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

May 11, 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 and private source code. Exploits in code can negatively affect consumers and products, which is a detrimental to 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 and compare multiple machine learning algorithms that automate the process by taking in a function written in C and running a binary classification model on the function to label it as vulnerable or non-vulnerable. This would alleviate the pain of manual review, and help programmers create more secure applications.

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. According to IBM's Cost of a Data Breach Report 2024, the average global cost of a data breach is $4.88 million, a 10% increase from 2023 [9]. This highlights the critical importance of cybersecurity as more and more aspects of modern life become digitized and connected. Therefore, 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 and compare multiple machine learning models to accurately detect flaws in code. In order to do so, the model would take code files as input and output a binary digit – 0 signifying safe and 1 signifying vulnerable. Currently, there are multiple issues present with adapting a machine learning model to cyber security efforts. These issues are noted in [3], which examines numerous top-tier security papers from the past decade that rely on machine learning concepts and identifies three recurring problems. These problems include inappropriate threat models — where the assumed threats are not reflective of actual security risks – adversarial vulnerabilities – where malicious inputs can deceive machine learning models – and sampling bias – when training data does not accurately represent real-world data. During model training, the third issue, sampling bias, significantly affected performance: it degraded results when ignored but improved them when properly addressed. Before developing a model, data will be taken from multiple databases in order to obtain training data which is formatted and filtered to avoid duplications and inconsistencies in syntax and layout. After collecting the data, we conducted a series of experiments. We began with using an OpenAI API as a baseline model. From there, we explore a basic linear model used to create an Adaboost style gradient boosting network. Then, we transitioned to a graph-based neural network approach, for more advanced graph-based learning. Lastly, we concluded with a transformer model, leveraging its ability for complex sequence learning and attention mechanisms.

2 Related Works

The main agents in the vulnerability workspace are large-scale models such as ChatGPT [16] which allows for general querying of code, as well as StarCoder2 [14], a LLM trained on a large amount of data related to coding specifically. The prior benchmark for model success was calculated using 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 (evaluation methodology detailed in section **5.2**) to a 3.03% F1 score [5]. 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. [5]

Flawfinder is a naive approach to vulnerability detection. This application works by searching within a code file for known vulnerabilities, such as buffer overflows, format string problems, and more. This simple approach allows for quick identification of simple, surface-level bugs, however, cannot understand data types of function parameters, nor analyze control flow or data flow. To perform deeper analysis on vulnerabilities, machine learning techniques will be necessary.

AMPLE [26] is one of the state-of-the-art vulnerability detection models that uses graph neural networks (GNNs). This model works by taking in input source code and converting it into a code property graph (CPG). Then, it uses type-based graph simplification and variable-based graph simplification to reduce size and complexity of the graph. Then, the code graphs are processed through an edge-aware graph convolutional network (EA-GCN), which computes node vectors according to their type, and then enhances node representation based on a multi-head attention mechanism and returns an edge-enhanced node representation matrix. Then, a kernel-scaled representation module is used to analyze close and long-range relationships between nodes. To do this, kernel-scaled convolutions are used on the edge-enhanced node representation matrix using a large and small kernel. Then, two fully-connected layers and a softmax function are used to perform binary classification. AMPLE demonstrates massive improvements on other models, achieving 92.71% accuracy on the Reveal dataset. A traditional token-based model, SySeVR, only achieves 74.33% accuracy.

Additionally, Devign stands out as one of the most prominent graph-based vulnerability detection approaches. Devign leverages CPGs to represent the data flow and control flow of coding sequences. The architecture of Devign combines these graphs with a Gated Graph Recurrent Network (GGRN), enabling the model to capture both local and global dependencies between code components. The GGRN architecture enhances the learning of long-term dependencies by incorporating recurrent mechanisms. This allows the model to consider both the current and past states of node embeddings, enabling it to capture important patterns that are crucial for accurate vulnerability detection. Devign also takes into account natural language sequences by creating word embeddings of code using a popular tool Word2Vec. Devign's model architecture can be seen in Figure 1. Devign will be an important model to compare the performance of our model against.

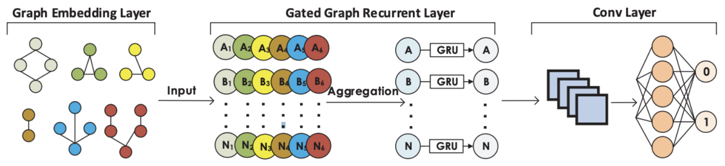

Figure 1: Devign model architecture [29]

3 Data

To get as much data as possible, we utilized three popular C/C++ vulnerability databases: PrimeVul [5], DiverseVul [4], and BigVul [6].

PrimeVul: The PrimeVul dataset contains a total of 224,533 entries, only 6,004 of which being vulnerable (Table 1). Despite its small sample of vulnerable code, the developers of this dataset ensured high label accuracy by using novel labeling techniques, achieving around 94% accuracy [5]. PrimeVul also contains a large variety of vulnerabilities, covering 140 CWE (Common Weakness Enumeration) types.

BigVul: The BigVul dataset contains a total of 185,997 entries, with 10,786 of them being vulnerable. The developers of this dataset ensure label accuracy by taking into account information from the CVE database, project bug reports, along with information from code commits. The BigVul dataset also contains 91 unique CWE types.

DiverseVul: DiverseVul is one of the largest publicly-available vulnerability databases with 330,492 functions. With 18,945 vulnerable functions, it helps address the data imbalance experienced by other datasets. The dataset also spans a variety of vulnerabilities, covering 150 CWEs. 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.

Combined Dataset: To address the data imbalance problem, we combined all three datasets to get a larger and more diverse dataset. This allowed us to expand the number of CWEs we covered, and also gain access to more vulnerable functions. After duplicate data was removed, the combined dataset consisted of $29,867$ vulnerable functions and $506,084$ safe functions (Table 1).

Dataset	Total Entries	Vulnerable Entries	Safe Entries
PrimeVul	224,533	6,004	218,529
BigVul	185,997	10,786	175,211
Diversevul	330,492	18,945	311,547
Combined	535,951	29,867	506,084

Table 1: Number of Total, Vulnerable, and Safe Entries in Various Datasets

Models

4 ChatGPT

As a baseline we decided to explore ChatGPT 4o-mini. We approached this by first converting the data which we had from the PrimeVul database into a more refined and focused dataset, removing redundant information. For this, we had to implement a web scraper in order to pull the ground truth CVSS scores from the NVD website for each of the entries. After that process, we developed a script that sent a uniform vulnerability analysis query to the ChatGPT API for each function in the dataset, in order to ensure consistency across all evaluations thanks to the fixed text provided.

The prompt included a description of the setting, framing the model as a Senior Software Engineer and setting the objective at identifying vulnerabilities in the code functions. The model was then asked to return only a binary output (without explanation), which needed to be 1 if a vulnerability was present and 0 otherwise.

The initial version of the request was highly detailed, explaining which scoring guidelines to refer to in evaluating the functions (CVSS 3.x), but, comparing the model's predictions with the actual vulnerability evaluation, the accuracy registered was really low (almost 40%). Subsequently, simplifying the prompt and making it shorter helped us improve the performance of the model, resulting in a 49.5% success rate.

We are not the first ones to attempt using large language models for vulnerability assessment. Recently (2023), some researchers from Nanyang Technological University and University of Melbourne conducted a study aimed to check whether ChatGPT is able to substitute software engineers with this kind of tasks [7]. They divided their analysis into four different sections, changing the prompt at every step: vulnerability prediction, classification, severity estimation and repair. Our approach differs mainly in the task being framed as binary. Their model, considering a mean of the four tasks, registered an accuracy of 45%, comparable to the performance of our experiment.

4.1 Performance Results

The system performance metrics can be seen below in Table 2. Similarly, Table 3 summarizes the confusion matrix for our evaluation.

Metric	**Value**
Accuracy	49.47%
Precision	31.75%
Recall	88.84%
F1 Score	46.78%

Table 2: Performance metrics of the ChatGPT 4o-mini Model

	Predicted Vulnerable	**Predicted Safe**
Actually Vulnerable	True Positives: 1,600	False Negatives: 201
Actually Safe	False Positives: 3,439	True Negatives: 1,964

Table 3: Confusion matrix for the ML-based vulnerability detector

These results indicate that while the system has a high ability to identify vulnerable code (recall of 88.84%), it suffers from a significant number of false alarms (precision of only 31.75%). This demonstrates the system's tendency to classify code as vulnerable even when it is safe, resulting in a high false positive rate. The overall accuracy of 49.47% suggests that the model performs only marginally better than random chance, despite its strong recall performance. The F1 score of 46.78% reflects this trade-off between high recall and low precision. It is evident that large-language-models like ChatGPT are not yet equipped to effectively classify vulnerabilities and more novel approaches will be required.

