Evaluating Pre-Trained Models for Multi-Language Vulnerability Patching

Zanis Ali Khan, Aayush Garg, Yuejun Guo, and Qiang Tang

Luxembourg Institute of Science and Technology (LIST), Luxembourg

Abstract. Software vulnerabilities pose critical security risks, demanding prompt and effective mitigation strategies. While advancements in Automated Program Repair (APR) have primarily targeted general software bugs, the domain of vulnerability patching, which is a security-critical subset of APR, remains underexplored. This paper investigates the potential of pre-trained language models, CodeBERT and CodeT5, for automated vulnerability patching across diverse datasets and five programming languages. We evaluate these models on their accuracy, computational efficiency, and the impact of patch length on performance. Our findings reveal promising accuracy levels, particularly for CodeT5 on datasets with complex vulnerability patterns, while CodeBERT demonstrates strengths in handling fragmented or context-limited datasets. CodeT5 further showcases superior efficiency, making it well-suited for large-scale applications. However, both models face challenges in maintaining performance as patch length increases, highlighting the complexity of addressing extended sequences in vulnerability-focused program repair. This study benchmarks model performance, highlights key limitations, and offers insights to improve automated vulnerability patching for practical security applications.

Keywords: vulnerability patching · code patching · automated program repair

1 Introduction

Software vulnerabilities pose a persistent threat to modern software systems, exposing them to potential exploitation by attackers. These vulnerabilities, ranging from memory management issues to injection flaws, can lead to unauthorized access, data breaches, and service disruptions [1]. Addressing these vulnerabilities promptly is critical to maintaining the security and reliability of software systems [2]. However, the manual process of identifying and remediating these issues is labor-intensive, error-prone, and unable to keep up with the increasing complexity and scale of modern software ecosystems[3].

Automated Program Repair (APR) has become a promising approach to address this issue, utilizing computational methods to generate bug patches autonomously [4]. While APR has seen significant advancements in fixing general software bugs, vulnerability-focused program repair—a domain targeting

the unique challenges of security vulnerabilities—remains underexplored. Unlike functional bugs, vulnerabilities require patches that address not only the immediate issue but also ensure that the fix is secure, robust, and resistant to future exploitation [5]. This added complexity makes vulnerability patching a particularly demanding subset of APR.

Existing approaches in vulnerability-focused APR rely on either static analysis tools or traditional machine learning models trained on specific vulnerability patterns. While these methods have shown promise in detecting vulnerabilities, their ability to generate meaningful and effective patches is limited. Static analysis tools, for example, excel at identifying flaws but often fail to produce usable fixes [6]. Similarly, conventional machine learning models are constrained by their reliance on limited datasets [7], which hampers their generalizability and effectiveness across diverse programming languages and vulnerability types [8].

Recent progress in deep learning, especially with the advent of pre-trained language models for code like CodeBERT [9] and CodeT5 [10], offer a new avenue for automated vulnerability patching. These models leverage large-scale code corpora to learn syntactic and semantic representations of programming languages [11], enabling them to perform complex tasks like code generation, summarization, and translation [12]. Their capacity to identify patterns from extensive datasets makes them well-equipped to address the complexities of vulnerability-focused program repair. However, applying these models to vulnerability patching is far from straightforward [13]. Programming languages differ significantly in syntax, semantics, and vulnerability patterns, and most pre-trained models are designed for monolingual or narrowly scoped tasks. Consequently, evaluating their effectiveness across multiple programming languages is critical yet underexplored.

In this paper, we address these gaps by systematically evaluating the performance of pre-trained language models in vulnerability-focused program repair across multiple programming languages. Specifically, we investigate the capabilities of CodeBERT and CodeT5 in generating patches for known vulnerabilities across 9 datasets spanning multiple programming languages. Our evaluation examines the effectiveness of these models in terms of their ability to generate accurate patches and their computational efficiency in handling large-scale vulnerability patching tasks. Additionally, we examined the impact of patch length on the accuracy of *CodeBERT* and *CodeT5* by using *CodeBLEU* and *CrystalBLEU* scores across nine datasets.

Our results reveal that while both models demonstrate notable strengths in generating vulnerability patches, they also exhibit distinct limitations. CodeT5 generally outperforms CodeBERT in accuracy, particularly on datasets with complex or diverse vulnerability patterns. However, both models face challenges in handling datasets with fragmented contexts or sparse data, which hinder their ability to generate accurate fixes. Additionally, our analysis reveals that increasing patch lengths significantly influence the accuracy of both models, with their performance declining to varying degrees depending on dataset characteristics and model design.

Hence, our contributions in this paper are threefold:

- We provide an evaluation of CodeBERT and CodeT5 for vulnerability-focused program repair, covering a diverse set of 9 datasets across multiple programming languages.
- We establish benchmarks for model performance in generating vulnerability patches, serving as a foundation for evaluating pre-trained models in dataset-driven vulnerability patching scenarios.
- We identify the challenge posed by increasing patch lengths on model performance, offering insights into their implications for automated vulnerability patching.

2 Background and Related Work

2.1 Software Vulnerability and Mitigation

Software vulnerabilities are security flaws or weaknesses in code that can be exploited by attackers [14]. For example, a buffer overflow vulnerability occurs when a program attempts to store more data in a buffer than its allocated capacity, causing the excess to overwrite adjacent memory locations and potentially enabling attackers to execute malicious code [15]. The increasing sophistication of such vulnerabilities presents significant challenges in implementing effective mitigation measures.

While much of the research has focused on vulnerability detection, fewer efforts have addressed patch generation. In detection, traditional static analysis tools have been widely used, though their rule-based nature often limits their ability to capture complex patterns [16]. AI-based approaches have gained prominence for their ability to analyze large code corpora and identify intricate vulnerabilities. Notable models include *CodeBERT*[17] and *GraphCodeBERT*[18], which have shown promise in source code analysis, including applications in vulnerability detection and analysis [19,20]. Moreover, large language models (LLMs) such as OpenAI's GPT-4, Meta AI's Llama2, and Mistral AI's Mistral have demonstrated effective adaptation to vulnerability detection tasks [21].

