Code Companion: Using ML to Detect Vulnerabilities in Code

Andrew Bevington
Computer Science
Boston College
Class of 2025

Sara Madani
Math
Bocconi University
Class of 2025

Bryan O'Keefe
Computer Science
Boston College
Class of 2026

Alex Velsmid
Computer Science
Boston College
Class of 2026

February 1, 2025

Abstract

A big problem for every software engineer and tech company is ensuring their applications are secure. Without proper security, vulnerabilities in code can allow for hackers to gain unauthorized access to user data, private source code, etc. Exploits in code can negatively affect consumers and products, which is a big problem for companies. In the past, thousands of lines of code would have to be manually reviewed by professionals to find and remove these vulnerabilities. The goal of this project is to create a machine learning algorithm that automates this process by taking in a GitHub repository, scanning through the files, and outputting a risk score and weak points in the code. This would alleviate the pain of manual review, and help programmers create more secure programs.

1 Introduction

With the rapidly increasing pace of software development as a result of modern tools like generative AI, the need to ensure code security is becoming increasingly more valuable. Cybersecurity is important in modern society as unauthorized access to private records and data can have severe effects. In an effort to increase cybersecurity and code review efficiency, our goal is to create a machine learning model to accurately detect flaws in code. In order to do so, the model would take code files as input and output areas and behaviors that are vulnerable, their severity, and suggestions to improve. Currently, there are multiple issues present with adapting a machine learning model to cyber security efforts. These issues are noted in the "Pitfalls of Machine Learning for Computer Security" article where it states that "30 top-tier security papers from the past decade that rely on machine learning [. . . suffer] from at least three pitfalls; even worse, several pitfalls affect most of the papers" [1]. These problems include sampling bias – when training data does not accurately represent real-world data – to inappropriate threat models — where the assumed threats are not reflective of actual security threats. Therefore, an important area to focus on is training data that accurately reflect real-world issues as well as being meticulously separated between training and test data. Before developing a model, data will be taken from multiple databases in order to obtain holistic code that is fairly representative of the real-world. Then to tackle the model, our design process will follow steps of increasingly independent models beginning with an AI wrapper and progressively evolving into an independent code base.

1.1 Data

There are two sources of data which we have been exploring. The first of these is the DiverseVul dataset [2]. This dataset contains a large number of vulnerabilities written in C/C++. These have been split into two categories: those which contain vulnerabilities and those where vulnerabilities are not present. In the vulnerabilities present category, there are 281,000 files, and in the vulnerabilities not present category, there are around 50,000 examples of code. For each of these examples, there is a certain ammount of data. Each entry contains the code snippet itself (in most cases a large chunk of code), the commit message, and if there is one then the vulnerability code which is present. However, an issue that exists with this dataset is that there is a significant lack of human attention to the data and its integrity. This means that there are significant instances of code duplication and misidentification of vulnerabilities.

Another dataset that we have been exploring is the PrimeVul [3] dataset. This dataset is an improvement on the DiverseVul dataset in that it was created with more attention and human intervention. The dataset contains 6,968 vilnerable functions as well as 228,000 non- vulnerable functions. The data is also pre-split into training and testing sets. This ensures that there is no code duplication between the two sets. The PrimeVul dataset is the one that we will start our experimentation with and will serve as the main dataset for our project. If we find that we need more data, then we will migrate to the DiverseVul dataset. However, if this is deemed necessary, then we will need to take steps to ensure that the data is clean. This will likely involve removing duplicates and ensuring that the data is correctly labeled, all of which is possible with python scripts to reformat and refactor the data.

1.2 Behaviors

Behaviors in the general use case of this model we see it as having a fairly streamlined approach to scanning through code. In the correct usage, a user would upload a set of files to the model. From these files the model would then look through and determine where there are code vulnerabilities and their severity. In the best case we see the model being able to point out the line or lines of code where there is a code insecurity. With this identification would also come an explanation of why the code poses a risk, what the scale of that risk is, and some ways in which the user can fix the issue. This will be given to the user in the form of a text file or a chatbot-based system depending on how our experimentation progresses. It is easy to foresee errors in this ideal flow of information. The most common error which we are most likely to see would be an absence of identification of code vulnerabilities. If we feed it in code with vulnerabilities and the model fails to see these vulnerabilities then the model is not doing what it is supposed to be doing and not functioning in an optimal way.

With the usage of a datasource like PrimeVul or DiverseVul we are able to create a model which can identify individual lines of code as well as large sections of vulnerable code. With its training on entire Git commits it will be able to function in an environment which more closely resembles that which a user is likely to upload to the model. It will be able to parse through the code and identify behaviours which a line by line model would not be able to see. Also, with the addition of the NIST database we are able to web scrape the provided link in order to allow the model to return a human-readable response explaining what the vulnerability is, how it works in the code, and just how bad it could be if it is left to be unchecked.

