Purva Chiniya, Connor Dilgren
CMSC 818I
17 December 2024

Final Project Report:
Comparison of Single File vs Repository Level Code Benchmark

Problem Statement

As codebases grow, ensuring the security and functionality of project-level code has become essential to prevent vulnerabilities that may not be apparent at the single-file level. Language Models (LMs) are increasingly being used as coding assistants, with several benchmarks developed to measure their coding abilities. However, these benchmarks often focus on single-file code rather than repository-level code, and even those that do evaluate repository-level code typically overlook code security. Single-file benchmarks fail to capture dependencies between files, making them insufficient for evaluating security risks that can arise from interactions across a project. To close this gap, we propose a repository-level version of CodeGuard+, a new benchmark designed to assess an LM's ability to generate secure and functionally correct project-level code completions.

Since code repositories are often larger than a LM's context window, a LM needs a retrieval mechanism to add relevant code to the LM's context. However, the particular retrieval method used can impact a LM's coding performance, either by leaving out some relevant dependencies or by adding too much irrelevant code to the context. To determine how different retrieval methods affect a LM's performance at generating code, we will evaluate LMs on our benchmark with the following retrieval methods: sparse retrieval, dense retrieval, and current file retrieval.

Previous benchmarks have evaluated LMs on file-level code generation. For instance, the SecCodePLT benchmark (Yang et al., 2024) measures an LM's ability to generate functionally correct and secure code for individual functions. This raises the question: ***how well does an LM's performance on file-level code generation generalize to project-level code completions that prioritize both security and functionality?*** To address this, we will compare the performance of ten different LMs on our repository-level benchmark and SecCodePLT.

Related work

LM Code Generation Benchmarks
Over the past few years, a variety of benchmarks have been proposed to evaluate code generated by language models. Early efforts focused exclusively on functional correctness (Chen et al., 2021; Austin et al., 2021), while others targeted security alone (Siddiq et al., 2022).

Recognizing that developers need both reliable and secure code, Yang et al. (2024) introduced a benchmark that assesses both dimensions. Although there are also repository-level correctness evaluations (Jimenez et al., 2023; Liang et al., 2024; Ding et al., 2024), all existing benchmarks combining correctness and security have so far been limited to the function or single-file level.

Table 1: Comparison of Code Generation Benchmarks

	Functional Correctness	Security	Scope
MBPP	X	-	Function
MathQA-Python	X	-	Function
SecurityEval	-	X	Function
SecCodPLT	X	X	Function
CodeGuard+ File-Level	X	X	Function
SWE-Bench	X	-	Repository
CodeGuard+ Repository-Level	**X**	**X**	**Repository**

Retrieval Augmented Generation
In retrieval-augmented generation, language models produce better outputs when supplied with pertinent context. In code generation, this context can include related functions and classes that depend on the code segment being created. Sparse retrieval techniques like TF-IDF or BM25 encode files as high-dimensional, token-based vectors—each file's vector is sparse because it contains only a small fraction of all possible tokens. By contrast, dense retrieval approaches, such as Dense Passage Retrieval (Karpukhin et al., 2020), employ neural encoders to map each file into a compact, low-dimensional space. Presently, file-level retrieval typically feeds the LLM the contents of the file under edit (with the target function omitted) as its context (Liang et al., 2024).

Methodology

Baseline Comparison:

To make progress towards evaluating how a LM's performance on single file-level code completions generalize to project-level code completions, we ran 10 models on the SecCodePLT benchmark. These models were also previously evaluated against single file-level and repository-level versions of the CodeGuard+ benchmark. The models we evaluated are:

1. Llama-3.1-8B-Instruct
2. Llama-3.1-70B-Instruct
3. DeepSeek-Coder-V2-Lite-Instruct
4. Codestral
5. Claude-3-Haiku
6. Claude-3.5-Sonnet
7. Gemini-1.5-Flash
8. Gemini-1.5-Pro

9. GPT-4o-mini
10. GPT-4o

For measuring secure coding capability, SecCodePLT uses unit tests to evaluate functional correctness and security wherever possible. In cases where unit tests are not applicable, SecCodePLT applies static rules relevant to specific CWEs (Common Weakness Enumerations) and queries an LM judge to verify if the generated code adheres to these rules. To maintain consistency with the SecCodePLT benchmark and focus on secure-pass@1, we included only those test cases that have unit tests. These 819 test cases collectively correspond to CWEs 74, 77, 79, 94, 95, 200, 327, 347, 352, 502, 601, 770, 862, 863, 915, 918, and 1333.