On the other hand, generating effective patches remains a formidable challenge. Most automated patch generation research focuses on repairing general buggy code rather than addressing vulnerabilities specifically. The following sections will review approaches in this broader area.

2.2 Traditional Approaches to Code Repair

Traditional approaches to automated code repair are broadly categorized into heuristic and constraint-based methods [22]. Heuristic methods explore a search space of possible patches, aiming to identify one that satisfies all test cases. To make this process manageable, techniques like transformation schemas are used to generate candidates efficiently [23]. Approaches such as GenProg [24] and

PAR [25] utilize genetic programming to refine the search, while others rely on random or deterministic search strategies to improve efficiency.

Constraint-based methods, on the other hand, leverage symbolic execution [26] to guide patch generation. By abstracting program inputs into symbolic values, these techniques explore multiple input paths simultaneously. Tools like SemFix [27] and Angelix [28] use symbolic execution to infer repair constraints, while methods such as Nopol [29] focus on specific scenarios, like fixing conditional statements.

2.3 ML-Based Code Repair

Machine learning has become a pivotal approach for automating code repair, leveraging models to generate patches for vulnerabilities and bugs. Early methods primarily relied on neural machine translation (NMT) models with encoder-decoder architectures to transform buggy code into patched counterparts. Tools like SequenceR [30] and CODIT [31] introduced attention mechanisms to improve focus on relevant input regions during decoding.

Recent advancements have shifted toward transformer-based architectures, which excel at capturing long-range dependencies and nuanced contextual information. Enabled by attention mechanisms, these models dynamically prioritize relevant code segments, making them highly effective for program repair. Ding *et al.* [32] demonstrated the potential of transformers for program repair, paving the way for widespread adoption.

Building on transformers, large language models (LLMs) like CodeBERT [17] and TFix [33] have become dominant in vulnerability patching due to their pretraining on extensive code corpora. These models have been fine-tuned for targeted repairs, showing promising results across diverse datasets.

However, vulnerability patching remains uniquely challenging. Unlike general bug repair, it requires highly contextualized and security-specific modifications, demanding models to generalize effectively across complex scenarios. Addressing these challenges necessitates fine-tuning LLMs and developing techniques to enhance their adaptability to diverse datasets and nuanced security requirements.

3 Research Questions

The use of pre-trained language models in automated program repair (APR) has shown promising results, particularly for general bugs. However, their effectiveness in vulnerability-focused program repair, which demands highly accurate and context-aware patches, remains underexplored [34]. Unlike general-purpose bug fixes, vulnerability patches must address specific security threats while aligning with language-specific patterns, coding styles, and contextual nuances.

Prominent models like *CodeBERT* and *CodeT5* have excelled in various code-related tasks [35,12]. Yet, their ability to adapt to vulnerability datasets, particularly those spanning diverse programming contexts, is uncertain [36]. Understanding how well these models generalize across such datasets is critical for advancing their real-world applicability. Hence, we ask:

RQ1 Accuracy Across Datasets. How effectively do CodeBERT and CodeT5 generate accurate patches for known vulnerabilities across diverse datasets?

Beyond patch accuracy, scalability is crucial for applying pre-trained models in vulnerability repair. Modern software systems generate vast amounts of code, demanding solutions that are both accurate and efficient [37]. Optimized processes have demonstrated effective scaling across larger systems, making computational efficiency particularly important for integration into CI/CD pipelines or large-scale vulnerability assessments [38]. Moreover, scalability ensures that these models can handle the dynamic and evolving nature of software development, where new vulnerabilities are discovered continuously. Evaluating inference time and resource usage is essential to assess their practicality in real-world scenarios. Such assessments provide insights into the trade-offs between model accuracy and efficiency, which are critical for deploying these models at scale. Thus, we ask:

RQ2 Efficiency Trade-offs. What are the computational trade-offs, in terms of inference efficiency, when using CodeBERT and CodeT5 for large-scale vulnerability patching tasks?

Also, research indicates that as the length of generated text increases, LLMs often experience declines in coherence and accuracy. The existing research has shown that these models often struggle to maintain coherent event sequences in longer texts [39] and suffer significant performance degradation in tasks requiring with extended contexts [40]. Therefore, it is crucial to examine whether *CodeBERT* and *CodeT5* maintain their effectiveness in generating accurate patches as patch length increases. Understanding the relationship between patch length (for LLM-generated patches) and model performance will provide insights into their robustness across varying output lengths. Hence, we ask:

RQ3 Impact of Patch Length. How does the length of generated patches affect the accuracy of CodeBERT and CodeT5 in vulnerability-focused program repair?

4 Methodology

The methodology for this study is structured into several stages, as shown in Figure 1, encompassing dataset collection, preprocessing, training, and evaluation. Below, we describe each stage in detail.

4.1 Dataset Preparation

For this study, we collected nine publicly available datasets containing code samples with known vulnerabilities and their corresponding patches. These datasets span multiple programming languages, including Go, PHP, Java, C and C++,

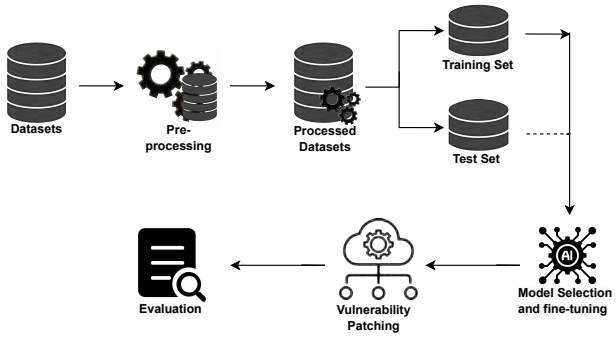

Fig. 1: Overview of our Methodology.

ensuring diverse code structures and vulnerability patterns. The inclusion of diverse datasets allowed us to evaluate the models' ability to generalize across varied programming contexts. Details about these datasets, including their references are provided in Section 5.1, offering a comprehensive overview of their sources. This diversity in datasets not only enhances the robustness of our evaluation but also reflects real-world scenarios where vulnerabilities span multiple languages and coding paradigms.

