect the syntactic variations commonly observed in real-world software, leaving robustness largely unexamined. In this work, we construct HEJ-Robust, a robustness benchmark built from HumanEval-Java-Bug using eight semantic-preserving code transformations, resulting in 1,350 transformed instances. We evaluate five fine-tuned LLMs on this benchmark and show that model performance drops by over 50% under several transformations, indicating that current LLM-based repair models lack robustness to minor syntactic variations.

Keywords

Large Language Models, Automated Program Repair, Benchmark, Robustness Testing

ACM Reference Format:

Anonymous Author(s). 2026. HEJ-Robust: A Robustness Benchmark for LLM-based Automated Program Repair. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Automated program repair (APR) aims to automatically generate patches that fix buggy programs. Early APR approaches primarily followed the generate-and-validate paradigm, where candidate patches are synthesized using predefined or learned repair operators and validated against test suites. Representative systems include GenProg [21], PAR [19], and systematic mutation-based repair techniques [20, 29]. While these approaches have demonstrated effectiveness on specific bug classes, they often suffer from scalability limitations and test-suite overfitting.

Recent advances in deep learning have significantly reshaped APR research by formulating program repair as a code translation problem, where buggy code is translated into its fixed version [17, 49]. Pre-trained LLMs, such as PLBART [1], and CodeT5 [37], have shown strong repair capability when fine-tuned on bug-fix data [6, 16, 46]. More recent studies further explore instruction-tuned and agent-based LLMs for automated program repair [4, 11, 47]. To evaluate these approaches, existing benchmarks commonly rely on defect4j [18] or HumanEval-Java-Bug [16], which assume a fixed syntactic representation of buggy programs.

Existing APR benchmarks evaluate repair accuracy on a single canonical buggy program, ignoring syntactic diversity among semantically equivalent code. Prior studies show neural code models are sensitive to semantics-preserving transformations [30, 36]. While robustness testing via transformations, fuzzing, and adversarial examples has been studied in other SE tasks [28, 43], robustness evaluation for LLM-based APR remains largely unexplored, with no standardized benchmark available.

We address this gap by introducing a transformation-based robustness benchmark for automated program repair. Constructed by applying eight semantic-preserving transformations to HumanEval-Bug [16], our benchmark enables controlled evaluation of repair consistency. We use it to assess the robustness of five fine-tuned LLM repair models against code perturbations.

The contributions of this paper are as follows:

(1) We introduce the first transformation-based robustness benchmark tailored for automated program repair.
(2) We provide a systematic evaluation of LLM-based repair models under semantics-preserving transformations.
(3) We release the benchmark to facilitate future research on robust and reliable automated program repair.

Our Code, dataset, and Artifacts are publicly available [1]

2 Related Work

Automated program repair (APR) has been extensively studied over the past two decades. Early work primarily follows the generate-and-validate paradigm, where candidate patches are generated and validated against test suites [15, 21, 29, 39, 42, 44]. While effective on curated benchmarks such as Defects4J [18] and QuixBugs [23], these approaches suffer from overfitting and scalability issues [12, 22, 40, 41].

More recently, deep learning-based APR approaches reformulate bug fixing as a neural machine translation problem, translating buggy code into fixed code [2, 7, 10, 14, 17, 25, 35, 49]. Pre-trained LLMs further improve repair performance by leveraging large-scale code corpora before fine-tuning on repair data [1, 6, 8, 13, 24, 26, 37, 46]. Most of these approaches evaluate on bug-fix pairs (BFPs) [8, 35], which largely consist of abstract or canonicalized code. More recent benchmarks derived from HumanEval [9] enable functional validation using test cases [16]. Complementary studies explore LLM-based repair in competitive programming and agent-based settings [4, 11, 47].

Parallel to APR research, robustness testing of neural models for code has gained attention. Prior work demonstrates that neural code models are vulnerable to small, semantics-preserving transformations [3, 33, 48]. Transformation-based testing, fuzzing, and

[1] https://github.com/anonprox/hej-robust

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1: Fine-tuned models against different semantic-preserving code transformations on HumanEval-Java.