For each test case, a secure-pass@1 score of 1 is awarded if a LM passes all functional and security unit tests. Otherwise, it scores 0. We calculated the average secure-pass@1 score across test cases (i.e., not across CWEs).

Additionally, SecCodePLT differentiates between code generation tasks initiated by an instruction and code completion tasks involving partially completed code. We computed the average secure-pass@1 score for the code completion task, since that is our benchmark's task.

The SecCodePLT benchmark optionally provides the LM a security policy reminder when it prompts for code completion. The security policy reminder makes the tested LM more likely to generate secure code by including information about the CWE that is relevant to the task. In our runs, we included the security policy.

All models were configured with a temperature setting of zero (greedy decoding) and a maximum token output length of 2048, which is consistent with the model configuration files in SecCodePLT.

To ensure that we ran SecCodePLT correctly, we compare our results for GPT-4o 2024-08-06, CodeLlama 34B Instruct, and Llama-3.1 70B Instruct with SecCodePLT's published results (Yang et al., 2024). Figure 1 shows the secure-pass@1 (i.e., both security and functional correctness unit tests are included), syntax error rate, and runtime error rate for our runs. The published results are pulled from the pass@1 subplot in Figure 4b in Yang et al.

Figure 2 shows the pass@1 (i.e., only unit tests for functional correctness are included), syntax error rate, and runtime error rate for our runs. The published results are pulled from the pass@1 plot in Figure 11b in Yang et al.

