

**CS5351 Research Paper**

**(2024/2025 Semester B)**

**Group 6**

**Research Paper Title**

NAME 724XXXXX

|  |  |  |  |
| --- | --- | --- | --- |
| **Group Members** | | | |
| HAO YANG | 72403219 | YIMENG WANG | 72404093 |
| XUANHAO YAN | 72405380 | XI LUO | 72403582 |
| LU PENG | 72401459 | XINYU HU | 72405606 |
| JIEPEI CHEN | 72405876 | CIYUAN YU | 72402757 |

**Content**

[Code Generation and Optimization 3](#_Toc14277)

[Section 1: [HAO YANG] 3](#_Toc15263)

[Section 2: [XUANHAO YAN] 6](#_Toc28842)

[Section 3: [XI LUO] 8](#_Toc1201)

[Section 4: [LU PENG] 10](#_Toc4185)

[Code Review and Quality Control 12](#_Toc12955)

[Section 5: [JIEPEI CHEN] 12](#_Toc26325)

[Section 6: [YIMENG WANG] 15](#_Toc31002)

[System Safety and Reliability 17](#_Toc30115)

[Section 7: [CIYUAN YU] 17](#_Toc25965)

[Section 8: [XINYU HU] 19](#_Toc10129)

[Group Summary 21](#_Toc10128)

[References 23](#_Toc9092)

**Code Generation and Optimization**

**Section 1: [HAO YANG]**

**Language Models for Code Completion: A Practical Evaluation**

**1 Motivation**

The rising popularity of transformer-based language models has impacted development practices, with more developers increasingly adopting these models for code completion. Although these models perform well theoretically, there exist lots of challenges in real development environments. Various models proposed in research are complex and precise, but only a few of them can be integrated into development tools.

Additionally, existing research mostly focuses on proprietary models and their performance in specific scenarios, lacking a systematic evaluation of how these models perform in a real-world development environment, especially in the analysis of performance degradation. This is a critical issue for software engineers because the performance of development tools directly impacts development efficiency. Therefore, the authors aim to promote the transition of code completion models from theoretical research to engineering applications through systematic studies and open evaluations, ultimately providing developers with more practical and efficient development tools.

**2 Contributions**

This research addresses the insufficient evaluation of existing code completion models in real development environments by proposing a hybrid evaluation method based on real-world data. First, the authors developed an open-source IDE plugin called Code4Me, which integrates three models—InCoder, UniXcoder, and CodeGPT—and collected over 600,000 real-world completion instances from more than 1,200 developers across 12 programming languages through this plugin. To comprehensively evaluate model performance, the authors employed a combination of quantitative and qualitative evaluation methods: in the quantitative analysis, six standard metrics, including ROUGE-L and acceptance rate, were used to assess model performance across different languages; in the qualitative analysis, 1,690 completion requests were open-coded, leading to the identification of 18 categories of reasons for completion failures. Additionally, the authors compared online (real-world scenarios) and offline (synthetic data) evaluation results, analyzed the impact of random masking and trigger-point masking strategies on model performance, and revealed significant discrepancies between traditional offline evaluations and actual developer usage scenarios.

The experimental results demonstrate that InCoder, leveraging its diverse training data and code infilling objectives, outperforms other models across all 12 programming languages, particularly excelling in mainstream languages like Python and Java (ROUGE-L exceeding 40). However, offline evaluations significantly overestimated model performance (ROUGE-L over 60), while online evaluations revealed a mere 4.91% acceptance rate for completions in mainstream languages, with 66.3% of failures stemming from inherent model limitations, such as incorrect variable name predictions. Further analysis of developer behavior shows that 52% of completion requests require modifications to the right-side context, and 21% necessitate code edits, underscoring existing models' deficiencies in context awareness and localized modification capabilities. Based on these findings, the authors propose enhancements, including cross-language training, context-aware augmentation, intelligent triggering mechanisms, and support for multiple completion suggestions, to better align with developers' real-world needs.