4.2 Preprocessing

We preprocessed the raw datasets to standardize their structure and improve compatibility with the models. Given the inherent noise in real-world vulnerability datasets [7,41], our preprocessing aimed to mitigate the effects of noisy or inconsistent data. Accurate preprocessing is crucial for minimizing noise and improving data quality, as emphasized in studies addressing identification challenges in noisy datasets [42,43]. By ensuring uniformity and consistency in the datasets, we aimed to create a robust foundation for reliable model training and accurate performance assessment. The following steps performed were critical in reducing noise and preparing the datasets for training and evaluation.

i. **Token Length Filtering.** Code sequences exceeding *512* tokens were truncated or excluded, as pre-trained models impose this token length limit.
ii. **Comment Removal.** Comments were removed from the code using language-specific regex patterns to focus solely on functional elements of the code.
iii. **Normalization.** Formatting inconsistencies, such as extra whitespace and irregular line breaks, were corrected to ensure uniformity across datasets.

4.3 Training and Testing Split

The preprocessed datasets were divided into *85%* training and *15%* testing subsets, a common practice in machine learning research to balance model training

and evaluation [44]. This split was chosen to maximize training data while preserving sufficient data for evaluation. The separation of training and testing data is crucial for assessing the generalization ability of the model on new, unseen instances. To avoid data leakage, we adhered to best practices for dataset splitting by eliminating any overlapping or duplicate instances between the training and testing sets, ensuring the reliability of experimental outcomes.

4.4 Model Selection and Fine-Tuning

We utilized two state-of-the-art pre-trained models, *CodeBERT*[9] and *CodeT5*[10], to evaluate their performance in vulnerability patching. CodeBERT, designed explicitly for programming tasks, was fine-tuned on the training datasets to learn patterns in vulnerable and patched code. This fine-tuning process allowed CodeBERT to adapt and specialize in recognizing specific vulnerabilities and their corresponding patches. Similarly, CodeT5, a model optimized for code generation and understanding, was fine-tuned to align with the vulnerability patching task. CodeT5's optimization also focused on enhancing its ability to deal with various code structures and programming languages. The fine-tuning process adjusted the models' weights based on the training data to optimize their ability to generate accurate patches.

4.5 Vulnerability Patching and Evaluation

During the evaluation phase, the testing datasets were used to assess the models. Each test instance, representing a buggy code snippet, was input into the trained models to generate a patched version. These datasets covered a broad spectrum of programming languages and vulnerability types, ensuring a diverse set of conditions for evaluating the models' effectiveness. The patches produced were then compared to the ground truth using the *CodeBLEU*[45], and CrystalBLEU [46] metrics, which evaluates the accuracy and quality of the generated patches. By utilizing these metrics, we ensure that our evaluation encompasses both the syntactic correctness and semantics of the patch. CodeBLEU was chosen as it extends the traditional BLEU metric by incorporating code-specific features, such as Abstract Syntax Trees (ASTs) and Semantic Data Flow [47]. This extension allows CodeBLEU to better capture the structural and logical consistency of code, making it particularly useful for vulnerability patching. While, CrystalBLEU incorporates a more nuanced approach by considering trivially shared n-grams and offering a refined evaluation metric for code generation tasks. This ensures that both the syntactic and semantic correctness of the generated patches are assessed, providing a comprehensive measure of model performance.Moreover, these metrics together offer a balanced perspective, accounting for both syntactic accuracy and functional correctness, which are essential for high-quality vulnerability repair.

5 Experimental Setup

All experiments in this study were performed on a High-Performance Computing (HPC) cluster featuring nodes equipped with 2.20GHz Intel Xeon Silver 4210 processors and NVIDIA Tesla V100-PCIE-32GB GPUs. Model training and evaluation utilized the PyTorch 2.0.1 framework with CUDA 12 compatibility.

5.1 Datasets

To address the research questions introduced in Section 6, we utilized publicly available datasets containing extensive collections of vulnerable source code and their fixed versions as ground truth. Specifically, we employed nine datasets, including Go and PHP [1], BigVul[2][48], MegaVul_Java, MegaVul_C_2023, and MegaVul_C_2024 [3] [49], SVEN[4][50], and also CodeXGlue_Small_Java and CodeXGlue_Medium_Java [5] [51]. These datasets include diverse programming languages such as C, C++, Java, Go, and PHP, providing a comprehensive foundation for evaluation. Before using the datasets, we applied preprocessing steps as mentioned in Section 4.2.

Table 1: Datasets

Dataset	I_{rows}	$R_{tok.}$	$R_{comm.}$	$R_{norm.}$	T_{rows}
Go	1,472	551	357	0	921
PHP	6,696	335	4,923	1	6,360
BigVul	10,900	3,756	0	0	7,144
MegaVul_Java	2,433	62	0	75	2,296
MegaVul_C_2023	17,975	3,147	0	302	14,526
MegaVul_C_2024	17,975	3,147	0	302	14,526
SVEN	803	104	0	4	695
CodeXGlue_Medium_Java	65,455	0	0	0	65,455
CodeXGlue_Small_Java	58,350	0	0	0	58,350

Table 1 reports on the size of our datasets, in terms of the number of rows (I_{rows}), rows affected by tokenization ($R_{tok.}$), rows affected by comment removal ($R_{comm.}$), rows affected by normalization ($R_{norm.}$), and the total number of rows remaining after pre-processing ($T_{rows.}$).