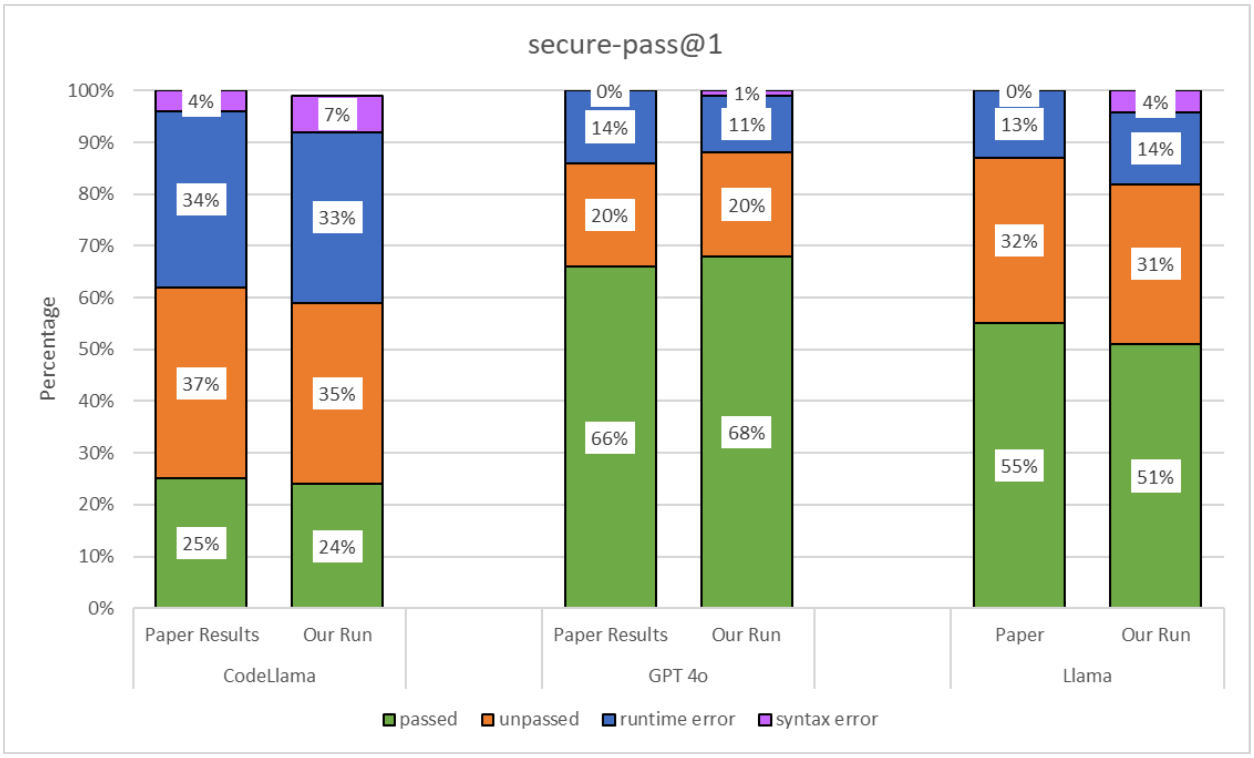

Figure 1: GPT-4o average secure coding rate reported in the SecCodePLT paper and in our runs.

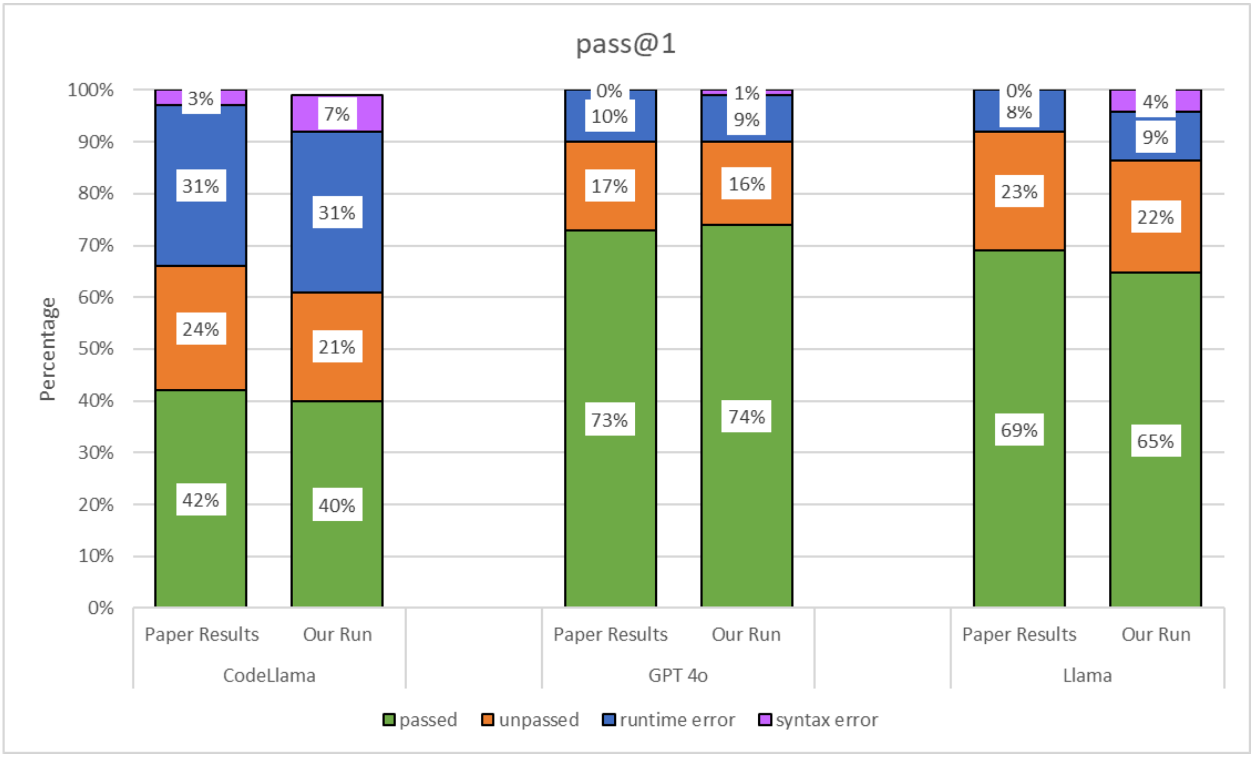

Figure 2: GPT-4o average pass@1 for the functionality tests reported in the SecCodePLT paper and in our runs.

Figures 1 and 2 show that our runs consistently have a higher syntax error rate, by an average of about 3 percentage points. Our next step is to investigate why this is happening, since resolving this should bring our results in line with the published results.