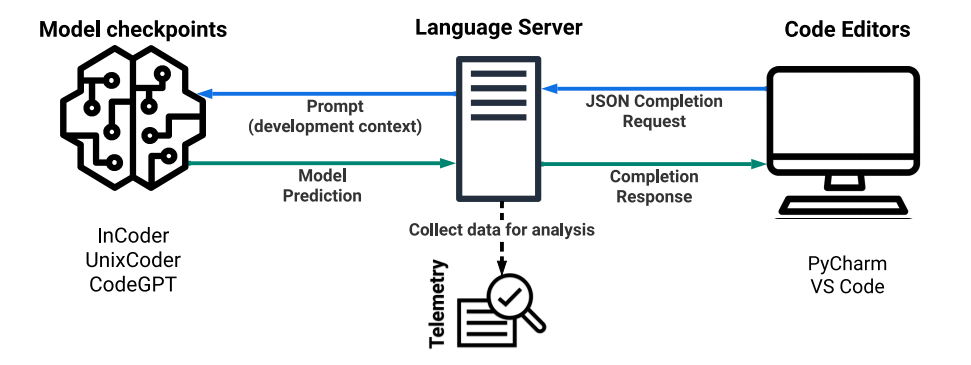


Figure 1: Code editors and models communication pipeline [1]

**3 Assessment**

The paper's strength lies in its extensive real-world dataset and dual-mode evaluation, offering practical insights. However, limitations include potential biases in user-selected contexts, a disparity between offline and online performance, and limited exploration of underrepresented languages.

However, this study has certain limitations. While it utilizes real-world data from Code4Me, the restricted user base may limit its representativeness. Additionally, only three models—InCoder, UniXcoder, and CodeGPT—are evaluated, excluding major commercial models like GitHub Copilot and Amazon CodeWhisperer. The assessment primarily relies on traditional metrics such as ROUGE-L and acceptance rate, which may not fully capture semantic correctness or developer preferences. Moreover, although the study emphasizes refining training objectives, it lacks concrete strategies for optimizing data and model architectures to address issues like variable naming errors and inaccurate parameter predictions.

**4 Improvement**

Regarding limitations, improvements can be made in two key aspects. On the one hand, data collection and evaluation methods can be expanded. Existing studies are constrained by reliance on a single tool as the data source and traditional evaluation metrics. Future research could integrate data from developers across different IDEs, regions, and industries to ensure a more representative user sample. Additionally, comparisons with commercial products (e.g., GitHub Copilot, CodeWhisperer) could provide a more comprehensive evaluation. Furthermore, fine-grained evaluation criteria such as semantic correctness and contextual appropriateness should be introduced, along with human review, to compensate for the limitations of traditional metrics like ROUGE-L in capturing the true quality of generated code.

On the other hand, improvements in model training and architectural optimization are necessary. Incorporating richer code context and structured data, such as static code analysis and project dependencies, could enhance the model's accuracy in variable naming and parameter prediction. Reinforcement learning techniques could also be leveraged to learn from real-world developer interactions, thereby mitigating issues of semantic errors and redundant or verbose code generation. For instance, multimodal training strategies from projects like Codexglue[2] could be explored to develop more efficient training methodologies, further narrowing the gap between theoretical advancements and practical applications.

**5 Self-reflection**

The reason for choosing this article is that I also utilize large language models to assist with coding tasks, such as leveraging Copilot to explain codes or help me debug issues. Therefore, I am particularly interested in models that leverage large language models to aid programmers in their development work.

After reading the article and reflecting on my own programming habits, I have come to realize that large language models can sometimes struggle with mismatched contextual code relationships in real-world programming environments, and they may even influence the coding habits of developers. Therefore, when evaluating code completion systems, we should also focus on the interaction habits of developers themselves. By tailoring prompts to different users, we can enhance the model's adaptability to diverse developer preferences for the same programming task.

**Section 2: [XUANHAO YAN]**

**ECFuzz Effective Configuration Fuzzing for Large-Scale Systems**

**1 Motivation**

Modern distributed systems rely heavily on intricate configuration settings to function properly. With hundreds or even thousands of adjustable parameters, these systems present unique testing challenges that conventional methods struggle to address effectively. Traditional approaches often examine parameters in isolation, missing critical interactions between settings that can lead to system failures. This oversight results in inefficient testing processes and overlooked vulnerabilities.

**2 Contributions**

The ECFuzz methodology introduces two groundbreaking techniques to revolutionize configuration testing. First, it implements an advanced parameter generation system that recognizes and accounts for five distinct types of parameter relationships. Unlike simpler methods, this approach doesn't just randomly alter values - it intelligently adjusts groups of related parameters together. For instance, when dealing with control dependencies, it ensures enabling parameters are properly set before modifying dependent values.

Second, the system employs a novel validation process that significantly reduces testing overhead. By leveraging existing unit tests as an initial filter, it can quickly eliminate unpromising configurations before committing resources to full system testing. This dual-phase approach has demonstrated remarkable results in real-world applications, identifying critical issues that other tools miss while dramatically reducing wasted testing time.