5 Rule-Based Static Analysis for Vulnerability Detection

In this section, we examine a rule-based static analysis system for identifying security vulnerabilities in C/C++ code. Unlike machine learning approaches that require extensive training data, this detector employs pattern matching and heuristic rules to detect common security issues. The system processes source code by tokenizing and extracting identifiers, with specialized modules targeting distinct vulnerability classes including buffer overflows, memory management issues, format string vulnerabilities, integer-related problems, and race conditions. We looked at this approach to serve as a benchmark for what could be done without the use of machine learning and see if a deterministic approach is viable for dependable vulnerability detection.

5.1 Risk Assessment Framework

Our implementation utilizes a risk assessment framework that assigns weights to different vulnerability categories and applies multipliers to calculate an overall risk score for each function. This approach allows for prioritization of findings rather than simple binary classification, with things like buffer overflows and array bounds violations receiving higher weights than race conditions or integer overflows. The final vulnerability determination combines the calculated risk score with the ranking to classify code as either "vulnerable" or "safe," with an associated confidence rating.

Let us denote the risk score for a code snippet c as:

$$R(c) = \sum_{i=1}^{n} w_i \cdot I_i(c) \cdot M_i(c) \tag{1}$$

where:

- w_i is the weight assigned to vulnerability category i
- $I_i(c)$ is the indicator function that equals 1 if vulnerability pattern i is detected in code c, and 0 otherwise
- $M_i(c)$ is a multiplier based on the severity and context of the detected vulnerability pattern
- n is the total number of vulnerability patterns examined

The classification decision is then made based on a threshold θ:

$$\text{Classification}(c) = \begin{cases} \text{“Vulnerable”}, & \text{if } R(c) \geq \theta \\ \text{“Safe”}, & \text{otherwise} \end{cases} \tag{2}$$

5.2 Evaluation Methodology

We evaluated the detector on a comprehensive dataset comprising 11,947 code samples spanning multiple Common Weakness Enumeration (CWE) categories. The evaluation metrics include:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5}$$

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6}$$

where TP, TN, FP, and FN represent true positives, true negatives, false positives, and false negatives, respectively.

5.3 Performance Results

The system achieved an accuracy of 72.21%, with precision of 40.79%, recall of 24.67%, and an F1 score of 30.75%. Table 4 summarizes the confusion matrix for our evaluation.

	Predicted Vulnerable	**Predicted Safe**
Actually Vulnerable	True Positives: 737	False Negatives: 2,250
Actually Safe	False Positives: 1,070	True Negatives: 7,890

Table 4: Confusion matrix for the rule-based vulnerability detector

These results indicate that while the system can reliably identify safe code (high true negative rate), it struggles more with comprehensive vulnerability detection (lower recall) and generates a significant number of false alarms (lower precision).

5.4 Performance by Vulnerability Type

Analysis of performance across specific CWE categories reveals considerable variation in detection capabilities, as shown in Table 5.

Vulnerability Type	**CWE**	**Sample Size**	**Detection Rate**
Missing Authentication	CWE-255	Limited	100.00%
Reference Time Comparison	CWE-273	Limited	100.00%
Access Control Problems	CWE-639	Limited	100.00%
Resource Management Errors	CWE-772	Moderate	31.71%
Command Injection	CWE-78	Moderate	30.43%
Denial of Service	CWE-400	Moderate	14.63%
Memory Corruption	CWE-119	757	16.25%
Use-After-Free	CWE-416	357	9.24%
Improper Input Validation	CWE-20	568	11.62%

Table 5: Detection rates by Common Weakness Enumeration (CWE) category

The detector performed relatively well on certain vulnerability types, including missing authentication (CWE-255), reference time comparison issues (CWE-273), and access control problems (CWE-639), all achieving 100% detection rates, albeit on limited samples. More substantively, the detector showed moderate performance on issues like resource management errors (CWE-772) with 31.71% detection rate, command injection vulnerabilities (CWE-78) with 30.43% detection, and denial of service vulnerabilities (CWE-400) with a 14.63% detection rate.

However, the system demonstrated notable weaknesses in detecting certain critical vulnerability categories. For memory corruption vulnerabilities (CWE-119), which represented a significant portion of the dataset with 757 samples, the detection rate was only 16.25%. Similarly, for use-after-free vulnerabilities (CWE-416) with 357 samples, the detection rate was just 9.24%. The detector also struggled with improper input validation (CWE-20), achieving only an 11.62% detection rate across 568 samples.

5.5 Advantages and Limitations

The rule-based approach demonstrates several advantages, including its ability to identify multiple vulnerability classes without training data. However, the relatively low precision and recall metrics highlight the limitations inherent to pattern-based detection. The reliance on regular expressions for code parsing inhibits the ability to comprehensively reason about the code in relation to the setting around it. These limitations likely contributed to both the high false negative rate and moderate false positive rate observed in our evaluation.

5.6 Implications and Applications

In the context of comprehensive security analysis, this rule-based detector provides a valuable baseline approach with advantages in interpretability, efficiency, and independence from labeled training data. While it does not match the detection capabilities of state-of-the-art techniques, particularly for complex vulnerability categories, it offers an accessible entry point for vulnerability detection that can be readily integrated into development workflows and serves as a foundation for more sophisticated analysis approaches.

The significant variation in detection rates across different vulnerability categories suggests that a hybrid approach combining rule-based detection with more advanced techniques might be beneficial. In particular, the strong performance on certain well-defined vulnerability patterns indicates that rule-based components could complement learning-based approaches by providing targeted detection for specific vulnerability classes.

6 Machine Learning-Based Vulnerability Detection: Model Structure and Performance Analysis

In this section, we present a hybrid approach to vulnerability detection that combines statistical machine learning techniques with rule-based feature extraction. Unlike purely rule-based approaches described in Section 5, our model leverages both pattern-based features and learned representations to identify potential security vulnerabilities in C/C++ code.

6.1 Model Architecture

Our vulnerability detection system, `BalancedVulnDetector`, employs a two-pronged feature extraction approach followed by a classification model:

1. **Text-based feature extraction**: We utilize a `HashingVectorizer` to transform code snippets into a high-dimensional sparse representation, capturing n-gram patterns within the code:

$$\mathbf{X}_{\text{text}} = \phi_{\text{text}}(c) \in \mathbb{R}^{n \times d_{\text{text}}} \tag{7}$$

   where c represents a code snippet, ϕ_{text} is the hashing vectorizer transformation, and d_{text} is the dimensionality of the text features (set to 1000 in our code).

2. **Vulnerability-specific feature extraction**: We implement a custom feature extractor (`VulnFeatureExtractor`) that identifies both vulnerability patterns and safety checks using regular expressions:

$$\mathbf{X}_{\text{code}} = \phi_{\text{code}}(c) \in \mathbb{R}^{n \times d_{\text{code}}} \tag{8}$$

where ϕ_{code} represents the vulnerability-specific feature extraction function.

The combined feature representation is given by:

$$\mathbf{X}_{\text{combined}} = [\mathbf{X}_{\text{text}} \ \mathbf{X}_{\text{code}}] \in \mathbb{R}^{n \times (d_{\text{text}} + d_{\text{code}})} \tag{9}$$

For classification, we employ a Random Forest classifier with calibrated class weights to handle class imbalance. The probability of a code snippet being vulnerable is modeled as:

$$P(y = 1|\mathbf{X}_{\text{combined}}) = f_{\text{RF}}(\mathbf{X}_{\text{combined}}; \theta) \tag{10}$$

where f_{RF} is the Random Forest classifier and θ represents the model parameters.

6.2 Class Imbalance Handling

To address the prevalent class imbalance in vulnerability datasets, we employ a dynamic weighting strategy:

$$w_i = \begin{cases} 1.0, & \text{if } i = 0 \text{ (non-vulnerable)} \\ \frac{N}{2 \times N_{\text{vuln}}}, & \text{if } i = 1 \text{ (vulnerable)} \end{cases} \tag{11}$$

where N is the total number of samples and N_{vuln} is the number of vulnerable samples. This approach ensures that vulnerable samples receive a controllably higher importance during training, decreasing the bias towards the majority class.