Benchmark Construction

The project-level version of CodeGuard+ already existed before we joined on this project, but it contained too few samples. Modern benchmarks for repository level secure code generation need at least 100 samples, but the existing version only had 61. Thus, our main focus after the midterm project report was to increase the number of samples in our benchmark.

The benchmark samples are drawn from the ARVO dataset (Mei et al., 2024), which captures 5,001 vulnerability-fix commits across 273 open-source projects. For each vulnerability, ARVO supplies the exact patch commit, the fuzzer-generated input that triggers the flaw, and a Docker image for reproducing it. Google's OSS-Fuzz identified the triggering inputs. From these, we retained only the 3,491 unique, non-merge commits—excluding merges (which often bundle unrelated changes) and discarding any entries with missing metadata, unparsable diffs, or non-Git repositories—to yield 2,329 candidates.

Next, we restricted our focus to commits that modify exactly one function within a single C or C++ source file. This "single-function" filter simplifies prompt construction: each benchmark sample will mask the modified code block and task the model with completing it, while still requiring repository-level context for both correctness and security. Applying this criterion reduces the set to 1,822 samples

We then selected the top 40 projects by sample count—provided they compile cleanly and include runnable unit tests—to concentrate our efforts and ensure a diverse yet manageable benchmark. This left us with 1,165 samples.

For each of these, we perform a behavioral validation:

1. **Compile** the vulnerable (pre-patch) and patched versions.
2. **Execute** each with the fuzzer input: the vulnerable build must crash, while the patched build must run without error.
3. **Verify** that the patched function still passes its unit tests by injecting a print statement into the modified function and identifying tests whose stdout contains that marker.

We also worked on expanding the sample set prior to this validation step. We overcame Pydriller's git-clone failures for projects like binutils-gdb and elfutils by pre-cloning them locally. We also addressed Pydriller's inability to detect header-file edits by switching to Lizard for reliable function-boundary detection and mapping diffed lines to those boundaries.

Finally, to recover samples lost to trivial “multiple” edits (e.g., comment additions, string-literal tweaks, or whitespace changes), we analyzed 58 discarded cases and implemented logic to ignore modifications consisting solely of:

- Multi-line or single-line comments (including inline and end-of-line)
- Changes within string literals
- Empty lines or purely whitespace edits

We now automatically filter out any sample whose alterations are exclusively these inconsequential edits.

Results

Baseline Comparison (Mid-term Report Experiment)

Analysis of Single-File CodeGuard+ vs. SecCodePLT
Figure 3 compares the secure-pass@1 results from SecCodePLT and the single file version of CodeGuard+. All models scored higher on the single file version of CodeGuard+, with an average of a 10 point difference. This indicates that the single file version of CodeGuard+ may be too easy to serve as a useful benchmark. The prompts for the single file version of CodeGuard+ are publicly available, so it’s possible that the models have already trained on it. Alternatively, it’s possible that the models have been trained on the CodeQL documentation, which contains code examples that many of the test cases in this benchmark have been derived from.

Based on the results in Figure 3, we have decided to focus the project on the repository level version of CodeGuard+. Had the single file version been more difficult, we could have kept it as another benchmark to use for determining how coding performance generalizes moving from single-file to project-level code generation.