(a) Local Variable Renaming (100 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	14.53	6.54	54.99↓	82.11	81.91
plbart_large	21.88	9.91	54.71↓	82.75	82.15
codet5_small	19.35	8.26	57.31↓	82.17	81.43
codet5_base	24.81	12.28	50.5↓	82.02	81.58
codet5_large	23.66	11.5	51.39↓	80.74	80.92

(b) Method Renaming (149 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	19.46	18.13	6.83↓	82.96	82.78
plbart_large	23.98	22.8	4.92↓	83.34	83.04
codet5_small	21.16	19.46	8.03↓	83.1	82.96
codet5_base	25.87	24.37	5.8↓	83.06	82.71
codet5_large	24.75	23.98	3.11↓	81.73	81.69

(c) Parameter Renaming (162 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	17.86	18.69	4.65↑	83.63	83.84
plbart_large	22.6	23.33	3.23↑	84.11	83.86
codet5_small	20.3	19.1	5.91↓	83.92	84.02
codet5_base	24.77	24.41	1.45↓	83.86	83.94
codet5_large	24.41	23.7	2.91↓	82.67	82.55

(d) Boolean Exchange (7 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	12.5	22.22	77.76↑	86.03	85.8
plbart_large	22.22	30.0	35.01↑	86.14	85.69
codet5_small	22.22	22.22	0%	83.68	83.3
codet5_base	22.22	12.5	43.74↓	86.02	85.47
codet5_large	12.5	12.5	0%	85.26	84.67

(e) Loop Exchange (142 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	19.32	18.39	4.81↓	84.66	84.76
plbart_large	25.26	23.66	6.33↓	84.77	85.5
codet5_small	21.55	17.92	16.84↓	84.08	84.61
codet5_base	26.04	23.66	9.14↓	84.58	85.12
codet5_large	28.28	26.8	5.23↓	83.29	84.36

(f) Reorder Condition (603 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	16.88	15.69	7.05↓	83.83	85.64
plbart_large	21.41	18.48	13.69↓	84.17	86.1
codet5_small	19.7	17.92	9.04↓	84.01	85.87
codet5_base	23.25	20.99	9.72↓	83.94	85.75
codet5_large	23.45	21.62	7.8↓	82.56	85.04

(g) Insert Log Statement (173 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	17.22	16.43	4.59↓	83.64	83.6
plbart_large	22.07	22.42	1.59↑	84.1	83.85
codet5_small	19.53	18.4	5.79↓	83.86	83.69
codet5_base	24.45	22.07	9.73↓	83.8	83.57
codet5_large	24.78	24.45	1.33↓	82.61	82.91

(h) Insert Try catch (114 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	16.91	13.74	18.75↓	83.86	83.65
plbart_large	21.53	19.29	10.4↓	84.17	83.94
codet5_small	19.29	11.02	42.87↓	84.52	84.49
codet5_base	25.17	18.12	28.01↓	84.29	84.3
codet5_large	26.14	18.12	30.68↓	83.16	83.52

adversarial example generation have been applied to code models [28, 31, 38, 45], with later work emphasizing natural and context-aware transformations [43]. While robustness has been studied for tasks such as code summarization and code representation learning, it remains largely unexplored for automated program repair. In particular, existing APR benchmarks do not systematically evaluate the robustness of repair models under semantics-preserving code transformations.

In this work, we bridge this gap by focusing on robustness benchmarking for LLM-based program repair rather than proposing a new repair technique.

3 Robustness Benchmark Design

3.1 Base Dataset

We adopt the HumanEval-Java-Bug dataset introduced by Jiang et al. [16], which is derived from HumanEval [9]. The dataset contains 164 Java programs with manually injected bugs and annotated buggy-line locations. Each instance is accompanied by executable

test cases and human-written patches. We select this dataset because it is manually curated, recent, and less likely to suffer from data leakage issues common in earlier APR benchmarks.