[1] Go and PHP–https://doi.org/10.5281/zenodo.13870382

[2] BigVul– https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset

[3] MegaVul_Java, MegaVul_C_2023, and MegaVul_C_2024–https://github.com/Icyrockton/MegaVul

[4] SVEN–https://github.com/eth-sri/sven

[5] CodeXGlue_Small_Java and CodeXGlue_Medium_Java– https://github.com/microsoft/CodeXGLUE

5.2 DL Models

For the vulnerability patching, we considered *CodeBERT*[9], and *CodeT5*[10] models, which are considered state-of-the-art models in deep learning-based code analysis and vulnerability detection. These models have been widely recognized for their robustness and versatility, making them a benchmark choice for tasks involving code repair and analysis. Their ability to integrate with modern software development workflows further emphasizes their practical applicability in real-world scenarios.

CodeBERT[9] is a pre-trained model aimed at bridging the gap between programming and natural languages, improving tasks like code completion, summarization, and vulnerability detection. Using the transformer architecture, it is trained on a diverse dataset of natural language pairs and source code, enabling it to learn both syntactic and semantic relationships. This dual understanding is crucial for accurate vulnerability identification and remediation. Moreover, its scalability makes it an attractive option for projects ranging from small-scale open-source contributions to large enterprise-level software systems.

CodeT5[10] is a model optimized for code generation and understanding, excelling in vulnerability detection and patching. Based on the T5 framework, it adapts to programming languages, translating between code and natural language. One of its distinguishing features is its ability to generate context-aware patches that address vulnerabilities while preserving the intent of the original code. Pre-trained on extensive programming data, it generates code, identifies vulnerabilities, and suggests patches with high accuracy. Its ability to handle multiple languages and perform well on benchmarks makes it a powerful tool for software security and code quality improvement. CodeT5 has demonstrated its effectiveness in preserving high-level semantics during decompilation tasks in advancing vulnerability detection and repair [52] frameworks.

5.3 Evaluation Metrics

We used *CrystalBLEU*[46] and *CodeBLEU*[45] as metrics to evaluate the accuracy of the aforementioned deep learning models. CrystalBLEU extends BLEU [53] by addressing the limitations of traditional n-gram matching, especially when applied to programming languages. Unlike BLEU, which is designed for natural language and fails to capture crucial code-specific syntax and semantics, CrystalBLEU incorporates a more nuanced approach by considering trivially shared n-grams and offering a refined evaluation metric for code generation tasks. Additionally, CodeBLEU improves on BLEU by integrating n-gram matching with abstract syntax tree (AST)-based code structures and semantic analysis through data flow, making it particularly suitable for evaluating the quality of code. Both CrystalBLEU and CodeBLEU provide more accurate and meaningful assessments of code generation models by considering both syntactic and semantic aspects of the generated code.

6 Results and Discussion

6.1 Accuracy Across Datasets (RQ1)

Table 2 displays the CodeBLEU, and CrystalBLEU scores of CodeBERT and CodeT5 across nine datasets used in our evaluation. Examining the performance of both models on these datasets reveals key insights into how pre-training data diversity and model architecture impact the models' effectiveness in vulnerability patching tasks. CodeT5 consistently outperforms CodeBERT across the majority of datasets, with notable improvements in datasets like BigVul, MegaVul_Java, and SVEN. While the differences are less pronounced on MegaVul_C_2023 and MegaVul_C_2024, CodeT5 still demonstrates a clear advantage over CodeBERT when evaluated using both CodeBLEU and CrystalBLEU accuracy scores. This result aligns with the fact that CodeT5 has been pre-trained on diverse data that spans a variety of programming languages and textual formats, enabling it to capture more generalized patterns and nuances in code. In particular, the BigVul and MegaVul_Java datasets—where CodeT5's performance excels—include a range of real-world vulnerabilities and coding structures that may benefit from CodeT5's wider pre-training coverage.

Table 2: Accuracy Scores

	CodeBLEU		CrystalBLEU	
Dataset	*CodeBERT*	*CodeT5*	*CodeBERT*	*CodeT5*
Go	0.7641	0.6499	0.6557	0.5264
PHP	0.7351	0.6924	0.4624	0.3727
BigVul	0.6254	0.8148	0.4141	0.6161
MegaVul_Java	0.2736	0.832	0.0407	0.7747
MegaVul_C_2023	0.8396	0.8549	0.7893	0.8131
MegaVul_C_2024	0.8395	0.8549	0.7893	0.8131
SVEN	0.2127	0.8297	0	0.8089
CodeXGlue_Medium_Java	0.8674	0.8667	0.8277	0.824
CodeXGlue_Small_Java	0.7763	0.7639	0.6482	0.6291

The Go and PHP datasets present exceptions, with CodeBERT achieving higher CodeBLEU scores. By analyzing these two datasets, we observed that they often contain incomplete functions or isolated snippets lacking full context. This could potentially lead to lower performance for CodeT5, as it relies on contextual understanding from diverse sources that might not align well with fragmented or incomplete code. Conversely, CodeBERT, which is also trained on a broad variety of programming languages, may still benefit from its fine-tuned focus on code structure, making it more adaptable to such fragments.

These findings suggest that CodeBERT's architecture might be inherently more robust when handling incomplete or context-limited code, a factor that

could contribute to its better performance on Go and PHP. Moreover, despite CodeBERT generally being outperformed by CodeT5, its comparable performance on CodeXGlue datasets further emphasizes its effectiveness in Java-related tasks. This outcome suggests that CodeBERT, though lacking the extensive pre-training diversity of CodeT5, can still achieve near-competitive results in certain domains, particularly for language-specific tasks. It is worth noting that the CrystalBLEU score for CodeBERT on the SVEN dataset (*0%*) contrasts sharply with the CodeBLEU score on the same dataset (*0.21%*), highlighting an interesting anomaly specific to SVEN. While CodeBLEU has proven effective on other datasets, this discrepancy suggests that certain characteristics of the SVEN dataset may influence how these metrics evaluate patch quality. A closer manual examination of the patches generated by CodeBERT for SVEN revealed that they were often highly buggy and significantly different from the ground truth. This observation underscores the need for further investigation to better understand the interplay between dataset characteristics and metric sensitivity, rather than drawing generalized conclusions about the performance of CodeBLEU or CrystalBLEU.