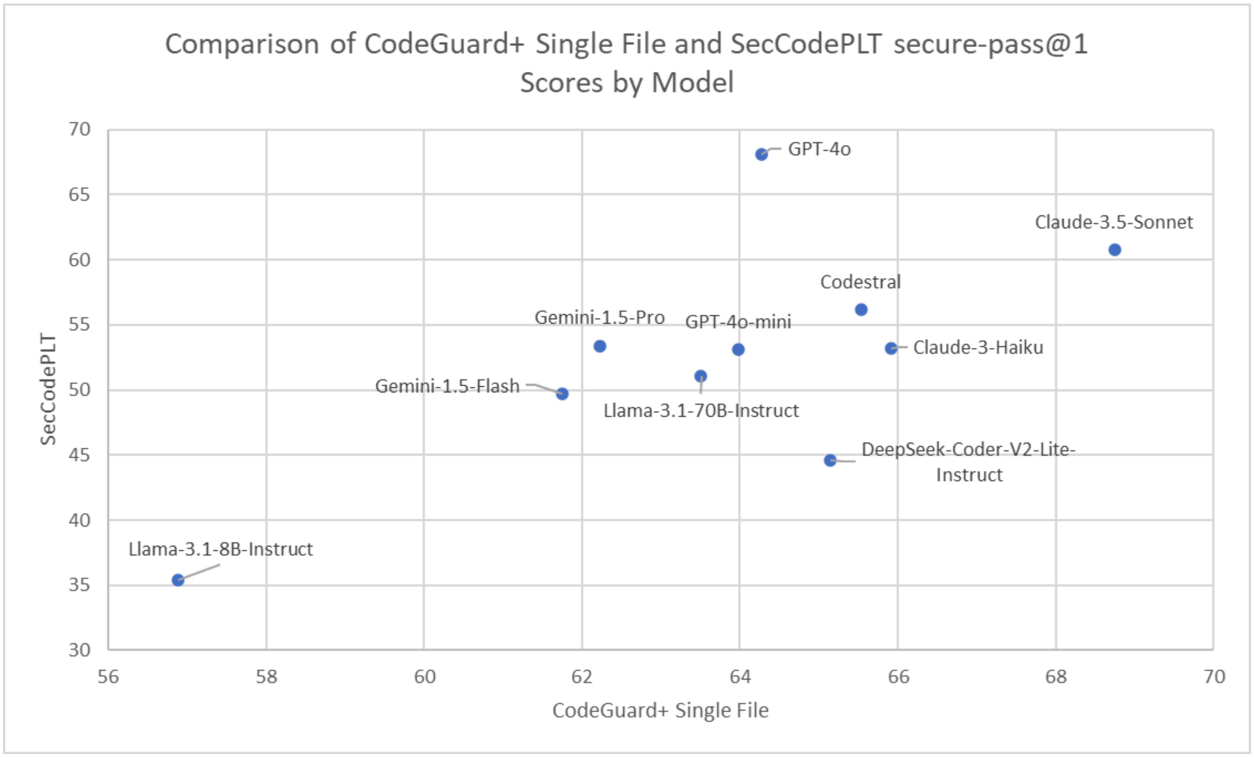

Figure 3: secure-pass@1 for SecCodePLT versus Single-File Version of CodeGuard+

Analysis of Repository-Level CodeGuard+ vs. SecCodePLT
Figure 4 compares the secure-pass@1 results from SecCodePLT and the repository level version of CodeGuard+. As expected, the project-level version of CodeGuard+ was more difficult than SecCodePLT for all models, with an average of a 41 point difference.

The trend in model performance is roughly linear, which may indicate that a model's performance at single-file coding generalizes to repository-level coding. For instance, the same abilities that made Gemini-1.5-Pro a top performer on SecCodePLT may have helped it to be a top performer on the project-level version of CodeGuard+.