2 Related Works

The main agents in the vulnerability workspace are large-scale models such as ChatGPT [4] which allows for general querying of code, as well as StarCoder2 [5], a LLM trained on a large amount of data related to coding specifically. The prior benchmark for model success was running them on the BigVul dataset. However, upon further research, it was found that this dataset was flawed and a poor representation of real world applications. A large issue was in code duplications as well as poor labeling of data. When trained on PrimeVul, a more refined and accurate dataset, StarCoder2 went from a 68.7% F1 score to a 3.03% F1 score [3]. It is found through experimentation that these large pre-trained models are ineffective at accurately finding and diagnosing malicious code in a real world environment and that novel approaches need to be made to allow for real world applications of this product. [3]

3 Pitfalls

This article notes some of the common struggles when applying machine learning to cybersecurity efforts. The paper analyzed 30 top cybersecurity papers that applied machine learning and found that all 30 had at least 3 of these pitfalls with many having several.

Pitfall 1: Data Snooping Bias

- Using test data during the training phase leads to artificially high accuracy and overestimates the model's real-world effectiveness.

Pitfall 2: Incorrect Baselines

- Comparing the ML model against weak or outdated baselines makes it appear more effective than it actually is.

Pitfall 3: Inappropriate Threat Model

- Assuming unrealistic attack scenarios that do not reflect actual cybersecurity threats, leading to ineffective models in real-world applications.

Pitfall 4: Disparate Data Distributions

- Training data does not match the distribution of real-world attack scenarios, causing poor generalization when deployed.

  Pitfall 5: Poor Feature Engineering

- Extracting the wrong features or failing to include key security-relevant information can lead to inaccurate ML models.

  Pitfall 6: Lack of Ground Truth

- Without clear and accurately labeled data, ML models may learn incorrect patterns, reducing their reliability.

  Pitfall 7: Temporal Bias

- Training on outdated data that does not reflect evolving cybersecurity threats results in models that become obsolete quickly.

  Pitfall 8: Adversary Adaptation

- Attackers can modify their tactics to bypass ML-based security systems, reducing the model's long-term effectiveness.

  Pitfall 9: Evaluation Methodology Flaws

- Inadequate testing setups, such as evaluating models in unrealistic environments, can produce misleading performance results.

  Pitfall 10: Misleading Performance Metrics

- Using inappropriate metrics (e.g., accuracy instead of precision-recall in imbalanced datasets) can give a false sense of security effectiveness.

4 Next Steps

The following is a general list of the steps we plan to take through the semester. This is a rough outline and is subject to change as we progress through the project.

1. Upload the PrimeVul dataset to a cloud-based storage system. This will clear storage on our local machines and allow for easier access to the data from a Colab environment.
2. Obtain an LLM API key and run initial tests on the dataset. Our goal is to undersntand the data as well as how accurate the existing models are at solving the task of detecting vulnerabilities.
3. Take an existing LLM and fine-tune it on the PrimeVul dataset. This will allow us to see how well the model can be adapted to our specific task while maintaining an existing and proven architecture.
4. Train a model from scratch on the PrimeVul dataset. This will allow us to see how architectural decisions can affect the performance of the model and if fundamental changes need to be made in order to achieve the desired results.

5 Contribution Timeline

01/22: Alex, Sara, Drew, Bryan First team meeting where project ideas were brainstormed. Google doc was created to store ideas and research
01/23 Drew, Bryan Team meeting with Prof. Bento to discuss project ideas. Code Vulnerability project was chosen
01/27 Alex, Sara, Drew, Bryan Team meeting to discuss strategy, proficiencies, and research approach

01/29 Alex Research on VulKG knowledge-graph. Wrote the abstract.
01/30 Drew Research on DiverseVul database. Wrote behaviours section
01/31 Alex, Drew Meeting with Professor Bento, discussing progress and databases
01/31 Alex Research on the PrimeVul database and wrote about it.
02/01 Bryan Research on current machine learning cybersecurity issues. Wrote introduction. Found and wrote up a summary about the OSV database
02/01 Drew Wrote the Related Works section

References

[1] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Pitfalls in machine learning for computer security," *Communications of the ACM*, vol. 67, pp. 92–101, Nov. 2024.

[2] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," *arXiv preprint*, Apr. 2023. arXiv:2304.00409.

[3] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, "Vulnerability detection with code language models: How far are we?," *arXiv preprint*, Mar. 2024. arXiv:2403.18624.

[4] OpenAI, "GPT-4 technical report," *Technical Report*, 2024.

[5] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint*, Feb. 2024. arXiv:2402.19173.