Our results highlight the benefits of model diversity in deep learning-based vulnerability patching. CodeT5's broad pre-training excels on datasets with complex vulnerabilities, while CodeBERT's focused design performs well on datasets with more traditional, syntactically constrained samples. These insights show that model choice should depend on dataset characteristics. CodeBERT's simpler architecture likely makes it less reliant on context, while CodeT5 handles diverse inputs more effectively. Thus, while CodeT5 is suited for complex, varied data, CodeBERT is valuable in environments with incomplete or non-standard code snippets.

6.2 Efficiency Trade-offs (RQ2)

Table 3 presents the average inference efficiency of CodeBERT and CodeT5 across nine datasets, measured in execution time per test instance (in seconds). A comparative analysis of the results highlights a noticeable performance advantage for CodeT5, which consistently achieves faster inference times across different datasets when compared to CodeBERT. This performance difference underscores the enhanced efficiency of CodeT5, especially significant given the growing need for scalable, real-time processing in software engineering tasks.

The datasets in this analysis vary substantially in test instance counts and domain focus, with smaller, dataset like SVEN containing 105 test instances, and larger test sets like $CodeXGlue\_Medium\_Java$ encompassing 9,819 instances. It is important to note that the column $TestInstances$ in Table 3 reflect only the number of instances present in the test datasets. This diversity in test instance quantity and domain specificity helps evaluate the inference efficiency of CodeBERT and CodeT5 across varied real-world conditions, from niche, targeted datasets to larger, general-purpose code datasets. Despite this variability, CodeT5 maintains lower execution times, revealing its robust ability to handle diverse dataset characteristics and volumes more effectively. For instance,

on $MegaVul\_C\_2024$, CodeT5 demonstrates a marked reduction in execution time (1.5246s per instance) compared to CodeBERT's 2.0295s, suggesting a consistent edge in managing complex vulnerability datasets. This efficiency gain is even more pronounced in the Java datasets (except $MegaVul\_Java$), where CodeT5's execution time is less than half that of CodeBERT, reflecting its adaptability and optimized performance for higher data throughput demanding tasks.

Table 3: Efficiency Comparison (*In Seconds*)

Dataset	$CodeBERT$	$CodeT5$	$TestInstances$
Go	2.4421	1.2477	139
PHP	1.4735	1.2787	954
BigVul	2.8394	1.2221	1,072
MegaVul_Java	1.9033	1.4191	345
MegaVul_C_2023	2.0327	1.2176	2,179
MegaVul_C_2024	2.0295	1.5246	2,179
SVEN	1.7208	1.5349	105
CodeXGlue_Medium_Java	1.5536	0.3776	9,819
CodeXGlue_Small_Java	0.5162	0.1895	8,753

The motivation for focusing on inference efficiency in models like CodeBERT and CodeT5 stems from the increasing integration of machine learning models in continuous integration and deployment pipelines, where rapid processing is critical. Lower inference times translate directly to cost savings in computational resources, making models like CodeT5 more viable for industrial applications where high-volume, real-time processing is essential. Moreover, the faster execution times associated with CodeT5 enable researchers and practitioners to experiment with larger datasets and iterate more frequently, accelerating the model development and testing phases.

In summary, Table 3 illustrates that CodeT5 not only outperforms CodeBERT in average inference time but does so consistently across diverse datasets, making it a promising tool for efficient and scalable software engineering solutions. This efficiency advantage may guide future choices in model selection for applications requiring low latency and high throughput, particularly as the demand for real-time code analysis continues to grow.

6.3 Impact of Patch Length (RQ3)

In addressing RQ3, we examined the impact of patch length on the accuracy of *CodeBERT* and *CodeT5*, using both *CodeBLEU* (Figure 2 and Figure 4) and *CrystalBLEU* (Figure 3 and Figure 5) scores across nine datasets. The results indicate that patch length significantly influences model performance, with each model varying in its ability to handle longer patches.

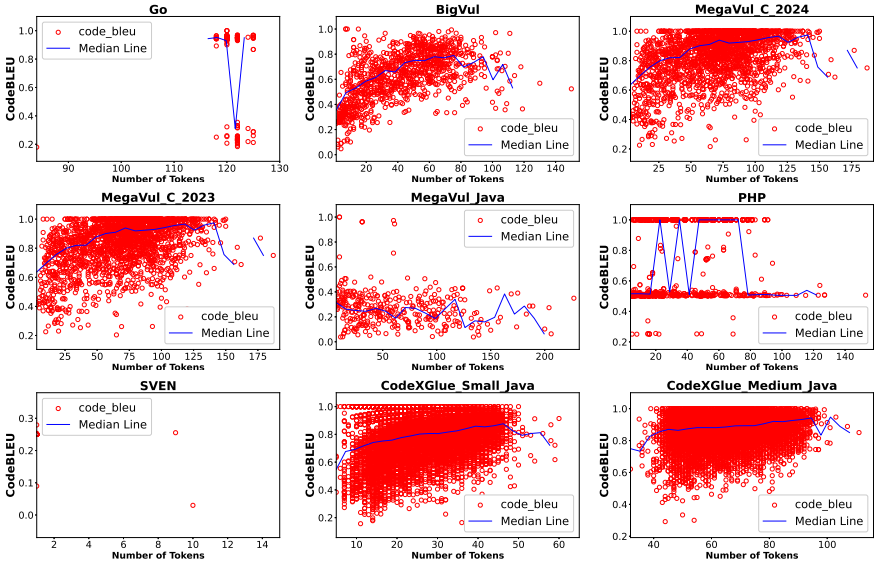

Fig. 2: Correlation Between Patch Length and CodeBLEU (CodeBERT)