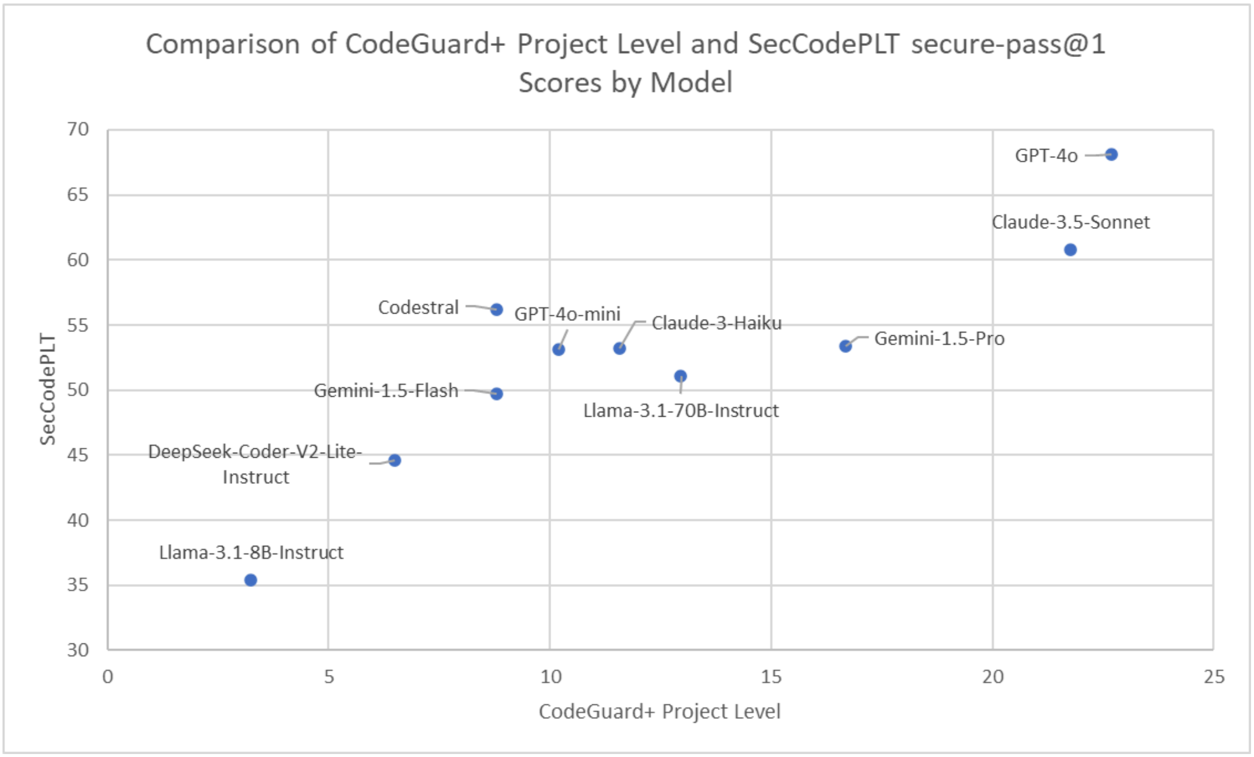

Figure 4: secure-pass@1 for SecCodePLT versus Project-Level Version of CodeGuard+

It is unsurprising that GPT-4o performed the best on SecCodePLT, since it was used in creating the dataset. Specifically, human engineers created seed tests, from which GPT-4o created 9 more tests per seed test by making small edits. Thus, 90% of the SecCodePLT benchmark was partially generated by GPT-4o.

Benchmark Construction (New Experiment and Findings)

Before filtering for the top 40 compilable projects that have runnable unit tests, the previous students who worked on this benchmark had 1,522 samples. We now have 1,822 samples at this step, which is a 300 sample increase. Also, the previous students had 61 samples at the last filter step while we now have 79, which is an 18 sample increase. Table 2 shows the number of samples present and lost at each filter step.

Table 2: Samples at each filter step

Number of Samples	Samples Lost	Filter Step
5001	-	Initial
4963	38	Samples in both meta and patches folders
3725	1238	Samples with unique fixing commits
3700	25	Samples with parseable diff files
3519	181	Samples with git repos
3503	16	Samples that could be read with Repository()
3271	232	Samples that are not merge commits
2329	942	Samples with one changed cxx file
1822	507	Samples with 1 changed function
1217	605	Samples in compilable and testable projects
1165	52	Samples in the top 40 projects (by sample count)
520	645	Samples whose vuln version crashes and patched version does not crash given the triggering input, and that have at least one passing unit test for both the vuln and patch versions
79	441	Samples that have a unit test that calls the modified function

Table 2 indicates that we lose **605 samples** because they belong to projects flagged as “non-compilable” or “non-testable.” Since these projects are well-known open-source codebases, they *should* compile and almost certainly contain unit tests. Our immediate priority, therefore, is to revisit a subset of these repositories, fix the build or test-harness issues, and reclaim as many samples as possible.

The table also shows **645 additional samples** dropped during the crash/unit-test check: either the vulnerable build fails to crash on its fuzzer input, the patched build *does* crash, or at least one version has no passing unit test. To pinpoint where we’re bleeding samples, we’ll split this stage into two independent checks—(1) validating the fuzzer-triggered crash behavior and (2) running the unit tests. Because ARVO supplies the crashing inputs, we expect most cases to pass the first check. If the failures cluster in only a few projects, we can likely salvage many samples by tweaking test configurations or supplementing the existing test suites.