6.3 Threshold Optimization

We also optimized the classification threshold to balance recall and the review burden (the amount of code that is flagged as vulnerable and therefore in need of review):

$$\text{score}(\tau) = 0.7 \times \text{recall}(\tau) - 0.3 \times \text{review burden}(\tau) \tag{12}$$

with an adjustment factor for recall below a critical threshold:

$$\text{adjusted score}(\tau) = \begin{cases} \text{score}(\tau), & \text{if recall}(\tau) \geq 0.8 \\ \text{score}(\tau) \times \frac{\text{recall}(\tau)}{0.8}, & \text{otherwise} \end{cases} \tag{13}$$

where τ represents a possible threshold. The optimal threshold τ^* is determined by:

$$\tau^* = \underset{\tau \in [0.3, 0.9]}{\text{argmax}} \text{ adjusted score}(\tau) \tag{14}$$

6.4 Detection Analysis

Beyond binary classification, our system performs rule-based analysis to identify specific vulnerability types. For each code snippet, we apply a series of pattern-matching rules and return structured results:

$$\text{analyze}(c) = \{\text{is\_vulnerable}, \text{probability}, \text{issues}\} \tag{15}$$

where `issues` is a list of detected vulnerability types and descriptions.

6.5 Experimental Results

We evaluated our model on a comprehensive dataset of C/C++ code samples. The performance metrics are as follows:

The confusion matrix reveals the distribution of predictions as shown in Table 7. These results indicate that our model achieves high recall (83.36%), effectively identifying most vulnerable code snippets. However, the precision is relatively low (34.98%), resulting in a significant number of false positives.

The vulnerability type analysis shows that our model is particularly effective at detecting general vulnerabilities (83.4% recall for the “Other” category). However, the model’s performance may vary across specific vulnerability types.

Metric	Value
Accuracy	57.10%
Precision	34.98%
Recall	83.36%
F1 Score	49.28%
False Positive Rate	51.65%
Review Burden	59.60%

Table 6: Performance metrics of the machine learning-based vulnerability detector

	Predicted Vulnerable	Predicted Safe
Actually Vulnerable	True Positives: 2,490	False Negatives: 497
Actually Safe	False Positives: 4,628	True Negatives: 4,332

Table 7: Confusion matrix for the vulnerability detection model

6.6 Comparative Analysis

Comparing our machine learning approach with the rule-based system described in Section 4, we observe significant differences in detection capabilities, as shown in Table 8.

Metric	Rule-Based System	ML-Based System
Accuracy	72.21%	57.10%
Precision	40.79%	34.98%
Recall	24.67%	83.36%
F1 Score	30.75%	49.28%

Table 8: Comparative analysis of rule-based and machine learning-based approaches

This comparison highlights a fundamental trade-off: the rule-based system achieves higher precision and accuracy but misses many vulnerabilities (low recall), while our machine learning approach prioritizes vulnerability detection (high recall) at the cost of more false positives (lower precision).

The substantial improvement in recall (from 24.67% to 83.36%) suggests that our hybrid approach effectively captures vulnerability patterns that rule-based systems miss. However, the increased review burden (59.6%) may present practical challenges for integration into development workflows.

6.7 Limitations and Future Work

Despite the promising recall, our model faces several limitations:

1. The relatively high false positive rate (51.65%) indicates room for improvement in distinguishing between vulnerable and safe code.
2. The current feature extraction approach may not fully capture semantic relationships and data flow information that are crucial for certain vulnerability classes.
3. The model's performance varies across different vulnerability types, suggesting that specialized models or ensemble approaches might be beneficial.

Future work should explore deeper code representations, potentially incorporating abstract syntax trees or program dependency graphs, to improve precision while maintaining high recall. Additionally, incremental learning strategies could help adapt the model to evolving vulnerability patterns over time.

7 Graph Neural Network

7.1 Problem Formulation

To detect code vulnerabilities at the function level, many researchers have leveraged the use of Graph Neural Networks (GNNs) [26, 29]. Let a sample of data be defined as follows: $(c_i, y_i) \in D$, $i \in \{1, \ldots, n\}$ where n is the number of entries in the dataset, D is the dataset, c_i is the code sample for entry i, and $y_i \in \{0, 1\}$ is the vulnerability label for that sample. c_i is then encoded as a graph represented as follows: $G(V, E)$, where V is the set of vertices and E the set of edges (in the form of an edge index array). Each graph also has a corresponding node feature matrix X, and an edge-attribute matrix A. Each node, $v_i \in V$, typically corresponds to a program-element such as a variable or function call. Each edge $e_i \in E$ captures semantic information such as control flow or data. The goal of this GNN is to learn a mapping from G to Y, $f : G \rightarrow Y$, to predict whether a function contains a vulnerability or not. The function f can be learned by minimizing the following function:

$$min \sum_{i=1}^{n} L(f(G_i(V, E)), y_i | c_i) + \lambda \sum_{j}^{m} ||W_j||^2 \tag{16}$$

where $L(\cdot)$ is the focal loss function (17) [13] , m is the number of learnable weight matrices, and W_j represents the j-th weight matrix.

$$L_{\text{focal}} = -\alpha_t (1 - p_t)^{\gamma} \log(p_t) \tag{17}$$

Here, p_t is the predicted probability for the true class, and α_t and γ are tunable hyperparameters for class weighting and focusing, respectively.

7.2 Data Processing

7.2.1 Graph Embedding

During graph generation, code is extracted from each entry in the database. The code is not standardized, and passed into graph generation as-is. An example non-vulnerable function from the PrimeVul dataset can be seen in Code Block 1. Then, a joint graph is generated for each function, using regular expressions to extract syntactic and semantic code elements, a rule-based approach similar to the URG-J algorithm described in [10]. An example graph can be seen in Figure 2.

Code Block 1: Example C++ function that checks for metadata presence.

```cpp
bool findMetadata(const Metadata::Filter& filter, const int32_t val) {
    if (filter.isEmpty()) return false;
    if (filter[0] == Metadata::kAny) return true;

    return filter.indexOf(val) >= 0;
}
```

7.2.2 Node Feature Matrix Generation

Firstly, a node feature matrix, X, will need to be generated in order to capture node information. To capture semantic information for node identifiers (variable or function names), we utilized the popular natural-language-processing tool Word2Vec [15] to create vector embeddings from words. The model was trained on entire tokenized functions from the dataset. Vectors in X were then generated containing:

1. A one-hot encoded vector $t_i \in \{0, 1\}^7$ representing the node type of node i. The vector contains a single 1 at the index corresponding to the node's type (e.g., `FunctionCall`, `Variable`, etc.), and 0s elsewhere.

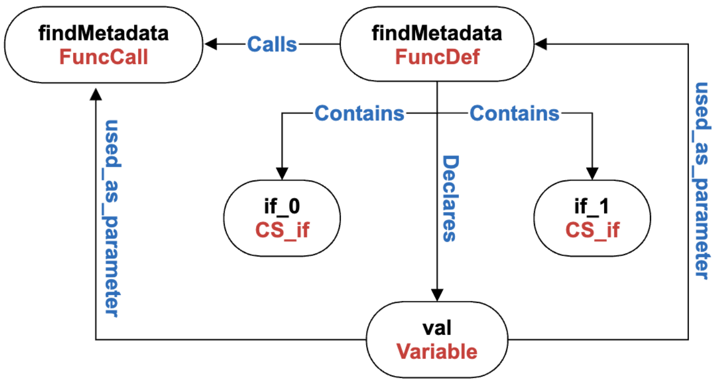

Figure 2: Graph generated from the code seen in Code Block 1

2. An embedding vector, $e_i \in \mathbb{R}^{100}$, computed as the average of the Word2Vec embeddings of the subtokens extracted from the node's name. If no subtokens are matched in the Word2Vec vocabulary, e_i defaults to the zero vector.