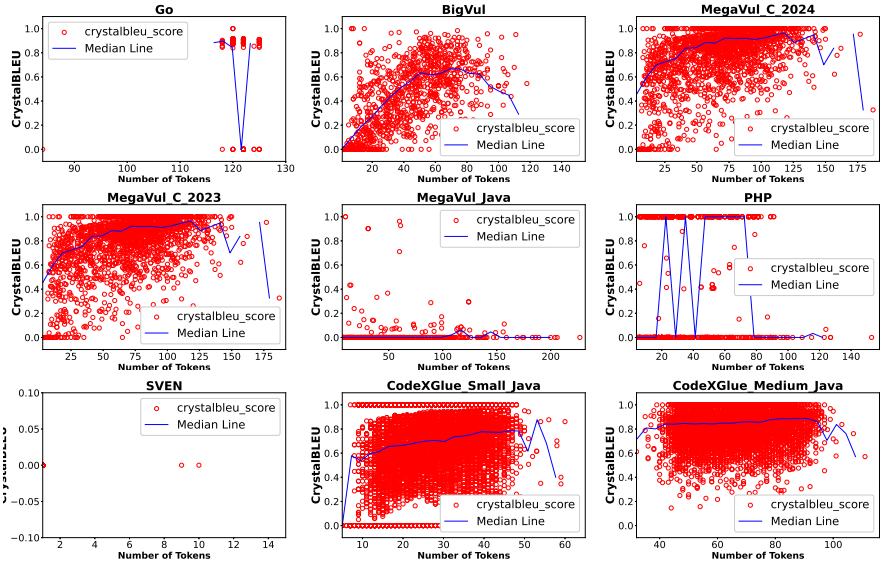

Fig. 3: Correlation Between Patch Length and CrystalBLEU (CodeBERT)

For *CodeBERT*, accuracy generally declines as patch length increases, as shown by both *CodeBLEU* and *CrystalBLEU* scores. Datasets like BigVul, MegaVul_C_2023, and MegaVul_Java exhibit sharp declines in accuracy for longer patches, highlighting the model's challenges in maintaining coherence over extended sequences. In the PHP dataset, performance fluctuates considerably, demonstrating instability in handling longer patches. However, on the CodeXGlue_Medium_Java dataset, *CodeBERT* performs relatively well, maintaining stable scores for longer patches, likely due to the dataset's consistent structure. In sparse datasets like SVEN, the notably low *CrystalBLEU* scores highlight significant struggles with fragmented contexts, as also observed in *CodeBLEU* evaluations. For the Go dataset, accuracy initially drops sharply but later retains for longer patches, highlighting challenges in preserving code structure at varying patch lengths.

CodeT5 exhibits similar variability in performance to *CodeBERT* across most datasets, indicating that both models face challenges in maintaining consistent accuracy with increasing patch lengths. In BigVul, MegaVul_C_2023, MegaVul_C_2024, and CodeXGlue_Medium_Java, both metrics indicate that *CodeT5* generally performs better than *CodeBERT*, demonstrating its relative effectiveness in handling extended sequences. However, maintaining accuracy with increasing patch lengths remains a challenge in datasets like PHP, where both models experience a decline in performance. While *CodeT5* demonstrates a marginally better ability to retain accuracy compared to *CodeBERT*, the differences are not substantial. Similarly, the sparse and fragmented data in the Go and SVEN datasets lead to inconsistent accuracy, particularly evident in *CrystalBLEU* scores, highlighting the difficulty of preserving syntactic and semantic integrity in such contexts.

When comparing the two models, *CodeT5* generally demonstrates better performance than *CodeBERT* in handling longer patches, although the differences in accuracy are not substantial across most datasets. Both models exhibit notable challenges with sparse or fragmented datasets, such as Go and SVEN, where longer patch lengths amplify performance issues. The variations observed between *CodeBLEU* and *CrystalBLEU* scores emphasize the difficulties in maintaining both syntactic and semantic integrity in extended patches. This highlights the complexity of generating accurate and contextually coherent patches for vulnerability-focused program repair as patch length increases.

7 Conclusion

In this study, we evaluated the effectiveness of two advanced pre-trained language models, CodeBERT and CodeT5, in addressing vulnerability-focused program repair across a varied collection of datasets. Through systematic experimentation, we analyzed their accuracy in patch generation, computational efficiency, and the impact of patch length on performance, providing valuable insights into their potential and limitations for automated vulnerability patching.

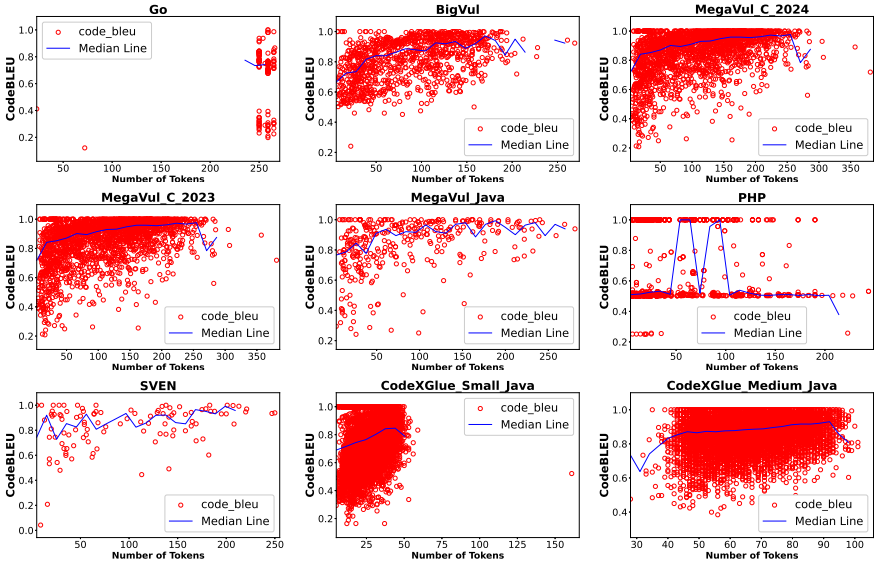

Fig. 4: Correlation Between Patch Length and CodeBLEU (CodeT5)