**3 Assessment**

In comprehensive evaluations across five major open-source platforms, this approach outperformed existing solutions by wide margins. The testing framework uncovered numerous previously undetected flaws, including several that could cause serious system failures. One particularly notable case involved a configuration combination that could potentially wipe critical system directories if deployed improperly.

The system's effectiveness stems from its nuanced understanding of configuration relationships combined with its efficient testing pipeline. Where conventional methods might test thousands of configurations blindly, this approach strategically focuses testing efforts where they're most likely to reveal meaningful issues.

**4 Improvement**

While the current implementation shows great promise, opportunities for improvement remain. The dependency mapping process, while effective, currently requires some manual input. Future versions could incorporate runtime analysis to automatically detect parameter relationships. The testing framework might also benefit from adaptive algorithms that learn from previous test cycles to further optimize the testing process.

For organizations managing complex systems, this represents a significant step forward in configuration testing. By combining intelligent parameter analysis with efficient validation techniques, it offers a practical solution to a longstanding challenge in system reliability and security. The open-source nature of the project encourages community involvement and continuous refinement, ensuring it remains at the forefront of configuration testing technology.

**Section 3: [XI LUO]**

**Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning**

**1 Motivation**

The paper addresses a critical limitation in code comment generation: existing approaches generate comments describing only one aspect of a code snippet, while developers often need comments summarizing code from multiple perspectives (e.g., functionality, usage, implementation). This one-to-one mapping fails to meet developers' needs for comprehensive code understanding. The study explores the potential of large language models (LLMs) to generate multi-intent comments, leveraging their pre-trained knowledge of code and natural language semantics.

**2 Contributions**

The paper tackles the \*\*multi-intent comment generation\*\* problem, where the goal is to generate multiple comments for a single code snippet, each addressing a different intent (e.g., "what," "why," "how-to-use"). This is a significant improvement over traditional approaches that produce only one comment per code snippet.

The study employs "in-context learning" with OpenAI's Codex, where the model is provided with prompts containing examples and instructions. Two key strategies are introduced: 1) Demonstration Selection: Token-based and semantic-based strategies are used to select similar code snippets from a corpus to construct effective prompts. 2) Reranking: Generated comments are reranked based on their similarity to comments from similar code snippets, using token-based and semantic-based metrics.

LLMs outperform the state-of-the-art supervised learning approach (DOME) when adequately prompted with 10 examples. Semantic-based demonstration selection and token-based reranking yield the best results, with significant improvements in BLEU, ROUGE-L, and METEOR scores. Human evaluation confirms that LLM-generated comments are more natural, adequate, and useful than those from DOME.

Experiments are conducted on two Java datasets (Funcom and TLC). The study evaluates zero-shot, one-shot, and few-shot learning settings, with and without demonstration selection and reranking strategies. Performance is measured using BLEU, ROUGE-L, and METEOR metrics, and a human evaluation is performed to assess comment quality.

**3 Assessment**

The study demonstrates that LLMs can effectively generate multi-intent comments, establishing a new baseline for this task. The in-context learning approach avoids the need for task-specific fine-tuning, making it more flexible and scalable. The proposed demonstration selection and reranking strategies significantly improve comment quality.

The study is limited to Java, and it is unclear how well the approach generalizes to other programming languages. The reliance on pre-trained LLMs raises concerns about data leakage, as the model may have seen the test data during pre-training. The reranking strategies are based on simple heuristics, which may not fully capture the semantic nuances of comments.

**4 Improvement**

**Code Complexity Analysis:** Analyze the target code snippet to determine its complexity (e.g., number of tokens, depth of control flow, presence of advanced constructs like recursion or concurrency). Use metrics like cyclomatic complexity or token count to classify the code as simple, moderate, or complex.

**Intent-Specific Example Selection:** Based on the desired intent (e.g., "what," "why"), retrieve examples from the corpus that are not only similar to the target code but also align with the intent. For example, if the intent is "how-to-use," prioritize examples that describe usage scenarios or API calls.

**Adaptive Example Count:** Adjust the number of examples in the prompt based on the complexity of the target code. For simple code, fewer examples may suffice, while complex code may require more examples to guide the model effectively.

**Real-Time Feedback Loop:** Incorporate a feedback mechanism where the model's initial outputs are evaluated (e.g., using metrics like BLEU or semantic similarity), and the prompt is iteratively refined to improve performance.