Thus, the vector $x_i \in X$ for node $v_i \in V$ is: $[t_i, e_i]^T$. The full node feature matrix $X \in \mathbb{R}^{|V|\times 107}$ is defined as:

$$X = \begin{bmatrix} t_1 & t_2 & t_3 & \dots \\ e_1 & e_2 & e_3 & \dots \end{bmatrix} = [x_1, x_2, \dots, x_{|V|}] \tag{18}$$

Node Type	**Description**
`FunctionCall`	Represents a function call
`Variable`	A named variable in the code
`ControlStructure_if`	An if-statement branch
`ControlStructure_while`	A while-loop construct
`ControlStructure_switch`	A switch-case block
`ControlStructure_for`	A for-loop construct
`FunctionDefinition`	A function definition

Table 9: Predefined node types used in graph construction.

7.2.3 Edge Index and Edge Type Matrix Generation

To construct the edge-related data for the GNN, we utilize two structures:

1. The **edge index matrix** $E \in \mathbb{Z}^{2\times|E|}$, which encodes the graph's edges. Each column represents a directed edge (u, v), where u and v are node indices from the node feature matrix X. This is essentially an adjacency matrix compatible with PyTorch Geometric.

2. The **edge attribute vector** $A \in \mathbb{Z}^{|E|}$, where each entry corresponds to an edge type (Table 10) for the respective edge e_i.

Given a graph $G = (V, E)$, we iterate over each edge $(u, v) \in E$ and extract its edge type (e.g., `calls`, `declares`, etc.). Each edge is mapped to an integer according to a pre-defined mapping. If an edge type is not recognized, it is assigned a value of -1 to be ignored during training.

Each graph G is thus represented by three components: a node feature matrix X, the edge index matrix E, and the edge attribute vector A, which collectively define the input to the GNN model.

Edge Type	Description
`declares`	Indicates a declaration relationship.
`calls`	Represents a function call from one node to another.
`contains`	Denotes structural containment (e.g., a function contains a statement).
`used_in_condition`	Marks variables or expressions used in control conditions such as `if`, `while`, or `switch` statements.
`used_as_parameter`	Marks nodes that are passed as parameters to a function call.
`used_in_body`	Captures general usage of elements inside the body of a function or code block.

Table 10: Predefined edge types used in graph construction.

7.3 Architecture

7.3.1 The Convolution Layer

Many existing graph neural network approaches to vulnerability detection have used aggregation techniques like graph convolution networks (GCNs) [11], graph attention networks (GATs) [22], gated graph recurrent networks (GGRNs), and their variants. For this project, we designed and tested three models, each of which using either GCNs, RGCNs, or GATs.

Graph Convolution Layer (GCN)

GCN [11]: Uses the propagation rule:

$$\mathbf{H}^{(l+1)} = \sigma\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) \tag{19}$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ is the adjacency matrix of the graph G with added self-loops, and $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$ is the corresponding degree matrix (containing the number of connected edges per node). $\mathbf{W}^{(l)}$ is the trainable weight matrix for layer l, and $\sigma(\cdot)$ is a nonlinear activation function like ReLU. $\mathbf{H}^{(l)}$ denotes the output node embeddings after layer l, with $\mathbf{H}^{(0)} = \mathbf{X}$, the initial input features.

GAT [22]: Graph Attention Networks compute attention scores between nodes and their neighbors:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}h_i||\mathbf{W}h_j]))}{\sum_{k\in\mathcal{N}_i}\exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}h_i||\mathbf{W}h_k]))} \tag{20}$$

where $\|$ represents the concatenation operation. h_i represents the set of node features for node i. A shared linear transformation is applied to every node through the learnable weight matrix $\mathbf{W}$. Additionally, a shared attention mechanism, $\mathbf{a}$, is applied to every transformation, and is used to compute the importance of node j's features on node i. $\mathcal{N}_i$ represents the set of neighbor nodes of node i in the graph. α_{ij} represents the attention coefficient, representing the normalized importance of node j's features on node i. Then, the result of feature aggregation, h'_i, is obtained through multi-head attention mechanisms as follows:

$$h'_i = \|_{k=1}^{K}\sigma(\sum_{j\in\mathcal{N}_i}\alpha_{ij}^k\mathbf{W}^k h_j) \tag{21}$$

where α_{ij}^k are normalized attention coefficients computed at the k^{th} attention mechanism and $\mathbf{W}^k$ is the linear transformation weight matrix for attention mechanism k.

RGCN [19]: Relational Graph Convolution Networks extend GCNs to handle multiple edge types:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{N}_i^r}\frac{1}{c_{i,r}}\mathbf{W}_r^{(l)}\mathbf{h}_j^{(l)} + \mathbf{W}_0^{(l)}\mathbf{h}_i^{(l)}\right) \tag{22}$$

where $\mathcal{R}$ is the set of relation types, and $\mathcal{N}_i^r$ denotes the set of neighborhood indices to node i of relation r. $\mathbf{W}_r^{(l)}$ is a learnable weight matrix specific to relation r. $c_{i,r}$ is a problem-specific normalization constant that can either be learned or chosen in advance. $\mathbf{W}_0^{(l)}\mathbf{h}_i^{(l)}$ is added to account for self-loop contributions.

7.3.2 Model Flow

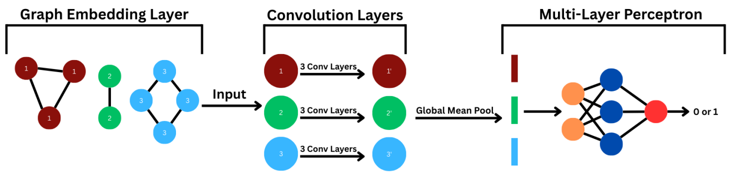

Figure 3: GNN model architecture

Each of our three models follow a similar flow (as seen in Figure 3):

1. The node features are passed through the respective graph convolution layers, followed by activation functions (ReLU) and dropout for regularization.
2. Node features are aggregated using global mean pooling to generate graph-level embeddings.
3. The graph-level features are concatenated with additional graph-level flags and passed through an multi-layer perceptron for the final classification.

7.4 Evaluation

To assess the performance of our graph-based model, we aimed to answer the following questions:
Q1: How does our model compare to the state-of-the-art graph-based binary classification models?
Q2: Which of the convolution models provides the best performance?
Q3: What types of vulnerabilities does the model continuously fail to identify?
Q4: How does the model handle class imbalance?
Q5: What are the next steps needed to improve the performance of vulnerability detection via machine learning methods?

7.4.1 Dataset Preparation

To test our GNN models, we used the combined and PrimeVul datasets defined in section 3. Safe entries were downsampled across the entire dataset. In the training dataset, vulnerable entries were randomly oversampled to achieve a 50/50 split between vulnerable and safe entries (Table 11). Entries in the combined dataset were then randomly shuffled.

7.4.2 Results

In the embedding layer, the dimension of Word2Vec for the initial node representation is 100. For the GCN, GAT, and RGCN layers, the dimension of hidden states was set to 128. Three convolution layers were used for all three convolution types. We use the Adam optimizer with a learning rate of 0.001, and batch size of 128. We additionally used an L2 regularization rate of 1×10^{-5}, and dropout of 0.3 to minimize overfitting. Each model was evaluated over 25 epochs. The decision threshold was adjusted based on Youden's J statistic from the ROC curve [1], aiming to balance vulnerable samples and false positives. All models were tested after training using the test dataset, while during training, they were tested using the validation dataset after each epoch. See Table 12 for results.

Dataset	**Total Entries**	**Percent Duplicates**	**Vulnerable Entries**	**Safe Entries**
Complete Dataset				
Train	354,258	44.00%	177,129 (50.00%)	177,129 (50.00%)
Test	80,393	0.00%	4,480 (5.57%)	75,913 (94.43%)
Validation	80,393	0.00%	4,480 (5.57%)	75,913 (94.43%)
PrimeVul Dataset				
Train	8,406	41.67%	4,203 (50.00%)	4,203 (50.00%)
Test	6,363	0.00%	900 (14.14%)	5,463 (85.86%)
Validation	6,364	0.00%	901 (14.16%)	5,463 (85.84%)

Table 11: Datasets used for training, testing, and validation for Complete and PrimeVul datasets

GCN: The model using the GCN convolution achieved very low precision, indicating a high number of misclassifications of safe samples as vulnerable. However, it demonstrated high recall, reflecting a strong ability to identify vulnerable samples. This is particularly valuable in vulnerability detection, where false negatives are more detrimental than false positives. Despite this, the model showed unstable validation accuracy compared to the more consistent training accuracy (Figure 5).

