Software Vulnerability and Functionality Assessment using LLMs

Rasmus Ingemann Tuffveson Jensen
rasmus.jensen@jpmorgan.com
JP Morgan AI Research
London, UK

Vali Tawosi
vali.tawosi@jpmorgan.com
JP Morgan AI Research
London, UK

Salwa Alamir
salwa.alamir@jpmorgan.com
JP Morgan AI Research
London, UK

ABSTRACT

While code review is central to the software development process, it can be tedious and expensive to carry out. In this paper, we investigate whether and how Large Language Models (LLMs) can aid with code reviews. Our investigation focuses on two tasks that we argue are fundamental to good reviews: (i) flagging code with security vulnerabilities and (ii) performing software functionality validation, i.e., ensuring that code meets its intended functionality. To test performance on both tasks, we use zero-shot and chain-of-thought prompting to obtain final "approve or reject" recommendations. As data, we employ seminal code generation datasets (HumanEval and MBPP) along with expert-written code snippets with security vulnerabilities from the Common Weakness Enumeration (CWE). Our experiments consider a mixture of three proprietary models from OpenAI and smaller open-source LLMs. We find that the former outperforms the latter by a large margin. Motivated by promising results, we finally ask our models to provide detailed descriptions of security vulnerabilities. Results show that 36.7% of LLM-generated descriptions can be associated with true CWE vulnerabilities.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Software development techniques**.

KEYWORDS

Software Security, Functional Validation, Large Language Models

ACM Reference Format:
Rasmus Ingemann Tuffveson Jensen, Vali Tawosi, and Salwa Alamir. 2024. Software Vulnerability and Functionality Assessment using LLMs. In *2024 ACM/IEEE International Workshop on NL-based Software Engineering (NLBSE '24), April 20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3643787.3648036

1 INTRODUCTION

Code review is the process whereby software developers analyze if peer contributions are of sufficient quality to be integrated into codebases. The practice reduces bugs, increases code quality, and facilitates knowledge transfer [12]. Code reviews can, however, be costly and tedious to carry out [5]. Furthermore, poorly performed reviews may foster a negative work environment, provide a false sense of security, and hinder innovation [16]. In an effort to address such problems, previous studies have proposed methods to automate code reviews, so far, though, with modest success [21].

Large language models (LLMs) have recently demonstrated remarkable performance on a variety of tasks, including code generation and question answering [3]. In this paper, we investigate whether and how LLMs can aid with code reviews. As models, we consider a mixture of open-source and proprietary LLMs, including models from the Dolly, Falcon, Llama, and GPT families. Acknowledging that reviews contribute to many things, e.g., knowledge sharing and maintainability, our experiments focus on two tasks: (i) flagging code with security vulnerabilities and (ii) performing software functional validation, i.e., ensuring that code meets its intended functionality.[1] The tasks motivate our research questions:

RQ1. Can LLMs flag code security vulnerabilities?
RQ2. Can LLMs do software functional validation?
RQ3. Can LLMs simultaneously flag security vulnerabilities and do software functional validation?
RQ4. Can LLMs provide feedback on security vulnerabilities?

2 RELATED WORK

Several studies have investigated how machine learning and natural language processing can support code reviews. The field is related to defect prediction, i.e., predicting if a code snippet contains a bug. Studies vary substantially in their approach and the code granularity being analyzed. They may, e.g., consider stand-alone functions, source code files, or file changes (i.e., "diffs" in pull requests).

Li *et al.* [8] consider classification of triplets made up of change descriptions and snippets of old and new code. The aim is to predict if a triplet is accepted by reviewers. The authors propose a deep learning model utilizing word2vec embeddings and convolution layers, reporting F1 scores from 0.44 to 0.50 on data from five software projects. Shi *et al.* [17] consider pairs of original and changed source files. Aiming to predict if changes are approved by a reviewer, the authors propose a deep learning model with convolution and LSTM layers. Using data from six projects, the authors report F1 scores ranging from 0.40 to 0.57. Kim *et al.* [7] aim to classify code changes from 12 projects as "buggy" or "clean." The authors use SVM on features that include bag-of-words metrics. Reporting 78% accuracy, the authors note that their data may contain an inflated number of bugs. Lu *et al.* [10] propose Llama-Reviewer, a framework to fine-tune LLMs for code reviews in a parameter-efficient manner. Considering code changes, the authors, in particular, report an F1 score around 0.70 trying to predict the necessity of code reviews.