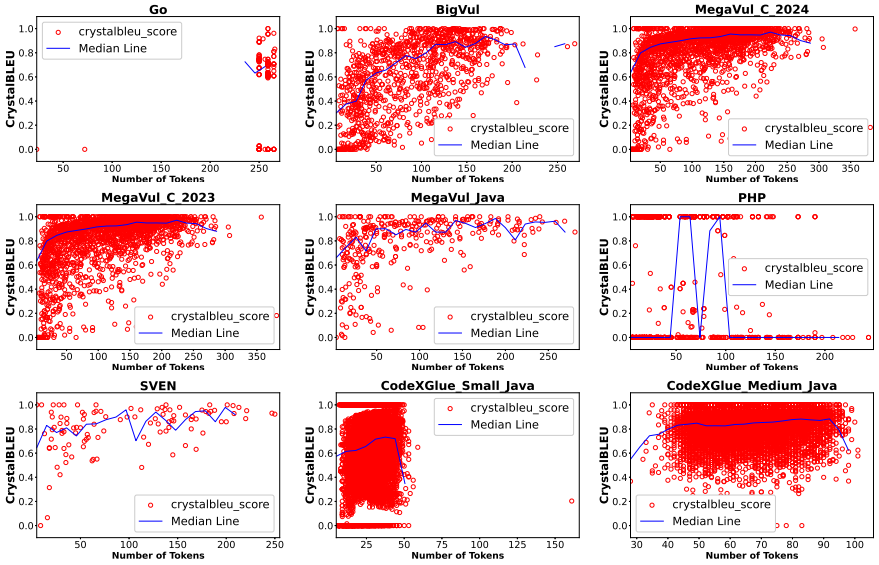

Fig. 5: Correlation Between Patch Length and CrystalBLEU (CodeT5)

Our findings show that CodeT5 outperformed CodeBERT in generating accurate patches, particularly for datasets with complex or diverse vulnerability patterns, such as BigVul and MegaVul. However, CodeBERT exhibited strengths in handling context-limited datasets like Go and PHP, where fragmented or incomplete code posed challenges.

In terms of computational efficiency, CodeT5 consistently demonstrated faster inference times across nearly all datasets, making it better suited for large-scale applications, such as integration into CI/CD pipelines or vulnerability assessments of extensive codebases. Meanwhile, CodeBERT's robustness in certain contexts suggests its simpler architecture can be advantageous for specific use cases, such as patching legacy systems or isolated code snippets.

The analysis of patch length revealed challenges for both models in generating accurate and coherent patches for extended sequences. While CodeT5 exhibited a relative advantage in maintaining performance across longer patches, both models experienced declines, emphasizing the need for future advancements in addressing longer contextual dependencies.

These findings underscore the complex interplay between dataset characteristics, model architecture, and task-specific requirements in vulnerability-focused program repair. Although CodeT5 demonstrates superior accuracy and computational efficiency, making it well-suited for large-scale and diverse datasets, its challenges with extended patch lengths reveal the limitations of current pre-trained models in managing long-range dependencies—a critical requirement for security patches in modern software ecosystems that demand nuanced and context-aware modifications. On the other hand, CodeBERT's performance in context-limited scenarios demonstrates the value of simpler architectures in addressing fragmented or incomplete code, suggesting that no single model can effectively address all aspects of automated vulnerability patching. These findings emphasize the need to align model selection and fine-tuning strategies with the unique characteristics of datasets and use cases, carefully balancing accuracy, efficiency, and adaptability. Future work should focus on hybrid approaches that combine the contextual understanding of advanced models like CodeT5 with the robustness of simpler architectures like CodeBERT, as well as improving dataset quality and diversity to better reflect the challenges of real-world vulnerability repair. Moreover, advancements in handling longer patch lengths, whether through enhanced pre-training strategies or specialized model architectures, are essential to overcome the bottlenecks observed in this study, paving the way for more scalable and effective solutions in automated vulnerability patching.

Acknowledgments. This research received funding from the European Commission through the Horizon Europe Programme as part of the LAZARUS project (`https://lazarus-he.eu/`) (Grant Agreement No. 101070303). The content of this article represents the sole responsibility of the authors and does not necessarily reflect the official views of the European Union.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Ahmet Okutan, Peter Mell, Mehdi Mirakhorli, Igor Khokhlov, Joanna C. S. Santos, Danielle Gonzalez, and Steven Simmons. Empirical validation of automated vulnerability curation and characterization. *IEEE Trans. Software Eng.*, 49(5):3241–3260, 2023.
2. Massimiliano Albanese, Hasan Çam, and Sushil Jajodia. Automated cyber situation awareness tools and models for improving analyst performance. In Robinson E. Pino, Alexander Kott, and Michael J. Shevenell, editors, *Cybersecurity Systems for Human Cognition Augmentation*, volume 61 of *Advances in Information Security*, pages 47–60. Springer, 2014.
3. Saad Khan and Simon Parkinson. Review into state of the art of vulnerability assessment using artificial intelligence. In Simon Parkinson, Andrew Crampton, and Richard Hill, editors, *Guide to Vulnerability Analysis for Computer Networks and Systems - An Artificial Intelligence Approach*, Computer Communications and Networks, pages 3–32. Springer, 2018.
4. Quang-Cuong Bui, Ranindya Paramitha, Duc-Ly Vu, Fabio Massacci, and Riccardo Scandariato. Apr4vul: an empirical study of automatic program repair techniques on real-world java vulnerabilities. *Empir. Softw. Eng.*, 29(1):18, 2024.
5. David de-Fitero-Dominguez, Eva García-López, Antonio García-Cabot, and José Javier Martínez-Herráiz. Enhanced automated code vulnerability repair using large language models. *Eng. Appl. Artif. Intell.*, 138:109291, 2024.
6. Goran Piskachev, Matthias Becker, and Eric Bodden. Can the configuration of static analyses make resolving security vulnerabilities more effective? - A user study. *Empir. Softw. Eng.*, 28(5):118, 2023.
7. Aayush Garg, Renzo Degiovanni, Matthieu Jimenez, Maxime Cordy, Mike Papadakis, and Yves Le Traon. Learning to predict vulnerabilities from vulnerability-fixes: A machine translation approach. *CoRR*, abs/2012.11701, 2020.
8. Niklas Risse and Marcel Böhme. Uncovering the limits of machine learning for automatic vulnerability detection. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.
9. Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
10. Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.
11. Reza Gharibi, Mohammad Hadi Sadreddini, and Seyed Mostafa Fakhrahmad. T5APR: empowering automated program repair across languages through checkpoint ensemble. *J. Syst. Softw.*, 214:112083, 2024.
12. Aayush Garg, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. On the coupling between vulnerabilities and llm-generated mutants: A study on vul4j dataset. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2024, Toronto, ON, Canada, May 27-31, 2024*, pages 305–316. IEEE, 2024.
13. Nguyen Ngoc Hai Dang, Tho Quan Thanh, and Anh Nguyen-Duc. *BERTVRepair: On the Adoption of CodeBERT for Automated Vulnerability Code Repair*, pages 173–196. Springer Nature Switzerland, Cham, 2024.