GAT: The model using the GAT convolution achieved moderate precision, indicating a lower rate of misclassifying safe samples as vulnerable than the GCN-based model. The GAT-based model also achieves moderate recall, though lower than that of the GCN. The model achieved validation accuracy closer to train accuracy (Figure 4).

RGCN: The RGCN-based model achieved a precision of 0.177—the highest among all architectures — highlighting its strong ability to minimize false positives. In terms of recall, the RGCN reached 0.639, which, while slightly lower than the values observed in GAT and GCN-based models, still reflects a strong ability to capture true positives. Additionally, the RGCN exhibited relatively stable and high validation accuracy throughout training, consistently ranging between 80% and 85% (Figure 6).

Devign: The state-of-the-art Devign model exhibited relatively high precision, indicating that it is conservative in predicting vulnerabilities and avoids misclassifying safe samples as vulnerable. However, its recall was significantly lower, meaning it misses a substantial number of true positives. Devign also demonstrated the lowest accuracy of all models at 43.6%.

Model	Accuracy	Precision	Recall	F1 Score
Complete Dataset				
GCN	60.84%	10.45%	79.60%	18.47%
GAT	75.45%	13.73%	64.44%	22.64%
RGCN	81.42%	17.70%	63.93%	27.72%
Devign	43.60%	44.70%	40.40%	42.40%
PrimeVul Dataset				
GCN	34.42%	16.54%	89.89%	27.94%
GAT	35.39%	16.77%	90.00%	28.27%
RGCN	39.29%	18.21%	94.33%	30.53%
Devign	48.50%	50.00%	48.10%	49.00%

Table 12: ML Stats for Complete and PrimeVul Datasets

Performance on specific vulnerabilities: Based on results from the RGCN model, the model classified the following vulnerabilities well: authentication bypass, improper initialization, and missing encryption,

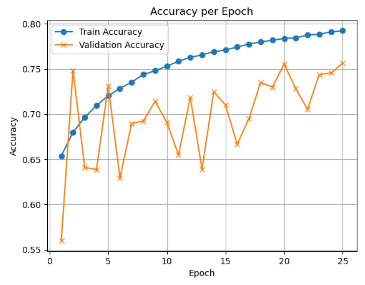

Figure 4: GAT Accuracy (Complete)

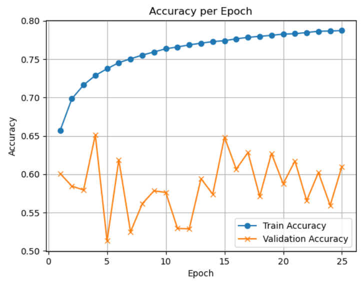

Figure 5: GCN Accuracy (Complete)

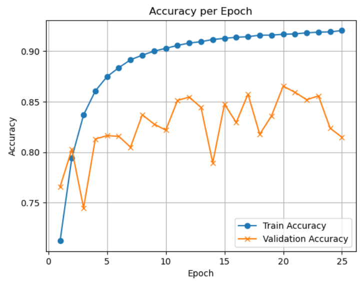

Figure 6: RGCN Accuracy (Complete)

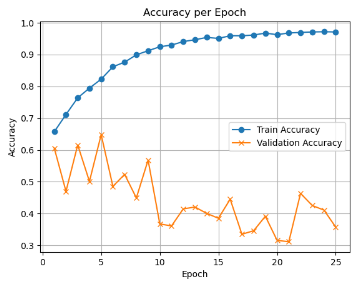

Figure 7: GAT Accuracy (PrimeVul)

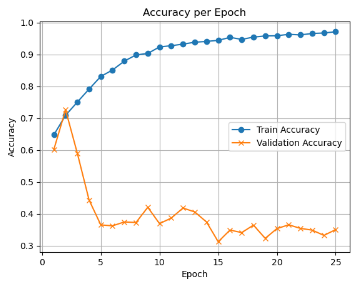

Figure 8: GCN Accuracy (PrimeVul)

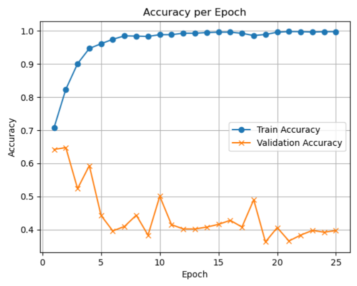

Figure 9: RGCN Accuracy (PrimeVul)

Figure 10: Accuracy Curves for GAT, GCN, and RGCN across Complete and PrimeVul Datasets

achieving 100% detection rates on all three (Table 13). The model fails to fully identify the following vulnerabilities: resource management error, information exposure, race condition, numeric error, and inadequate access controls, only achieving between 60-70% detection on these vulnerabilities (Table 13).

7.4.3 Discussion

The results indicate that, while Graph Neural Networks (GNNs) show potential in vulnerability detection, there remains significant room for improvement, especially when considering the low F1 scores observed across all models. The trade-offs between precision, recall, and F1 score highlight the challenges of achieving a balanced and effective model for vulnerability detection.

Comparison Between Datasets: A notable comparison is between the performance on the Complete dataset and the PrimeVul dataset. While the models performed relatively well on the Complete dataset, achieving an accuracy of around 60-80% for GCN, GAT, and RGCN, the performance on the PrimeVul dataset alone was much lower. Though the F1 scores are misleadingly higher (Table 12), the accuracy is extremely low. Additionally, during training, the validation accuracy plummets, indicating the PrimeVul dataset alone does not have enough information to allow the model to generalize. Thus, all models were trained on the combined dataset.

Precision vs Recall The GCN model demonstrated high recall but very low precision, meaning that while it was able to identify many of the vulnerable samples, it frequently misclassified safe samples as vulnerable. While it is important in vulnerability detection to minimize false negatives (i.e. avoid misclassifying vulnerable code as safe), the low F1 score, precision, and accuracy indicates the model is not generalizing well on the data.

In contrast, the GAT model displayed more stable validation accuracy, suggesting that it has better generalization capabilities. The model also demonstrates higher accuracy, precision, recall, and F1, indicating the GAT convolution is better at learning vulnerability patterns from the graph. However, there is still a lot of room for improvement.

The RGCN model performed the best overall, with the highest F1 score and validation accuracy. Though it has lower recall as compared to the GAT and GCN, it demonstrates the best balance between precision and recall. Additionally, its stable and high validation accuracy indicates a better ability to generalize insights from the train data on the validation dataset. Overall, the high accuracy and F1 achieved using the RGCN convolution make it the most reliable for vulnerability detection in this study. However, its slightly lower recall compared to the GCN suggests that further improvements can be made to maximize its sensitivity to vulnerabilities.

Most Frequently Misclassified Vulnerabilities: While the model is good at detecting certain vulnerabilities (Table 13), the sample sizes for these vulnerabilities is fairly low and inconclusive. The worst performing vulnerabilities, including resource management error, information exposure, race condition, numeric error, and inadequate access controls, involve complex patterns that the model may struggle to identify. Though a detection rate of 60-70% is not ideal, it is still better than a random guess.

Vulnerability Type	**CWE**	**Sample Size**	**Detection Rate (%)**
Best Performing			
Authentication Bypass	CWE-290	21	100.00%
Improper Initialization	CWE-665	15	100.00%
Missing Encryption	CWE-311	52	100.00%
Worst Performing			
Resource Management Error	CWE-399	4218	63.00%
Information Exposure	CWE-200	4865	65.60%
Race Condition	CWE-362	2755	69.30%
Numeric Error	CWE-189	4093	65.30%
Inadequate Access Controls	CWE-772	996	61.00%

Table 13: Comparison of Best and Worst Performing Vulnerabilities

Devign The state-of-the-art Devign model utilizes word embeddings (with Word2Vec) with graph-based features, similarly to our model. Devign uses Joern [20], a pre-existing tool to generate complex code property graphs, as compared to our fairly simple graph representations. Devign also utilizes Gated Graph Recurrent Networks [18], where we use GCNs, GATs, and RGCNs. Devign showed relatively high precision, meaning it is cautious in predicting vulnerabilities and avoids misclassifying safe samples as vulnerable. However, its low recall indicates that it misses a significant portion of true vulnerabilities. The model also demonstrated the lowest accuracy, which indicates that Devign struggles to generalize well to the validation dataset. The combination of low recall and accuracy points to significant limitations in its ability to perform well in vulnerability detection on the current dataset.