**Section 4: [LU PENG]**

**Automatic Semantic Augmentation of Language Model Prompt（for code summarization)**

**1 Motivation**

A key challenge identified in the paper is that large language models often overlook crucial code semantics when generating summaries. In particular, LLMs do not automatically extract or incorporate many of the same facts that software developers routinely notice when reading code—like the dataflow connections between variables, the significance of function names, or the project context from directory structure. The paper demonstrates that incorporating these "semantic facts" into the LLM prompt can substantially improve code summarization quality, yet the practical question remains: how do we implement this seamlessly?

**2 Contributions**

**Build a Lightweight Analysis Pipeline:**

Static Parsing: Use an existing parser (e.g., Treesitter) to generate an Abstract Syntax Tree (AST) from each function. From this AST, extract metadata such as function names, parameter lists, and variable declarations. Tag each identifier with its syntactic role (function name, parameter, local variable, etc.).

DataFlow Extraction: Apply a basic dataflow analysis to identify the flow of values between variables—particularly valuable in languages that heavily rely on variable passing or mutation. This can be done using offtheshelf libraries or small custom scripts that track definitions and uses of variables in each function.

Repository/Path Gathering: Parse the code repository structure (e.g., using Git commands or a filetree scan) to capture folder names, file paths, or module names. These typically reveal domain clues: whether the code belongs to "android," "network," or "graphics," which can guide more accurate summaries.

**Assemble an Augmented Prompt:**

Once the static facts have been gathered, structure them in a concise format that the model can digest. For example, prefix the code snippet with lines like `Repo: "owner/project"`, then a block labeled "Tagged Identifiers," and finally a short listing of dataflow connections. This is the semantic augmentation.

Follow it with a minimal number of exemplar codecomment pairs, retrieved from a local database or training set via BM25 (or another IR technique). Each exemplar also includes its own analysis output and final summary.

**LLM Integration:**

With the augmented prompt complete, send it to the model (e.g., GPT3.5turbo or Codex) for comment generation. Depending on the model's API, store or cache repeated parts of the prompt (like frequently used exemplars) to reduce overhead and cost.

If the function is large or the dataflow is intricate, apply sizebased trimming strategies (e.g., limiting the number of dataflow entries) to keep the prompt within token limits.

**3 Self-reflection**

By systematically extracting and embedding code semantics into the LLM prompt, this solution expands the model's effective "knowledge" to include the same facts developers rely on. The result is more concise and contextaware summaries, offering greater clarity and utility for maintenance or onboarding tasks in software projects.

**Code Review and Quality Control**

**Section 5: [JIEPEI CHEN]**

**AutoTransform: Automated Code Transformation to Support Modern Code Review Process**

**1 Motivation**

Code review is an important quality assurance practice in software development, but it still relies on a lot of manual work. Developers need to manually modify the code until it is approved. Although previous studies have proposed using neural Machine Translation (NMT) methods to automatically transform code into vet-approved versions, existing methods do not perform well when new identifiers or literals (such as renaming variables) appear in "after versions" and long code sequences. These are also the key problems that this paper focuses on.

**2 Contributions**

This paper presents AutoTransform, which leverages Byte pair encoding (BPE) to process new tokens and employs a Transformer-based NMT architecture to process long sequences. Code tokens are split into common subword units by BPE. By dividing code tokens into common subword units, BPE effectively reduces the vocabulary size and allows the model to generate new code tokens based on the existing subword combinations in the training data. The Transformer architecture relies entirely on the attention mechanism and is better able to handle long-distance dependencies, thus improving the accuracy of code transformation.

The experiment is based on 147,553 change methods extracted from the Gerrit code review repositories of Android, Google, and Ovirt. The data is divided into change methods with and without new tags, and further divided into small-scale (≤50 tags) and medium-sized (51-100 tags). BPE is used for subword segmentation, and Transformer and RNN-based NMT models are trained separately. The effectiveness of AutoTransform is verified by comparing the performance of different models. On a dataset containing 14,750 change methods, AutoTransform can correctly transform 1,413 change methods at beam width = 1, beating existing methods by 567%. For the change method in "after version" whether it contains new tags or not, the method in this paper can achieve method transformation efficiently and correctly. Through ablation studies, we demonstrate the significant contribution of BPE and Transformer architectures to the performance improvement.