14. Michael Ogata, Joshua Franklin, Jeffrey Voas, Vincent Sritapan, and Stephen Quirolgico. Vetting the security of mobile applications. Technical report, National Institute of Standards and Technology, 2019.
15. Aayush Garg, Constantinos Patsakis, Zanis Ali Khan, and Qiang Tang. Payload analysis of adversaries' tooling: Automated identification of fuzzers. *techrxiv preprint*, December 2024.
16. Andrei Arusoaie, Stefan Ciobâca, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 161–168, 2017.
17. Zhangyin Feng, Daya Guo, Duyu Tang, et al. Codebert: a pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547. Association for Computational Linguistics, November 2020.
18. Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.
19. Yuejun Guo, Constantinos Patsakis, Qiang Hu, Qiang Tang, and Fran Casino. Outside the comfort zone: Analysing llm capabilities in software vulnerability detection. In *European symposium on research in computer security*, pages 271–289. Springer, 2024.
20. Aayush Garg, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Vulnerability mimicking mutants. *CoRR*, abs/2303.04247, 2023.
21. Yuejun Guo, Qiang Hu, Qiang Tang, and Yves Le Traon. An empirical study of the imbalance issue in software vulnerability detection. In *European Symposium on Research in Computer Security*, pages 371–390. Springer, 2023.
22. Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.
23. Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 31–42, New York, NY, USA, 2019. Association for Computing Machinery.
24. Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering (ICSE)*, pages 3–13, 2012.
25. Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 802–811. IEEE, 2013.
26. Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.
27. Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In *35th International Conference on Software Engineering (ICSE)*, pages 772–781, 2013.

28. Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: scalable multi-line program patch synthesis via symbolic analysis. In *38th International Conference on Software Engineering (ICSE)*, pages 691–701, 2016.
29. Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43:34–55, 2017.
30. Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(09):1943–1959, September 2021.
31. Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. Codit: code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 48(4):1385–1399, 2018.
32. Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J. Hellendoorn. Patching as translation: the data and the metaphor. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE'20, page 275–286, New York, NY, USA, 2021. Association for Computing Machinery.
33. Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. Tfix: learning to fix coding errors with a text-to-text transformer. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*. PMLR, 18–24 Jul 2021.
34. Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. Gamma: Revisiting template-based automated program repair via mask prediction. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 535–547. IEEE, 2023.
35. Pasquale Salza, Christoph Schwizer, Jian Gu, and Harald C. Gall. On the effectiveness of transfer learning for code search. *IEEE Trans. Software Eng.*, 49(4):1804–1822, 2023.
36. Divyam Goel, Ramansh Grover, and Fatemeh H. Fard. On the cross-modal transfer from natural language to code through adapter modules. In Ayushi Rastogi, Rosalia Tufano, Gabriele Bavota, Venera Arnaoudova, and Sonia Haiduc, editors, *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022*, pages 71–81. ACM, 2022.
37. Aayush Garg, Renzo Degiovanni, Facundo Molina, Maxime Cordy, Nazareno Aguirre, Mike Papadakis, and Yves Le Traon. Enabling efficient assertion inference. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023*, pages 623–634. IEEE, 2023.
38. Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. Bridging pre-trained models and downstream tasks for source code understanding. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022.
39. Jian Guan, Xiaoxi Mao, Changjie Fan, Zitao Liu, Wenbiao Ding, and Minlie Huang. Long text generation by modeling sentence-level and discourse-level coherence. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 6379–6393. Association for Computational Linguistics, 2021.

40. Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Trans. Assoc. Comput. Linguistics*, 12:157–173, 2024.
41. Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel C. Briand. Impact of log parsing on deep learning-based anomaly detection. *Empir. Softw. Eng.*, 29(6):139, 2024.
42. Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel C. Briand. Guidelines for assessing the accuracy of log message template identification techniques. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1095–1106. ACM, 2022.
43. Aayush Garg, Renzo Degiovanni, Matthieu Jimenez, Maxime Cordy, Mike Papadakis, and Yves Le Traon. Learning from what we know: How to perform vulnerability prediction using noisy historical data. *Empir. Softw. Eng.*, 27(7):169, 2022.
44. Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.
45. Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.
46. Aryaz Eghbali and Michael Pradel. Crystalbleu: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.
47. Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the BLEU: how should we assess quality of the code generation models? *J. Syst. Softw.*, 203:111741, 2023.
48. Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA, 2020. Association for Computing Machinery.
49. Chao Ni, Liyu Shen, Xiaohu Yang, Yan Zhu, and Shaohua Wang. Megavul: A c/c++ vulnerability dataset with comprehensive code representations. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, New York, NY, USA, 2024. Association for Computing Machinery.
50. Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1865–1879, New York, NY, USA, 2023. Association for Computing Machinery.
51. Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.
52. Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. How effective are neural networks for fixing security vulnerabilities. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1282–1294. ACM, 2023.
53. Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing 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.