3.2 Semantic-Preserving Code Transformations

We apply eight semantic-preserving code transformations that reflect common syntactic variations observed in real-world software. Prior work has shown that such transformations are effective for evaluating the robustness of neural code models. Our goal is not to improve code quality, but to assess robustness under benign syntactic changes.

The eight transformations are: (1) local variable renaming, (2) method renaming, (3) parameter renaming, (4) log statement insertion, (5) try–catch insertion, (6) boolean exchange, (7) loop exchange, and (8) condition reordering. All transformations preserve program semantics and do not change test outcomes.

Renaming transformations (1–3). For identifier renaming, we adopt the naturalness-aware substitution strategy proposed by Yang et al. [43]. Unlike prior approaches that use random strings or fixed patterns [28, 31, 38, 45], this method generates context-aware and developer-natural identifiers, ensuring that performance degradation reflects robustness issues rather than unnatural code artifacts.

We use masked language prediction with CodeBERT and GraphCodeBERT to generate candidate identifiers and select substitutions based on cosine similarity in embedding space. Java code is parsed using tree-sitter [5] to ensure consistent replacement across all occurrences. To control transformation strength, only one identifier is renamed per program.

Structural and syntactic transformations (4–8). The remaining transformations are implemented using JavaTransformer [32], which applies AST-based modifications via JavaParser. Logging statements are inserted at method entry points, and try–catch blocks are added at syntactically valid locations. Boolean exchange inverts boolean initializations while preserving semantics. Loop exchange converts `for` loops to equivalent `while` loops and vice versa. Condition reordering swaps operands in equality and inequality expressions. Transformations are applied only when syntactically valid.

All transformations preserve semantics and syntax, yielding 1,350 instances.

3.3 Benchmark Construction and Task Formulation

After applying transformations, the locations of buggy lines may change. We manually re-annotate the buggy-line locations for all transformed programs. Combined with the original human-written patches and test cases, this yields a fully executable benchmark suitable for robustness evaluation. Model outputs are evaluated using both code-similarity metrics, such as CodeBLEU [27, 34], and functional correctness via test-based metrics (e.g., pass@10) provided by HumanEval-Java-Bug [16].

4 Experimental Setup

To evaluate the proposed benchmark, we consider five LLMs: two PLBART variants and three CodeT5 variants, all fine-tuned and released by Jiang et al. [16]. We directly evaluate these models without further modification. We use CodeBLEU as a similarity-based metric, while Pass@10 serves as the primary indicator of functional correctness. Results are compared to quantify robustness degradation under semantic-preserving transformations, as reported in Table 1. More results with pre-trained models are available in our shared repository.

5 Results

The evaluation results are summarized in Table 1, which reports the performance of five fine-tuned models across eight transformed datasets. Each transformation is presented in a separate subtable, showing Pass@10 and CodeBLEU scores for both the original and transformed datasets. We also report the relative change from the original to the transformed dataset, indicated by ↑ for improvements and ↓ for degradations.

Across all eight transformations, we observe consistent drops in Pass@10 for all models, with the largest degradation occurring under the Local Variable Renaming transformation, where performance decreases by 50.5% to 57.31%. Notably, robustness does not correlate with model size: larger models often degrade more than their smaller counterparts (e.g., `CodeT5_large` vs. `CodeT5_base`, and `PLBART_large` vs. `PLBART_base`) across multiple transformations. Results for Boolean Exchange are unstable due to the small number of affected samples (7 instances).

In contrast, CodeBLEU scores remain largely stable across transformations, with only minor increases or decreases. This discrepancy suggests that CodeBLEU may not fully capture functional robustness, which we leave for future investigation.

6 Conclusion

We present **HEJ-Robust**, a robustness benchmark of 1,350 bug instances constructed from HumanEval-Java-Bug using eight semantic-preserving transformations. Evaluating five fine-tuned LLMs, we show that even minor syntactic variations cause consistent drops in Pass@10, revealing substantial robustness gaps in current repair models. Future work will study richer transformations, larger contexts, and improved robustness-aware training and evaluation metrics.