[1] We do not aim to generate unit tests (which should be applied before a code review). Rather, we aim to determine if a code snippet meets its intended functionality without executing it.

NLBSE '24, April 20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0576-2/24/04…$15.00
https://doi.org/10.1145/3643787.3648036

Our paper adds to the existing literature by (i) investigating how LLMs may be used to flag (and describe) security vulnerabilities, not (just) logical errors. Secondly, we explore how LLMs may be used to determine if a piece of code meets its intended functionality without executing it.

3 METHODOLOGY

In this section, we outline our employed datasets, the LLMs we used for experimentation, and our experimental setup.

3.1 Datasets

Our experiments utilize three datasets with Python code snippets at the function level. For all datasets, we only consider code snippets that contain exactly one function for consistency. Furthermore, we only consider observations[2] that contain (i) exactly one docstring (from which we extract each code snippet's intended functionality) or (ii) at most one docstring but a clearly stated programming task in a separate column of the original dataset. Our datasets are:

1. **HumanEval** [3], from which we employ 148 observations. The original dataset contains 164 observations. However, we discard 16 observations that do not contain exactly one function and exactly one docstring.
2. **MBPP** [2], from which we employ 476 observations. The original dataset contains 974 observations. However, the authors only designate 500 observations for testing. Furthermore, we discard 24 observations that do not contain exactly one function or contain more than one docstring.
3. **SecurityEval** [18], from which we employ 36 observations with security vulnerabilities per the CWE system [11]. The original dataset contains 121 observations. However, we discard (i) 78 web-scraped observations (to avoid data leakage as our LLMs might have seen them during training), (ii) 6 observations that do not contain exactly one function and exactly one docstring, and (iii) a single observation associated with CWE-730, a CWE code actively discouraged from use.

None of our employed observations have, to the best of our knowledge, been seen by our LLMs during training. Furthermore, we only consider author-written observations from SecurityEval (published in late 2022). In relation to security vulnerabilities, we denote observations from SecurityEval as "dirty" and observations from HumanEval and MBPP as "clean." While our data only contains around 5% dirty observations, empirical studies, notably, suggest that security vulnerabilities are rare in real codebases [14].

3.2 Models

Our experiments consider nine LLMs; six open-source and three proprietary. The open-source models were chosen from the Hugging Face Open LLM Leaderboard in August 2023, aiming to represent influential models. Based on resource and hardware constraints, we only consider smaller versions of the models. Unless explicitly stated, all hyper-parameters are kept at default values (similar to Hugging Face deployments). The proprietary models are employed through APIs as of April/May 2023 (before OpenAI released updated model versions in June 2023).

[2]We use the terms "code snippet" and "observation" interchangeably.

> Review the provided code for potential security vulnerabilities. After your review, assign the code a label from [Yes, No]. 'Yes' implies that there are security vulnerabilities. 'No' implies that there are no security vulnerabilities. Return only the label.
> **Code:** [code_snippet]
> **Label:**

Figure 1: RQ1 Zero-shot prompt to flag security vulnerabilities.

Our open-source models are: Falcon-7b-instruct, Llama-2-7b-chat, Llama-2-13b-chat, Dolly-v2-3b, Dolly-v2-7b, and Dolly-v2-12b [1, 4, 20]. Our proprietary models are: Text-davinci-003, GPT-3.5-turbo, and GPT-4 [13].

3.3 Experimental Setup

For all experiments, we report model accuracy. Our data is highly imbalanced in terms of security vulnerabilities, motivating us to also report F1 scores. Previous studies have shown that LLM outputs can vary substantially given small input changes [6]. For robustness, we therefore run all experiments (i.e., prompts) 10 times, perturbing code snippets between runs. For each code snippet, we randomly apply one of the following transformations: splitting the longest line, replacing tabs with spaces, replacing frequent variable names with "xxxx", converting between CamelCase and snake_case, and doing nothing. The transformations are heavily inspired by [22], designed to change only the syntax, not functionality, of code snippets. For RQ1 through RQ3, we limit LLM answers to 8 new tokens. For RQ4, we limit answers to 100 new tokens.[3] Below, we address each of our research questions in turn, describing our experimental setup for each.