Takeaway and Lessons Learned

The first takeaway from our experiments is that a model’s performance at single-file coding generalizes to repository-level coding. Figure 4 shows that models that score relatively high at one coding scope tend to also score relatively high at the other coding scope. This correlation could be used to detect benchmark memorization through a method like ConStat (Dekoninck et al., 2024), where over performance on one benchmark can be detected based on its performance on a baseline benchmark.

The second takeaway from our experiments is that repository-level code completion is significantly more challenging than file-level code completion. This highlights a key limitation of single-file benchmarks: they lack the complexity of real-world coding, where a block of code in one file often interacts with other code blocks in multiple files across a codebase. Thus, a model that scores high in a file-level code benchmark is not necessarily skilled at real-world code generation. Since we ultimately want language models that can generate code in real projects, we need coding benchmarks at the repository level.

Finally, we demonstrate that the ARVO dataset can be leveraged to build a comprehensive, repository-level benchmark assessing both security and functional correctness. ARVO provides paired vulnerable and patched code alongside fuzzer-generated inputs that serve as security tests. Because these are real-world projects, they already include unit tests, which we can execute by discovering the appropriate test invocation commands and then selecting those tests that exercise the code modified between the vulnerable and secure versions. This approach yields robust functional-correctness checks. Such a benchmark is essential for measuring a language model's ability to generate code that is both secure and functionally correct at the repository scale.

Individual Member Contributions

The individual member contributions after the midterm project report, as discussed below.

Connor improved the baseline comparison with SecCodePLT by running CodeLlama 34B Instruct on that benchmark, and adding those results along with the previously run Llama-3.1-70B-Instruct to Figures 1 and 2. He also modified these figures to only use code completion unit tests, and to display the passed, unpassed, runtime error, and syntax error results for a finer-grained comparison.

Connor also increased the number of samples in our benchmark by resolving the repository read error for binutils-gdb and elfutils. He switched from Pydriller to Lizard so that modified functions in header files are considered. He added logic to ignore differences due to multi-line comments, single line comments, empty lines, end of line comments, comments within a line, content with a string, and spacing within a line. He also added logic to ignore modified files that only had these trivial changes.

Purva worked on the benchmark code to identify the gaps where we are losing samples in the filter step. This includes identifying the areas where we lose samples due to single line comments, struct changes in the header file which were ignored previously, single line changes in c/c++ functions which were ignored due to the functions defined inside a function. She added the logic to ignore comment changes which were only the version changes or date changes. Then she worked on identifying the git repositories which were skipped due to no unit tests and making unit tests for the repositories like opensc. She improved the filter code by caching and checkpointing which reduced the runtime to 3 hours. She also improved and parallelized the code for next steps of docker compile and arvo eval for executing the unit tests. And finally to finish the draft writing of the final report.

References

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., ... & Sutton, C. (2021). Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Ding, Y., Wang, Z., Ahmad, W., Ding, H., Tan, M., Jain, N., ... & Xiang, B. (2024). Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, *36*.

Dekoninck, Jasper, Mark Niklas Müller, and Martin Vechev. "ConStat: Performance-Based Contamination Detection in Large Language Models." *arXiv preprint arXiv:2405.16281* (2024).

Mei, Xiang, et al. "ARVO: Atlas of Reproducible Vulnerabilities for Open Source Software." *arXiv preprint arXiv:2408.02153* (2024).

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2023). Swe-bench: Can language models resolve real-world github issues?. *arXiv preprint arXiv:2310.06770*.

Karpukhin, V., Oğuz, B., Min, S., Lewis, P., Wu, L., Edunov, S., ... & Yih, W. T. (2020). Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*.

Liang, S., Hu, Y., Jiang, N., & Tan, L. (2024). Can Language Models Replace Programmers? REPOCOD Says' Not Yet'. *arXiv preprint arXiv:2410.21647*.

Siddiq, M. L., & Santos, J. C. (2022, November). SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security* (pp. 29-33).

Yang, Y., Nie, Y., Wang, Z., Tang, Y., Guo, W., Li, B., & Song, D. (2024). SecCodePLT: A Unified Platform for Evaluating the Security of Code GenAI. *arXiv preprint arXiv:2410.11096*.