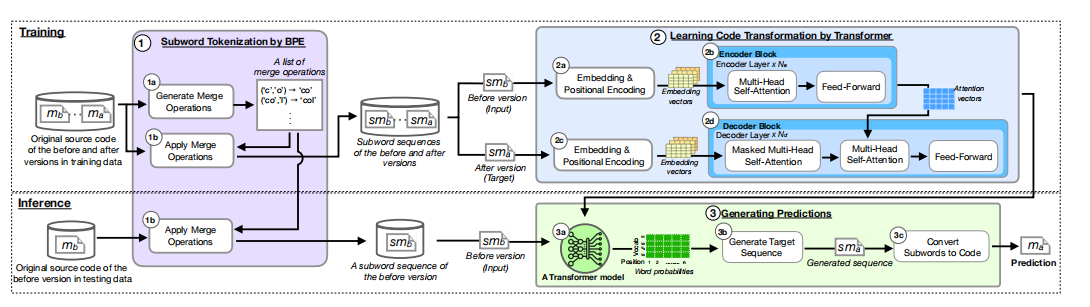


Figure 2: Overview of AutoTransform [3]

**3 Assessment**

Automated code transformation reduces manual intervention, avoids errors caused by fatigue or inattention, and improves the efficiency and quality of code reviews. At the same time, AutoTransform can deal with new identifiers or literals appearing in "after version", which solves the limitation of existing methods that can not deal with new tags, and has good performance when dealing with long code sequences. The drawback is that the model may not accurately generate the target code if the new tokens in the "after version" are significantly different from the subword combinations in the training data. And Transformer models are very sensitive to parameters, and the optimization process is difficult.

**4 Improvement**

From a model perspective, more lightweight Transformer architectures such as TinyBERT can be explored to reduce the training and inference time of the model while maintaining high performance. In addition, more context information can be introduced in the code transformation process, such as project dependencies, code specification comments, documentation, etc., to improve the accuracy and consistency of the generated code. In order to improve the model's ability to process long code sequences, especially when dealing with complex logic and nested structures, more efficient Transformer variants or block processing methods can be used to improve the model's ability to process complex code structures and further expand its scope of application. AutoTransform is mainly optimized for Java language, and its generalization ability may be limited. In order to improve the robustness and generalization ability of the model, we can train code snippets in multiple languages such as Python, C++, etc., to improve the cross-language generalization ability of the model.

**5 Self-reflection**

By introducing Byte-pair Encoding (BPE) and Transformer architecture, this paper effectively solves the problems of new token generation and long sequence processing in code transformation, which inspires me to dare to break through the limitations of traditional methods and find new technologies that are more suitable for the nature of the problem. Secondly, in terms of experimental design, the authors clearly demonstrate the necessity and effectiveness of improvement by comparing the performance of different methods. This comparative experimental method provides me with an important idea to verify the innovation point in scientific research, that is, to highlight the advantages of the new method through comparison with the existing technology. In addition, the optimization strategies mentioned in the article for long code sequences, such as dynamic padding and the memory-efficient attention mechanism of XFormers, made me realize the importance of optimizing details to improve model performance. These optimization strategies are not only suitable for code transformation tasks, but also provide reference for processing other long sequence data. At the same time, the multi-model integration and human intervention mechanisms mentioned in this paper further emphasize the importance of combining human experience and model prediction in automated tasks. The comprehensive research method of this paper helps me to better apply the research results in complex practical scenarios.

**Section 6: [YIMENG WANG]**

**ChatGPT Incorrectness Detection in Software Reviews**

**1 Motivation**

With AI tools like ChatGPT becoming a go-to resource for software engineers, there's a growing concern about the accuracy of their responses. Developers often treat ChatGPT as an intelligent assistant, but what happens when it confidently provides incorrect answers? This paper tackles the issue by introducing a method to automatically detect such inaccuracies, reducing the risk of AI-driven misinformation in software engineering (SE).

**2 Contributions**

The main challenge here is that ChatGPT can be wrong, but in a convincing way. This can mislead developers, especially in high-stakes tasks like selecting libraries, debugging, or understanding complex SE concepts. The authors propose a system, CID (ChatGPT Incorrectness Detector), to help filter out unreliable responses.

CID is built on iterative prompting, metamorphic testing, and machine learning (ML)-based classification. Instead of blindly trusting ChatGPT, CID challenges it by rephrasing questions and checking if the answers remain consistent. If inconsistencies arise, there's a high chance the response is unreliable. The model uses Support Vector Machines (SVM) for classification, showing promising results.