Conclusion The low F1 scores across all architectures suggest that there is still much to be done to improve the models' performance, particularly in terms of increasing recall while maintaining precision. A possible approach to achieving this goal would be to improve graph expression of data. One popular graph generation tool is Joern [20], which generates more complex code property graphs (CPGs). Using graphs generated by Joern on our model could improve performance. Along with this, further exploration of feature engineering could help create better code context, helping the model detect complex vulnerabilities easier. Additionally, further work will need to be done to find vulnerable samples, as the massive class imbalance in the dataset has limited the performance of the model.

Overall, when comparing our model with the state-of-the-art model Devign, it is clear we have made some improvements toward vulnerability detection. However, more work will need to be done in order to detect complex vulnerabilities more reliably.

8 Transformer Block Overview

Transformers are neural network architectures designed to process inputs represented as unordered sets of tokens. Each token $x_n^{(0)}$ is a vector in $\mathbb{R}^D$, and the full input is arranged as a matrix $X^{(0)} \in \mathbb{R}^{D\times N}$, where N is the number of tokens and D is the feature dimension. This design allows transformers to handle diverse data types, such as text or image patches, using a unified architecture. As Turner notes, this removes the need for "bespoke handcrafted architectures for mixing data of different modalities" [21]. The transformer operates by applying a block of operations repeatedly to the input:

$$X^{(m)} = \text{transformer-block}(X^{(m-1)}) \tag{23}$$

Each block consists of two main stages:

- **Stage 1: Self-Attention across the sequence**
- **Stage 2: Multi-Layer Perceptron (MLP) across features**

8.1 Stage 1: Self-Attention

Self-attention allows each token to incorporate information from the entire sequence by computing a weighted sum of all input representations. Given input matrix $X^{(m-1)} \in \mathbb{R}^{D\times N}$, we first project into queries and keys:

$$q_n \;=\; U_q\, x_n, \qquad k_n \;=\; U_k\, x_n, \quad U_q,\, U_k \in \mathbb{R}^{d_k\times D},\; n = 1,\ldots,N. \tag{24}$$

Next, we form the raw score matrix $W \in \mathbb{R}^{N\times N}$ with scaled dot-products:

$$W_{n,n'} = \frac{q_n^\top k_{n'}}{\sqrt{d_k}} \qquad (n, n' = 1,\ldots,N). \tag{25}$$

We then normalize each row of W via a softmax to obtain the attention weights:

$$A_{n,n'} = \text{softmax}\big(W_{n,*}\big)_{n'} = \frac{\exp\big(W_{n,n'}\big)}{\sum_{n''=1}^{N} \exp\big(W_{n,n''}\big)}. \tag{26}$$

Finally, the output of the self-attention layer is the weighted sum

$$Y^{(m)} = X^{(m-1)}\, A^{(m)}, \tag{27}$$

where $A^{(m)}$ collects the weights $A_{n,n'}$. This introduces the projections

$$q_n = U_q x_n, \quad k_n = U_k x_n, \tag{28}$$

which map each input vector into a "query" and "key" space where similarity scores are computed. In effect, self-attention re-expresses each vector as a weighted sum of all inputs: if A and B are similar (high dot-product in query/key space), then their outputs both become roughly $A + B$, and similarly for other groups like C and D. This lets the model dynamically aggregate information from semantically related tokens.

8.2 Multi-Head Self-Attention (MHSA)

To allow the model to attend to information in different representational subspaces, multi-head self-attention applies H independent heads in parallel:

$$\text{head}_h^{(m)} = V_h\, X^{(m-1)}\, A_h^{(m)}, \qquad h = 1,\ldots,H, \tag{29}$$

where each attention matrix is

$$[A_h^{(m)}]_{n,n'} = \frac{\exp\big((k_{h,n}^{(m)})^\top q_{h,n'}^{(m)}\big)}{\sum_{n''=1}^{N} \exp\big((k_{h,n''}^{(m)})^\top q_{h,n'}^{(m)}\big)}, \tag{30}$$

and

$$q_{h,n}^{(m)} = U_{q,h}^{(m)}\, x_n^{(m-1)}, \quad k_{h,n}^{(m)} = U_{k,h}^{(m)}\, x_n^{(m-1)}. \tag{31}$$

Rather than summing the heads, we concatenate their outputs and apply a final linear projection $W^O \in \mathbb{R}^{D\times(H\,D)}$:

$$Y^{(m)} = W^O \left[\text{head}_1^{(m)} \,\|\, \text{head}_2^{(m)} \,\|\, \cdots \,\|\, \text{head}_H^{(m)}\right], \tag{32}$$

where $\|$ denotes concatenation along the feature dimension.

8.3 Stage 2: MLP Across Features

After self-attention, each token's vector is refined using a non-linear transformation, applied independently across tokens:

$$x_n^{(m)} = \text{MLP}(y_n^{(m)}) \tag{33}$$

This operation is akin to the update step in Graph Neural Networks [21].

8.4 Stabilization: Residual Connections and Normalisation

Each sub-layer is wrapped with a residual connection:

$$x^{(m)} = x^{(m-1)} + \text{residual}(x^{(m-1)}) \tag{34}$$

and uses **Layer Normalisation** [21]

$$\text{LayerNorm}(x_{d,n}) = \frac{x_{d,n} - \mu_n}{\sqrt{\sigma_n^2}}\gamma_d + \beta_d \tag{35}$$

where

$$\mu_n = \frac{1}{D}\sum_{d=1}^{D} x_{d,n}, \quad \sigma_n^2 = \frac{1}{D}\sum_{d=1}^{D}(x_{d,n} - \mu_n)^2 \tag{36}$$

and γ_d, β_d are learned scale and shift parameters.

8.5 Positional Encoding

Because transformers are permutation-invariant, they require additional information to capture token order. This is done by adding or concatenating a positional embedding to each token:

$$x_n^{(0)} = Wp_n + e_n \tag{37}$$

where p_n is the input patch or token, W is the patch embedding matrix, and e_n is the position embedding. [21]

8.6 Conclusion

The transformer architecture consists of stacked transformer blocks, each combining multi-head self-attention and MLP layers, along with residual connections and layer normalization. By processing sequences in this way, transformers can model complex dependencies across tokens and have become a foundational tool in modern machine learning.

9 CodeT5

CodeT5 is a T5 model transformer model created by the Saleforce Research group. This model was created to be more apt at handling code and thus improve software development. The separating factor from CodeT5 and other transformer models like BERT and GPT is that it implements an encoder-decoder architecture which allows for better code understanding and generation. BERT is an encoder-only model while GPT is a decoder-only model which limits their effectiveness. In order to achieve this, the SaleForce team implemented the following [23] :

- Masked Span Prediction - Hiding chunks or spans of code or comments and having the model reconstruct them
- Identifier Tagging - Enhancing the encoder to allow for detection of variables which can be useful later on
- Masked Identifier Prediction - Hide variable names and allows the model to assign a label which negates variable name importance
- Bimodal Dual Generation - Training the model to turn comments into code and vice versa

This enables CodeT5 to excel at code generation, summarization, translation, and understanding tasks across multiple programming languages.

9.1 Training Procedure

Our code fine-tunes the CodeT5-base model by first loading the pre-trained model and tokenizer, which process code as sequences of tokens, enabling the transformer's self-attention mechanisms to model contextual relationships. Then using the pandas library the data is formatted in a data frame. For our complete dataset, the batch size, steps per epoch, and epochs are calculated to cover the dataset. Entering the training loop, each batch is tokenized, and passed through the model. The model outputs a 2-dimensional vector so that it can perform cross entropy loss in order to fine tune parameters using gradient descent. The script outputs the following features in the log for bookkeeping and for checking that the model is functioning:

- Batch Size
- Current Epoch
- Current Step
- The loss, accuracy, and the predictions paired with their actual classification
- The average loss over the epoch
- Time to complete the epoch

```
Starting training...
Batch size: 4

Epoch 1/700

Step 1 – loss 0.6656, acc 0.7500
(Prediction, output): [(0, 0), (1, 1), (0, 0), (1, 0)]

Step 2 – loss 0.6195, acc 0.7500
(Prediction, output): [(1, 1), (1, 0), (1, 1), (0, 0)]

Step 3 – loss 0.7734, acc 0.5000
(Prediction, output): [(0, 0), (0, 1), (1, 0), (1, 1)]

Step 4 – loss 0.6145, acc 0.7500
(Prediction, output): [(0, 0), (0, 0), (1, 0), (0, 0)]

Step 5 – loss 0.7395, acc 0.2500
(Prediction, output): [(0, 0), (1, 0), (0, 1), (1, 0)]
```

Figure 11: Example first few lines from the log file after training

```

Step 125 – loss 0.2208, acc 1.0000
(Prediction, output): [(0, 0), (1, 1), (0, 0), (0, 0)]

Epoch done in 43.6s, average loss: 0.7780

```

Figure 12: Example of log file at the end of an epoch

The batch size was set as four through trial and error on the Boston College High Performance Computer. Four data points per batch allowed for the GPU to perform the training without exceeding the GPU's memory. The following steps per epoch and epochs were calculated to cover as much of the data as a result of the batch size being restricted to four. Finally, after logging, the model will repeat this loop until all epochs are completed and then save out the model to a folder.