RQ1. We employ zero-shot, binary classification to answer RQ1. As positive observations, we use dirty code snippets (i.e., observations from SecurityEval). As negative observations, we use clean code snippets (i.e., observations from MBPP and HumanEval). To make predictions, we use the prompt in Figure 1. An answer (cast to be lowercase) that contains the word "yes" is coded as a positive prediction; otherwise, it is coded as a negative prediction.

RQ2. We employ zero-shot, binary classification to answer RQ2. To this end, we first extract the programming tasks associated with all code snippets. For MBPP, tasks are given in a column in the dataset. For HumanEval and SecurityEval, tasks are given in each code snippet's docstring. To construct positive observations, we pair each code snippet with its correct task description. To construct negative observations, we pair each code snippet with a "wrong but similar" description. To this end. we embed all descriptions with OpenAI's Ada-002 model and pair every code snippet with the nearest neighbour to its correct task description (treating each dataset independently). The idea is to make our classification problem as difficult as possible. Our setup doubles the number of observations in every dataset, making our problem balanced by construction. To classify observations, we use the zero-shot prompt in Figure 2. Predictions are coded as in RQ1.

[3]Limiting tokens allows faster inference. However, we must allow a sufficient number of tokens to ensure that our models can produce responses as desired.

Review the provided code and verify that it meets its intended functionality. After your review, assign the code a label from [Yes, No]. 'Yes' implies that the code meets its intended functionality. 'No' implies that the code does not meet its intended functionality. Return only the label.
Code: [code_snippet]
Intended functionality: [task_description]
Label:

Figure 2: RQ2 Zero-shot prompt for software functional validation.

RQ3. We consider both zero-shot and chain-of-thought prompting to answer RQ3, aiming to produce final "approve/reject" recommendations. We employ the same experimental data as in RQ2 (i.e., datasets with correct and wrong task descriptions). For the zero-shot approach, we use the prompt in Figure 3 without the bold italicised text. For the chain-of-thought approach, we use the prompt in Figure 3 with the bold italicised text, utilizing answers obtained by the prompts in Figures 1 and 2. Our setup runs all prompts in sequence, not mixing results between runs. For HumanEval and MBPP, half of our observations are positive, being clean and associated with correct task descriptions, while the other half are negative, being associated with wrong task descriptions even though they are clean. For SecurityEval, all observations are negative, being dirty regardless of their task descriptions. An LLM answer that contains the word "approve" is coded as a positive prediction (and otherwise as negative).

RQ4. To investigate if our LLMs can provide specific feedback on security vulnerabilities, we employ what may be described as multi-class, zero-shot classification. First, we ask our LLMs to describe security vulnerabilities in our 36 code snippets from SecurityEval. Our prompt is given in Figure 4. To evaluate the quality of the generated descriptions, we (i) embed them using openAI's Ada-002 model, (ii) embed 958 true vulnerability names from the CWE system, and (iii) measure how similar each LLM-generated description is to all true vulnerability names using cosine similarity.
We consider an LLM-generated description to be good if its closest CWE name corresponds to (i) the actual vulnerability in an underlying code snippet or (ii) an immediate relation (e.g., parent, child, or peer) to the former. We note that the 958 CWE vulnerabilities from which we consider true vulnerability names have 3.09 relations on average. The vulnerabilities in our employed data from SecurityEval have 6.39 relations on average.

4 RESULTS

Our results are displayed in Table 1. Note that the table displays results over our combined dataset.

For RQ1, Text-davinci-003 performs best with an accuracy of 95.6% and F1 score of 37.9%.[4] All open-source models perform poorly with F1 scores close to (and even below) what one would expect from a predictor assigning the same label to any observation.

For RQ2, our datasets are balanced by construction. Here, all open-source models consistently achieve accuracy scores around 50%, similar to predictors assigning the same label to all observations. Among the GPT-models, GPT-4 performs best with an accuracy of 88.7% and F1 score of 88.2%.

[4]One might be surprised that Text-davinci-003 performs better than GPT-3.5-turbo and GPT-4. We speculate that this might be due to the models' alignment.

Review the provided code for potential security vulnerabilities and verify that it meets its intended functionality.
When asked if the code contains a security vulnerability, a previous review yielded: [answer_from_RQ1].
When asked if the code meets its intended functionality, a previous review yielded: [answer_from_RQ2].
After your review, assign the code a label from [Approve, Reject]. 'Approve' implies that there are no security vulnerabilities and that the code meets its intended functionality. 'Reject' implies that there are security vulnerabilities or that the code does not meet its intended functionality. Return only the label.
Code: [code_snippet]
Intended functionality: [task_description]
Label:

Figure 3: RQ3 Zero-Shot and Chain-of-thought prompt to obtain final "approve or reject" recommendations.