The study found that inconsistencies correlate with incorrectness, and CID achieves an F1-score of 0.74-0.75 in detecting unreliable responses. Mutated prompts (slightly modified questions) improved performance, showing that ChatGPT struggles with robustness in SE-related queries.

The authors evaluated CID on 100 Stack Overflow-based queries, focusing on software library recommendations. They compared CID's results across different ML classifiers, confirming that SVM outperformed other models.

**3 Assessment**

Strengths: CID is automated, scalable, and model-agnostic, meaning it can work across different LLMs. The use of iterative questioning mimics how humans verify information.

Limitations: The assumption that inconsistencies always mean incorrectness isn't foolproof—ChatGPT can rephrase correct responses in different ways. Also, relying on textual similarity alone may not capture deeper conceptual mistakes. Visualizing confidence vs. correctness trends in a heatmap could provide further insights.

**4 Improvement and Self-reflection**

While CID is a step in the right direction, it's not enough to just detect inconsistencies—we need to understand why ChatGPT makes mistakes. Right now, CID assumes that if ChatGPT contradicts itself, it must be wrong. But in reality, some inconsistencies are just variations in phrasing, not genuine errors.

To improve CID, I propose integrating semantic validation—comparing ChatGPT's responses to trusted knowledge sources like official documentation or APIs. This would reduce false positives where ChatGPT simply gives different but valid explanations.

Additionally, CID's textual similarity metric is too surface-level. For a deeper check, we could incorporate BERT-based embeddings or knowledge graph verification, allowing CID to understand context, not just wording.

Another interesting idea is to introduce a confidence threshold for inconsistencies. If ChatGPT gives two conflicting answers, CID could ask a third variation of the question and check for a majority consensus. This would make the system more robust against minor variations.

Lastly, making CID interactive—allowing developers to provide feedback on false positives/negatives—would enable continuous learning, making CID a smarter and more adaptive tool over time.

By addressing these gaps, CID could evolve into a more intelligent AI reliability checker, benefiting developers far beyond just library selection.

**System Safety and Reliability**

**Section 7: [CIYUAN YU]**

**Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation.**

**1 Motivation**

Graph Database Management Systems (GDBMS) are extensively used in applications like social networks and recommendation engines. However, due to the complexity of their query processing, they are prone to logic bugs such as incorrect path matching or counting errors. Existing testing approaches like differential testing often suffer from high false positives due to implementation differences across systems. Other tools like GDBMeter focus mainly on testing predicates, neglecting the graph patterns (G-Pattern) crucial to many queries.

**2 Contributions**

The paper proposes a novel technique called Graph Query Transformation (GQT), which systematically transforms graph queries using injective and surjective mappings. These transformations generate test cases—categorized as equivalent, generalized, or restrictive—that are validated based on expected result relationships (e.g., equality or subset). A transformation like SymmetricPattern reverses edge directions to produce equivalent queries, and discrepancies in output signal logic bugs.

Based on GQT, the authors developed GraphGenie, a testing tool evaluated on six mainstream GDBMSs including Neo4j and RedisGraph. It discovered 25 previously unknown issues: 16 logic bugs, 3 internal errors, and 6 performance problems. Importantly, GraphGenie reported zero false positives, significantly outperforming existing tools such as Grand (with over 80% false positives) and even identified bugs missed by GDBMeter.

**3 Assessment**

The main merit of this work lies in its systematic and semantics-aware testing methodology, which ensures high accuracy and meaningful bug detection. However, a key limitation is that GraphGenie heavily depends on GDBMS-specific schemas and syntax. This may restrict its generalizability to systems with custom designs or non-standard query languages. Furthermore, its reliance on Cypher and Gremlin means it might not support all types of graph query languages.

**4 Improvement**

To improve GraphGenie, several enhancements could be considered:

First, incorporating machine learning could enable the system to learn from past bug patterns and prioritize high-probability test cases. This would be especially useful in large-scale datasets, where exhaustive testing is computationally expensive.

Second, the support for complex graph structures could be enhanced by introducing dynamic and adaptive mutation strategies. For instance, by randomizing path lengths or cycle structures during test case generation, the tool could more effectively stress-test the query engine’s handling of edge cases.

Third, to reduce computational overhead, a layered testing strategy could be introduced. By prioritizing high-impact transformations—like those that often reveal bugs—the system could reduce redundant mutations and focus resources on high-yield queries.