9.2 Testing The Trained Model

After the training script saves out the model, it is loaded into the test script. This script loads the data in a similar process as the training script. Then, each batch is feed through the model. The data point's true values are appended to an array with their corresponding predictions being appended to a separate array. These arrays are compared to ensure that the model has been feed all the entries. Then, using the sklearn library, the following metrics are calculated and outputted to a log file:

- Accuracy Score
- Precision Score
- Recall Score
- F1 Score

9.3 Model Evaluation

The model was trained on two data sets – the complete dataset and the PrimeVul dataset. Table 14 summarizes the overall classification metrics for each model.

The Complete Dataset model achieves a high accuracy of 88.48 %, driven largely by its ability to correctly identify the majority class (non-vulnerable functions). Its recall of 95.49 % indicates that it successfully captures almost all true vulnerable cases; however, at the expense of precision (32.96 %), leading to a moderate F1 Score of 49.01 %.

In contrast, the PrimeVul Dataset model shows lower overall accuracy (77.74 %) and recall (76.11 %), reflecting the increased difficulty of this focused subset. Its precision remains similar (32.16 %), yielding

Metric	Complete Dataset Model	PrimeVul Dataset Model
Accuracy	88.48%	77.74%
Precision	32.96%	32.16%
Recall	95.49%	76.11%
F1 Score	49.01%	45.21%

Table 14: Overall performance metrics on the two evaluation datasets.

an F1 Score of 45.21 %. This suggests that while the model generalizes well across the full dataset, its performance degrades somewhat when restricted to the prime-vulnerability domain.

To better understand where each model makes errors, we examine the confusion matrices below. Table 15 shows the Complete Dataset results on 27,900 examples, and Table 16 shows the PrimeVul Dataset results on 7,456 examples.

	Predicted Vulnerable	Predicted Safe
Actually Vulnerable	True Positives: 1,544	False Negatives: 74
Actually Safe	False Positives: 3,141	True Negatives: 23,141

Table 15: Confusion matrix for the Completed Dataset model (27,900 examples).

In the Complete Dataset confusion matrix, the model correctly labels 23,141 of 26,282 non-vulnerable samples (true negatives) and 1,544 of 1,618 vulnerable samples (true positives). Only 74 vulnerable samples are missed (false negatives), confirming the high recall. However, 3,141 non-vulnerable samples are incorrectly flagged as vulnerable (false positives), which drives down the precision.

	Predicted Vulnerable	Predicted Safe
Actually Vulnerable	True Positives: 685	False Negatives: 215
Actually Safe	False Positives: 1,444	True Negatives: 5,112

Table 16: Confusion matrix for the PrimeVul Dataset model (7,456 examples).

On the PrimeVul Dataset, the model still identifies the majority of non-vulnerable samples correctly (5,112 true negatives), but it misses more vulnerable cases (215 false negatives) relative to the smaller positive class. The increase in both false positives (1,444) and false negatives reflects the greater challenge of this domain and explains the drop in both accuracy and recall.

Overall, these results highlight the trade-off between capturing as many true vulnerabilities as possible (recall) and avoiding excessive false alarms (precision). Tuning this balance will depend on the practical needs of the security pipeline: whether catching every possible vulnerability is paramount or reducing investigator workload from false positives is more critical.

10 Comparison of Models

Below is Table 17 which compares our eight models explored in this paper across the four main metrics. For models that were trained on the Complete dataset and PrimeVul dataset, the Complete dataset was used for consistency among comparison. The highest result for each metric is bolded.

Based on these results, the model performing the best overall – highest F1 Score – is the ML-Based Model with an F1 Score of 49.28%. This model yields the most balanced approach between accurately detecting vulnerabilities – recall – and minimizing false positives – precision.

If catching vulnerabilities is the utmost paramount, then recall is prioritized. CodeT5 – at 95.49% recall – is the best choice for maximizing detection, though at the expense of a higher false-positive rate as its Precision Score is only 32.96%.

For those that do not want to waste time on wasteful reviews, then precision needs to be preferential. The Devign model is best suited for this with the highest Precision Score of 44.70%. Thus, it appeals to those with workflows that hinge on reducing wasteful time and only want to flag truly critical issues.

Metric	**GPT**	**Rule-Based**	**ML-Based**	**GCN**	**GAT**	**RGCN**	**Devign**	**CodeT5**
Accuracy	49.47%	46.78%	57.10%	60.84%	75.45%	81.42%	43.60%	**88.48%**
Precision	31.75%	40.79%	34.98%	10.45%	13.73%	17.70%	**44.70%**	32.96%
Recall	88.84%	24.67%	83.36%	79.60%	64.44%	63.93%	40.40%	**95.49%**
F1 Score	46.78%	30.75%	**49.28%**	18.47%	22.64%	27.72%	42.40%	49.01%

Table 17: Performance metrics for each of the eight evaluated models.

Data Availability: The source code for our project is available on our GitHub as well as our Google Colab. The combined dataset we used can be found here, and the oversampled PrimeVul here. A pre-trained CodeT5 model can be found here as well.

11 Contributions

11.1 Milestone 1 Work

Drew

- First team meeting where project ideas were brainstormed; contributed to creating the Google Doc for ideas and research.
- Team meeting with Prof. Bento to discuss project ideas; participated in selecting the Code Vulnerability project.
- Team meeting to discuss strategy, proficiencies, and research approach.
- Meeting with Professor Bento to discuss progress and databases.
- Wrote the Related Works section.
- Created code to run the functions on ChatGPT and record the loss.

Alex

- First team meeting where project ideas were brainstormed; helped create the Google Doc to store ideas and research.
- Team meeting to discuss strategy, proficiencies, and research approach.
- Researched the VulKG knowledge graph and wrote the abstract.
- Met with Professor Bento to discuss progress and databases.
- Researched the PrimeVul database and authored its description.

Sara

- First team meeting where project ideas were brainstormed; contributed to the Google Doc for ideas and research.
- Team meeting to discuss strategy, proficiencies, and research approach.
- Researched papers and literature related to the topic.
- Researched Python databases relevant to the project.

Bryan

- First team meeting where project ideas were brainstormed; created a Google Doc to store ideas and research.
- Team meeting with Prof. Bento to discuss project ideas; Code Vulnerability project was chosen.
- Team meeting to discuss strategy, proficiencies, and research approach.
- Researched current machine-learning cybersecurity issues; wrote the introduction; summarized the OSV database.

11.2 Milestone 2 Work

Drew

- Researched the NVD API and helped optimize its runtime.
- Researched fine-tuning of existing models for improved accuracy.
- Decreased the load time of CVSS scores from hours to minutes via a web-scraping algorithm.
- Scraped 44,000 entries in the training dataset to obtain CVSS 2.0 scores and reformatted the dataset.
- Created an algorithm to format data for Mistral fine-tuning.
- Ran fine-tuning on Mistral through a Python script (unsuccessfully).
- Met to discuss architecture and discovered Graph Neural Network (GNN) approaches.
- Researched Code Property Graphs (CPGs).
- Developed an algorithm to create graphs from code snippets and functions.
- Conducted research on Graph Neural Networks (GNNs).
- Wrote the experimentation section of the LaTeX document.