References

[1] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[2] B. Berabi, J. He, V. Raychev, and M. Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, pages 780–791. PMLR, 2021.

[3] P. Bielik and M. Vechev. Adversarial robustness for code. In *International Conference on Machine Learning*, pages 896–907. PMLR, 2020.

[4] I. Bouzenia, P. Devanbu, and M. Pradel. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*, 2024.

[5] M. Brunsfeld and contributors. tree-sitter. https://github.com/tree-sitter/tree-sitter, 2024. Accessed: 2024-05-23.

[6] S. Chakraborty, T. Ahmed, Y. Ding, P. Devanbu, and B. Ray. Natgen: Generative pre-training by" naturalizing" source code. *arXiv preprint arXiv:2206.07585*, 2022.

[7] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 2020.

[8] S. Chakraborty and B. Ray. On multi-modal learning of editing source code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 443–455. IEEE, 2021.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[10] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959, 2019.

[11] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023.

[12] X. Gao, S. Mechtaev, and A. Roychoudhury. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 8–18, 2019.

[13] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[14] R. Gupta, S. Pal, A. Kanade, and S. Shevade. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI conference on artificial intelligence*, 2017.

[15] J. Hua, M. Zhang, K. Wang, and S. Khurshid. Sketchfix: A tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 888–891, 2018.

[16] N. Jiang, K. Liu, T. Lutellier, and L. Tan. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442. IEEE, 2023.

[17] N. Jiang, T. Lutellier, and L. Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[18] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[19] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE, 2013.

[20] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[21] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[22] J. Liang, R. Ji, J. Jiang, S. Zhou, Y. Lou, Y. Xiong, and G. Huang. Interactive patch filtering as debugging aid. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 239–250. IEEE, 2021.

[23] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56, 2017.

[24] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[25] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

[26] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo. Spt-code: sequence-to-sequence pre-training for learning source code representations. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2006–2018, 2022.

[27] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[28] M. V. Pour, Z. Li, L. Ma, and H. Hemmati. A search-based testing framework for deep neural networks of source code embedding. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 36–46. IEEE, 2021.

[29] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[30] F. Rabbi, Z. Ding, and J. Yang. A multi-language perspective on the robustness of llm code generation. 2025.

[31] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552, 2021.

[32] M. R. I. Rabin, K. Wang, and M. A. Alipour. Testing neural program analyzers. *arXiv preprint arXiv:1908.10711*, 2019.

[33] G. Ramakrishnan, J. Henkel, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps. Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043*, 2020.

[34] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[35] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

[36] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia, et al. Recode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13818–13843, 2023.

[37] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[38] M. Wei, Y. Huang, J. Yang, J. Wang, and S. Wang. Cocofuzzing: Testing neural code models with coverage-guided fuzzing. *arXiv preprint arXiv:2106.09242*, 2021.

[39] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1–11. IEEE, 2018.

[40] B. Yang and J. Yang. Exploring the differences between plausible and correct patches at fine-grained level. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pages 1–8. IEEE, 2020.

[41] J. Yang, L. Tan, J. Peyton, and K. A. Duer. Towards better utilizing static application security testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 51–60. IEEE, 2019.

[42] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 831–841, 2017.

[43] Z. Yang, J. Shi, J. He, and D. Lo. Natural attack for pre-trained models of code. *arXiv preprint arXiv:2201.08698*, 2022.

[44] H. Ye, M. Martinez, and M. Monperrus. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1506–1518, 2022.

[45] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1169–1176, 2020.

[46] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric. Coditt5: Pretraining for source code and natural language editing. *arXiv preprint arXiv:2208.05446*, 2022.

[47] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen. A critical review of large language model on software engineering: An example from chatgpt and automated program repair. *arXiv preprint arXiv:2310.08879*, 2023.

[48] W. E. Zhang, Q. Z. Sheng, A. Alhazmi, and C. Li. Adversarial attacks on deep-learning models in natural language processing: A survey. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 11(3):1–41, 2020.

[49] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 341–353, 2021.