Finally, semantic constraints should be integrated into mutation rules. For example, including logical conflicts such as CREATE\_TIME > DELETE\_TIME or access control violations could expose deeper bugs in GDBMS logic that are currently overlooked.

These improvements would strengthen the framework by increasing mutation diversity, improving scalability, and enhancing its ability to detect subtle, real-world logic flaws across a broader range of systems.

**Section 8: [XINYU HU]**

**MalwareTotal: Multi-Faceted and Sequence-Aware Bypass Tactics against Static Malware Detection**

**1 Motivation**

Static malware detection plays a crucial role in cybersecurity but remains vulnerable to evasion tactics. Attackers continuously develop novel bypass strategies to evade detection by modifying malware features. This paper aims to systematically study and enhance our understanding of malware evasion tactics against static detection models.

**2 Contributions**

The study addresses how attackers can craft malware variants that evade detection while preserving malicious functionality. It proposes a novel framework, MalwareTotal, which simulates multi-faceted and sequence-aware evasion tactics against static detection models.

The authors introduce a structured approach that employs adversarial learning and sequence-aware transformations to systematically analyze static detection weaknesses. This approach models malware evolution and provides insights into improving static defenses.

Experiments demonstrate that MalwareTotal successfully evades various state-of-the-art static malware detection models with a significant reduction in detection rates. This highlights critical vulnerabilities in existing detection frameworks. The study evaluates multiple static malware detection models against MalwareTotal-generated adversarial samples. Metrics such as evasion success rate and feature importance analysis are used to assess detection robustness.

**3 Assessment**

**Merits:** 1) Provides a systematic evaluation of malware evasion strategies. 2) Introduces a sequence-aware adversarial approach, improving realism in evasion attacks.

**Limitations:**1) Focuses only on static detection, while many modern systems integrate dynamic analysis. 2) Adversarial modifications may not generalize well against hybrid or deep learning-based security models.

**4 Improvement**

To enhance the robustness of static malware detection, I propose incorporating hybrid detection techniques that combine static and dynamic analysis. While MalwareTotal effectively exploits static model weaknesses, dynamic behavior analysis could mitigate evasion attempts. By monitoring execution traces and runtime API calls, security systems could detect subtle behavioral similarities among malware families, reducing reliance on static features alone.

Additionally, adaptive adversarial training could help strengthen static models. Instead of training on fixed datasets, security models should iteratively update using adversarially generated malware samples from MalwareTotal. This would improve resilience against evasion attacks by continuously adapting to new threats.

Finally, ensemble learning can enhance detection by integrating multiple classifiers with different feature extraction techniques. A combination of static code analysis, dynamic execution monitoring, and machine learning-based heuristics could create a more robust defense mechanism.

These improvements ensure malware detection systems remain resilient against evolving evasion tactics while leveraging the strengths of multiple detection paradigms.

**Group Summary**

**ATM: Black-box Test Case Minimization based on Test Code Similarity and Evolutionary Search**

Despite the excellent performance of ATM[10] in fault detection rate, its average execution time of 1.1 to 4.3 hours may still be too long for large industrial systems. This may limit its widespread use in real-world development, especially in scenarios that require fast feedback. We can try to speed up the execution time by parallelizing the computation.

In the ATM method, the redundancy removal process relies on similarity measures to identify redundant test cases for elimination. However, this approach may be inefficient, waste plenty of time or fail to effectively identify redundant test cases, especially in complex test sets, or in the context of security testing. Based on the ideas in the AIM (Automated Input Minimization)[11] method, input selection and redundancy removal techniques can be enhanced to further optimize ATM's performance. AIM introduces a black-box clustering-based input selection strategy, which clusters test cases based on their security features, and retains representative test cases from each cluster. This not only reduces redundancy but also ensures the coverage of different vulnerabilities.

**Elicitation:**

Interview test engineers and developers about their execution time requirements for test case minimization tools in real-world development, especially in scenarios that require rapid feedback. Collect the needs and expectations of the development team for parallelized computing, focusing on the applicability of the tool in multicore processors or distributed computing environments.

Analyze the execution time bottleneck of existing ATM tools and identify computational tasks that can be parallelized, such as similarity calculation and search algorithm execution. Evaluate the potential performance benefits of parallel computing.

Identifying optimization opportunities by analyzing redundancy removal processes in existing tools such as ATM. Evaluating current similarity metrics and redundancy removal strategies, especially their performance on large test sets.