Alex

- Tested “blind guesses” on the first 1 000 entries of the training dataset.
- Investigated the Vul-LMGNN model and its GNN architecture.
- Reviewed pre-existing models (CodeT5, CodeBERT) for fine-tuning potential.
- Scraped and prepared the test dataset.
- Met to discuss overall architecture and identified Graph Neural Network approaches.
- Reviewed the ANGEL model paper.
- Wrote the Flawfinder and ANGEL sections.
- Researched the AMPLE GNN model.
- Drafted the write-up on the AMPLE model.

Sara

- Researched the NVD API and contributed to optimizing its runtime.
- Researched the cost of data breaches in recent years and how they have been handled.
- Investigated the mechanisms behind the pitfalls of ML in computer security.
- Researched further into the underlying mechanisms of those pitfalls.
- Analyzed community usage of the databases via the Papers with Code website.

Bryan

- Researched the NVD API and optimized its runtime.
- Experimented with batching and parallelizing the API script.

- Scraped 44 000 train-dataset entries to obtain CVSS 2.0 vulnerability scores and reformatted the dataset.
- Began implementing CodeT5 and performed troubleshooting.
- Met with the team to discuss architecture and discovered Graph Neural Network (GNN) approaches.
- Read and authored material on the general transformer architecture.
- Continued implementing CodeT5; measured training speeds and loss metrics.

11.3 Final Report Work

Drew

- Researched gradient boosting algorithms
- Attempted python implementation of gradient boosting
- Created a 3 layer neural network to learn from the data
- Researched weighted algorithms for balancing the dataset
- Researched deterministic algorithms for function prediction
- Researched common risk factors in C code
- Created a document of common risk factors for code
- Implemented a deterministic algorithm for function prediction
- Created code to convert json to machine readable data
- Trained a neural network on our data
- Streamlined the batch functionality of the neural network to preserve RAM
- Researched random forest architecture
- Implemented random forest on top of the neural network
- Created github functionality in the Colab
- Recorded section on my code in the video
- Developed VulnFeatureExtractor class for feature engineering
- Developed rule-based analysis in the SmartVulnerabilityDetector
- Implemented memory optimization with garbage collection and batch processing
- Created threshold optimization to balance recall and review burden
- Created weighted risk scoring system considering vulnerability types and severity
- Applied NLP techniques with text vectorization for code analysis
- Implemented memory-efficient processing for large dataset handling

Alex

- Took the basic graph neural network that Drew made and made some changes to the code
- Created code to train Word2Vec

- Ran and tested the original graph-based neural network model and discovered that it would always predict one. Discovered the issue and fixed it
- Tested the models performance on just a GCN and RGCN using only the PrimeVul dataset. Discovered very low performance and overfitting.
- Researched the RGCN convolutional type and implemented it. Better performance was observed but still high overfitting
- Researched data imbalance handling techniques, first implemented class weights with cross entropy loss. Then, discovered focal loss, a common technique used with cross entropy loss to put more attention toward minority entries in the dataset.
- Experimented with data undersampling of the majority class to address class imbalance, but did not fix the issue. Then tried oversampling, which demonstrated higher performance but not adequate enough
- Combined three popular datasets, PrimeVul, Diversevul, and BigVul to gain more representation of vulnerable samples. Observed better performance on upsampled version of this dataset
- Experimented with different L2, dropout, and learning rates to find the best results
- Implemented a learning rate scheduler that changes the learning rate when the validation accuracy plateaus
- Tested the Devign model on our datasets
- Attempted to get Joern-based graphs working with our model, but failed as my computer could not run the software
- Wrote about the GNNs on this report, researched and wrote about Devign, and wrote about our datasets.
- Created huggingface databases for our oversampled PrimeVul dataset, along with the combined dataset.

Sara

- Researched prior literature on ChatGPT's evaluation of code vulnerabilities
- Analyzed the architecture and training methodology of the Devign model. Tried to implement the model, some errors were encountered
- Conducted a comparative analysis of previous LLM-based approaches for code vulnerability detection
- Defined and refined the goal for the ChatGPT 4o-mini model (changed the output format from assessing a score to giving a binary output)
- Researched the functioning of the API key and worked on my personal ChatGPT page, adding the credits in order to originate the secret keys to work on
- Analyzed the reduced dataset extracted from PrimeVul and used to implement the ChatGPT model
- Created a first script to interact with OpenAI's API for automated vulnerability detection
- Implemented the code to automate ChatGPT's predictions and the one used to assess the model's evaluations
- Integrated error handling to manage potential API failures and ensure robustness
- Changed the initial prompt given to ChatGPT in order to compare the performances of the two models. Being this last one better, the final version of the script was defined and confirmed
- Recorded introduction part and section on my model in the video presentation

- Wrote about the ChatGPT model in this report, with an additional focus on prior literature and previous results

Bryan

- Meeting with Wei Qiu and TA Elliot about HPC and beginning transferring model to train on HPC.
- Altered CodeT5 model script to output into a file instead of terminal
- Implemented a learning rate scheduler
- Implemented the prediction and true vulnerability classification for each step to ensure the model was properly working
- Implemented a "try. . . catch" to deal with data entries that do not have a properly formatted e.o.s token which caused the model to fail
- refined the script to use the pandas library instead of ijson
- Ran T5-train-bin.py script on HPC and checked log file
- Created a test script and looked into the sklearn python library and implemented it for the test script to get the metrics
- Adds list comparisons to the test script to ensure that the test script went through all the data points
- Combined the sections of the paper and refined the comments in the abstract, introduction and transformer section
- Wrote the model evaluation section for the transformer model
- refined and added more detail to the feature that separates CodeT5 from the normal transformer model
- Created a hugging face database for the Trained CodeT5 model

References

[1] Area under the Curve — ibm.com. https://www.ibm.com/docs/en/spss-statistics/30.0.0?topic=schemes-area-under-curve. [Accessed 11-05-2025].

[2] Patricia S. Abril and Robert Plant. The patent holder's dilemma: Buy, sell, or troll? *Communications of the ACM*, 50(1):36–44, January 2007.

[3] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Pitfalls in machine learning for computer security. *Communications of the ACM*, 67(11):92–101, November 2024.

[4] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection, 2023.

[5] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we?, 2024.

[6] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Michael Fu, Chakkrit Tantithamthavorn, Van Nguyen, and Trung Le. Chatgpt for vulnerability detection, classification, and repair: How far are we?, 2023.

[8] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Erik Antelman, Alan Mackay, Marc W. McConley, Jeffrey M. Opper, Peter Chin, and Tomo Lazovich. Automated software vulnerability detection with machine learning, 2018.

[9] IBM. Cost of a data breach report 2024, 2024. Accessed: 2025-05-09.

[10] Dun Jin, Chengwan He, Quan Zou, Yan Qin, and Boshu Wang. Source code vulnerability detection based on joint graph and multimodal feature fusion. *Electronics*, 14:975, 02 2025.

[11] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[12] Chen Liang, Qiang Wei, Jiang Du, Yisen Wang, and Zirui Jiang. Survey of source code vulnerability analysis based on deep learning. *Computers & Security*, 148:104098, 2025.

[13] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection, 2018.

[14] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation. *arXiv preprint*, February 2024. arXiv:2402.19173.

[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

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

[17] Xin Peng, Shangwen Wang, Yihao Qin, Bo Lin, Liqian Chen, and Xiaoguang Mao. Keep it simple: Towards accurate vulnerability detection for large code graphs, 2024.

[18] Luana Ruiz, Fernando Gama, and Alejandro Ribeiro. Gated graph recurrent neural networks. *IEEE Transactions on Signal Processing*, 68:6303–6318, 2020.

[19] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks, 2017.

[20] The Joern Team. Joern: A static analysis platform for codebase exploration and vulnerability detection, 2020. Accessed: 2025-05-07.

[21] Richard E. Turner. An introduction to transformers, 2024.

[22] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[23] Yue Wang and H. Hoi Steven C.˙Codet5: The code-aware encoder-decoder based pre-trained programming language models. https://www.salesforce.com/blog/codet5/, September 2021. Salesforce Blog Post.

[24] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. https://github.com/salesforce/CodeT5, 2021. GitHub repository.

[25] Yue Wang, Weishi Wang, Shafiq Joty, and H. Hoi Steven C.˙Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021. Accepted to EMNLP 2021.

[26] Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M. Zhang, and Qing Liao. Vulnerability detection with graph simplification and enhanced graph representation learning, 2023.

[27] David Wheeler. Flawfinder.

[28] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs, 2014.

[29] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, 2019.