Review the provided code for potential security vulnerabilities. Provide a brief description of any vulnerability you find. Return only the description.
Code: [code_snippet]
Description:

Figure 4: RQ4 Zero-shot prompt used to obtain descriptions of security vulnerabilities.

For RQ3, the open-source models perform poorly regardless of prompting technique. In the proprietary models, we observe substantial performance increases when switching to chain-of-thought prompting. Thus, chain-of-thought prompting appears to increase performances for the proprietary models. Our best-performing model, GPT-4, specifically, sees an increase in its accuracy from 80.8% to 87.2% and an increase in F1 from 76.6% to 85.7%. Running the same experiments without code perturbations, we obtained similar results, indicating robustness to the perturbations.

For RQ4, the proprietary models appear to outperform the open-source models. Our best-performing proprietary model, GPT-4, in particular, generates vulnerability descriptions that on average can be associated with true CWE names 36.7% of the time. Considering each model family (GPT, Llama, and Dolly), we note that model performances generally appear to increase with model size.

5 THREATS TO VALIDITY

Regarding internal validity, we note that our setup for RQ2 implicitly assumes that all programming tasks in each dataset are unique, i.e., attempt to obtain different things. While we believe this to be the case for HumanEval, it is not given for MBPP and SecurityEval. We also note how LLM answers (i) are stochastic in nature and (ii) can vary given small input changes. To mitigate both effects, we perform perturbations and multiple experimental runs.

Regarding external validity, we stress that our experiments only consider smaller versions of open-source models and code snippets with single functions. Thus, our conclusions do not extend to larger open-source models or source code files; being possible directions for future work. In future work, our experiments may also

Table 1: Performance on RQ1-RQ4. Metrics averaged over 10 runs. Standard deviations in parentheses.

LLM	RQ1		RQ2		RQ3 (Zero-shot)		RQ3 (Chain-of-thought)		RQ4
	Accuracy	F1 (Dirty)	Accuracy	F1 (True Task)	Accuracy	F1 (Approve)	Accuracy	F1 (Approve)	Match Rate
GPT-4	74.7 (1.0)	29.3 (0.9)	**88.7 (0.3)**	**88.2 (0.4)**	**80.8 (0.5)**	**76.6 (0.7)**	**87.2 (0.2)**	**85.7 (0.3)**	**36.7 (3.4)**
GPT-3.5-turbo	82.2 (1.0)	16.2 (3.8)	57.9 (0.5)	70.1 (0.3)	50.5 (0.5)	65.5 (0.2)	55.8 (0.7)	67.4 (0.4)	20.3 (4.9)
Text-davinci-003	**95.6 (0.1)**	**37.9 (3.0)**	82.9 (0.4)	84.4 (0.3)	68.5 (0.3)	74.7 (0.2)	80.6 (0.4)	81.6 (0.3)	24.4 (6.1)
Llama-2-13b-chat-hf	50.6 (2.3)	9.3 (1.2)	51.6 (1.2)	53.8 (1.5)	48.1 (0.9)	53.8 (1.1)	50.3 (1.6)	50.2 (1.5)	19.2 (4.8)
Dolly-v2-12b	54.9 (2.9)	8.9 (1.8)	49.2 (0.6)	59.0 (0.5)	51.3 (1.5)	39.8 (1.6)	48.3 (1.2)	60.4 (0.8)	19.2 (3.1)
Falcon-7b-instruct	42.0 (1.1)	9.8 (0.9)	50.3 (0.5)	65.3 (0.5)	47.7 (0.6)	61.5 (0.5)	49.5 (1.6)	55.2 (1.2)	19.4 (7.0)
Dolly-v2-7b	9.7 (0.7)	10.0 (0.3)	50.0 (0.9)	64.6 (0.7)	47.2 (0.6)	62.0 (0.5)	47.7 (0.7)	62.5 (0.4)	16.7 (6.0)
Llama-2-7b-chat-hf	63.0 (1.3)	10.9 (2.1)	49.6 (1.4)	51.4 (1.2)	50.7 (1.4)	57.5 (1.4)	51.3 (1.7)	46.8 (2.1)	17.2 (6.6)
Dolly-v2-3b	22.5 (1.6)	10.0 (1.0)	50.2 (0.7)	63.3 (0.6)	48.1 (1.1)	53.7 (1.1)	48.0 (1.0)	57.2 (1.0)	14.7 (4.3)

be extended to consider specialized open-source models (e.g., Code Llama [15]), larger datasets (e.g., EvalPlus [9]), few-shot prompting (see, e.g., [19]), and make comparisons with methods using unit tests or static code analyzers.