**Specification:**

Functional: Provides parallel computing capabilities, supports multi-core processors and distributed computing resources. Implement task allocation and result merging mechanism to ensure the correctness and efficiency of parallel computing. Security-Feature-Based Selection: Prioritize test cases that represent diverse vulnerabilities to maximize coverage while minimizing redundancy.

Nonfunctional: The execution time is reduced to less than 1 hour on multicore processors. Tools should support multiple computing resources, and have ability to handle large test suites.

**Validation:**

Develop a prototype tool to support parallel computing, redundancy removal and verify its performance improvement in multi-core processors and distributed computing environments. Besides, we also need to verify coverage equivalence between minimized and original test sets. Collect feedback from test engineers and developers on the prototype tool to ensure that the performance improvement of the tool meets the actual needs.

**Negotiation:**

Negotiate with the development team about the priorities of the parallelized computing functions and the scope of the parallelized computing functions. Determine the human, time, and technical resources required by the development team during the development of parallelized computing functions. Aligning with developers on priorities, scope, and resource allocation. Reaching consensus on priorities, technical requirements, and resource allocation for successful implementation.

**References**

[1] Izadi M, Katzy J, Van Dam T, et al. "Language models for code completion: A practical evaluation", Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 2024: 1-13.

[2] Lu S, Guo D, Ren S, et al. "Codexglue: A machine learning benchmark dataset for code understanding and generation" , arXiv preprint arXiv:2102.04664, 2021.

[3] [Junqiang Li](https://dl.acm.org/doi/abs/10.1145/3597503.3623315" \o "Junqiang Li), [Senyi Li](https://dl.acm.org/doi/abs/10.1145/3597503.3623315" \o "Senyi Li), [Keyao Li](https://dl.acm.org/doi/abs/10.1145/3597503.3623315" \o "Keyao Li), et al. "ECFuzz: Effective Configuration Fuzzing for Large-Scale Systems." [Proceedings of the 46th IEEE/ACM International Conference on Software Engineering](https://dl.acm.org/doi/proceedings/10.1145/3597503).

[4] [Mingyang Geng](https://arxiv.org/search/cs?searchtype=author&query=Geng,+M), [Shangwen Wang](https://arxiv.org/search/cs?searchtype=author&query=Wang,+S), et al. "Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning", the 46th International Conference on Software Engineering (ICSE 2024).

[5] [Toufique Ahmed](https://arxiv.org/search/cs?searchtype=author&query=Ahmed,+T), [Kunal Suresh Pai](https://arxiv.org/search/cs?searchtype=author&query=Pai,+K+S), et al. "Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization)", International Conference on Software Engineering (ICSE-2024).

[6] [Minaoar Hossain Tanzil](https://dl.acm.org/doi/10.1145/3597503.3639194" \o "Minaoar Hossain Tanzil), [Junaed Younus Khan](https://dl.acm.org/doi/10.1145/3597503.3639194" \o "Junaed Younus Khan), et al. "ChatGPT Incorrectness Detection in Software Reviews", [Proceedings of the IEEE/ACM 46th International Conference on Software Engineering](https://dl.acm.org/doi/proceedings/10.1145/3597503).

[7] [Patanamon Thongtanunam](https://ieeexplore.ieee.org/author/37085394302), [Chanathip Pornprasit](https://ieeexplore.ieee.org/author/37088457315), et al. "AutoTransform: Automated Code Transformation to Support Modern Code Review Process" [2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)](https://ieeexplore.ieee.org/xpl/conhome/9793835/proceeding).

[8] [Yuancheng Jiang](https://dl.acm.org/doi/10.1145/3597503.3623307" \o "Yuancheng Jiang), [Jiahao Liu](https://dl.acm.org/doi/10.1145/3597503.3623307" \o "Jiahao Liu), et al. "Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation", [roceedings of the 46th IEEE/ACM International Conference on Software Engineering](https://dl.acm.org/doi/proceedings/10.1145/3597503).

[9] Shui He, Cai Fu, et al. "MalwareTotal: Multi-Faceted and Sequence-Aware Bypass Tactics Against Static Malware Detection", the 46th International Conference on Software Engineering (ICSE 2024).

[10] Rongqi Pan, Taher A, et al. "ATM: Black-box Test Case Minimization based on Test Code Similarity and Evolutionary Search", the 45th International Conference on Software Engineering (ICSE 2023).

[11] Chaleshtari N B, Marquer Y, Pastore F, et al. "AIM: Automated Input Set Minimization for Metamorphic Security Testing", IEEE Transactions on Software Engineering, 2024.