6 CONCLUSION AND DISCUSSION

We have developed an experimental framework to investigate how LLMs can aid in code reviews. Our results show that smaller open-source models generally perform on par with random or dummy classifiers. However, when used to flag security vulnerabilities, the best proprietary model achieves an accuracy of over 95.6% and an F1 score over 37.9%. When used to perform software functionality validation, the model achieves an accuracy and F1 score over 88.2%. Furthermore, vulnerability descriptions from the model can be matched to true vulnerabilities over 36.7% of the time.

DISCLAIMER

This paper was prepared for informational purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

ACKNOWLEDGMENTS

We are grateful to Ran Zmigrod for discussions and feedback.

REFERENCES

[1] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. Falcon-40B: An open large language model with state-of-the-art performance. https://huggingface.co/tiiuae/falcon-7b

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. arXiv:2108.07732

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374

[4] Mike Conover, Matt Hayes, Ankit Mathur, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin. 2023. Free Dolly: Introducing the World's First Truly Open Instruction-Tuned LLM. www.databricks.com/blog/2023/04/12/dolly-first-open-commercially-viable-instruction-tuned-llm

[5] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. 2015. Code reviews do not find bugs. how the current code review best practice slows us down. In *37th International Conference on Software Engineering*, Vol. 2. IEEE, Florence, Italy, 27–28.

[6] Akshita Jha and Chandan K Reddy. 2023. Codeattack: Code-based adversarial attacks for pre-trained programming language models. arXiv:2206.00052

[7] Sunghun Kim, James Whitehead, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on software engineering* 34, 2 (2008), 181–196.

[8] Heng-Yi Li, Shu-Ting Shi, Ferdian Thung, Xuan Huo, Bowen Xu, Ming Li, and David Lo. 2019. DeepReview: Automatic Code Review Using Deep Multi-instance Learning. In *Advances in Knowledge Discovery and Data Mining*, Qiang Yang, Zhi-Hua Zhou, Zhiguo Gong, Min-Ling Zhang, and Sheng-Jun Huang (Eds.). Springer International Publishing, Cham, 318–330.

[9] Jiawei Liu, Chunqiu Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. https://openreview.net/forum?id=1qvx610Cu7

[10] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. In *IEEE 34th International Symposium on Software Reliability Engineering*. IEEE, Florence, Italy, 647–658.

[11] MITRE. 2023. Common Weakness Enumeration. https://cwe.mitre.org/

[12] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 2015. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, Montreal, Canada, 171–180.

[13] OpenAI. 2023. Models. https://platform.openai.com/docs/models

[14] Yulong Pei, Salwa Alamir, Rares Dolga, and Sameena Shah. 2023. Code Revert Prediction with Graph Neural Networks: A Case Study at J.P. Morgan Chase. In *1st International Workshop on Software Defect Datasets*. ACM, San Francisco, CA, USA, 1–5.

[15] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. arXiv:2308.12950

[16] Jaydeb Sarker, Asif Kamal Turzo, Ming Dong, and Amiangshu Bosu. 2023. Automated Identification of Toxic Code Reviews Using ToxiCR. arXiv:2202.13056

[17] Shu-Ting Shi, Ming Li, David Lo, Ferdian Thung, and Xuan Huo. 2019. Automatic code review by learning the revision of source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. AAAI, Honolulu, HI, USA, 4910–4917.

[18] Latif Sunny and Joanna Santos. 2023. SecurityEval. www.github.com/s2e-lab/SecurityEval

[19] Vali Tawosi, Salwa Alamir, and Xiaomo Liu. 2023. Search-Based Optimisation of LLM Learning Shots for Story Point Estimation. In *International Symposium on Search-Based Software Engineering*. Springer, San Francisco, CA, USA, 123–129.

[20] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288

[21] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. arXiv:2201.06850

[22] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2022. ReCode: Robustness Evaluation of Code Generation Models. arXiv:2212.10264