Vulnerability Detection for Program Binaries with Graph Neural Networks

Mike Kadoshnikov Nitasha Fazal Jacob Tanner Dillon Shaffer Layton McCafferty

***Abstract*—Detecting vulnerabilities in software is a critical task for enhancing security; however, traditional—methods such as assembly or symbolic execution—require significant manual effort and are difficult to generalize. Code pre-trained models (Code PTMs) also often struggle with generalization, as they primarily rely on syntactic patterns. In this paper, we explore the potential of Graph Neural Networks (GNNs) to overcome these limitations and introduce WIMFG-VAn (WIMFG is Matt's Favorite Group's Vulnerability ANalyzer), a novel approach that leverages GraphSAGE (SAGE Convolution) for scalable node embedding. To enhance the model's effectiveness we integrate Abstract Syntax Tree (AST) graphs that capture both data dependencies and control flow, ensuring that the model learns from meaningful patterns in source code. Our approach outperforms existing baseline models, achieving a 100% F1 score in vulnerability detection and demonstrating significant improvements in the F1 score for vulnerability class identification compared to Devign, ReVeal, CodeBERT, GraphCodeBERT, and UniXcoder for the Juliet Software Assurance Reference Dataset (SARD).**

***Index Terms*—Graph Neural Network (GNN), SAGE-Conv, Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Dependency Graph (DDG), Data Augmentation, Masking**

I. Introduction

Detecting vulnerabilities using assembly or symbolic execution requires significant manual effort via coding and providing inputs, making it challenging to generalize across different types of software. On the other hand, Code PTMs, such as CodeBERT, have been increasingly applied for automatic vulnerability detection. While Code PTMs have shown promising results in recent years, they also face limitations with generalization. These models primarily rely on syntax and often struggle to capture the deeper semantics, relationships, and context of the code, which are crucial for detecting complex vulnerabilities.

In recent years, Graph Neural Networks (GNNs) have shown promising results across various fields. GNNs model code structures (e.g., Abstract Syntax Trees or control/data flow graphs), capturing both syntax and semantic relationships more effectively than sequence-based models. They also generalize better than Code PTMs (e.g., CodeBERT) in tasks where understanding semantic relationships is crucial. However, GNNs also pose several challenges, especially when dealing with larger codebases. Scaling GNNs to very large graphs becomes difficult due to high memory usage and aggregating information from distant nodes can lead to inefficiencies.

To address these challenges, we introduce WIMFG-VAn, a novel approach that leverages SAGE Convolution (GraphSAGE) for scalable node embedding by aggregating information from neighboring nodes. To enrich our model and prevent it from learning superficial mappings from source code, we incorporate Abstract Syntax Tree (AST) graphs, which capture both data dependency and control flow information. The key insight is to enhance the input data fed into the neural network, ensuring that it learns meaningful patterns rather than just superficial label mappings. Additionally, SAGE Convolution provides our model with the flexibility to scale by adjusting the aggregation of distant node layers.

WIMFG-VAn outperformed in detecting vulnerabilities, achieving a 100% F1 score and demonstrating superior effectiveness with an improvement in the F1 score by 4% compared to Devign, ReVeal, CodeBERT, GraphCodeBERT, and UniXcoder for Juliet dataset, discussed in Table 4.

Our contributions are summarized as follows:

- A method of aggregating ASTs, CFGs, and DDGs into a single detailed graph
- A binary classifier that determines if a program is vulnerable or non-vulnerable
- A CWE classifier that determines the type(s) of vulnerabilities present in a program
- An analysis of the effectiveness of GNNs for detecting vulnerabilities in program binaries

II. Related Work

The existing work areas that can be discussed in relation to this paper are vulnerability detection, code property graphs, and graph neural networks. The concept of automatic detection of CWE's in software has been around for some time. Existing solutions not only rely on human expertise to define these vulnerabilities, but are also prone to errors such as false-positives and false-negatives—where tools incorrectly report results. Programs often contain many simple or primitive vulnerabilities that only become exploitable when chained with other vulnerabilities. In sufficiently complex software, this fact often means that exploitability is the rule rather than the exception.

A. Vulnerability Detection

To overcome the challenges of automatic vulnerability detection, Ruitong Liu et. al. proposed *Vulnerability Deep Pecker* (VulDeePecker), a deep learning-based system for vulnerability detection [1]. The main obstacles with their approach was the representation of software for neural networks and the selection of an appropriate neural network. Code gadgets were used to handle software representation, and a *Bidirectional Long Short-Term Memory* (BLSTM) was chosen as the neural network model. The *National Vulnerability Database* (NVD) and the *Software Assurance Reference Dataset* (SARD) are used to train the neural networks. Code gadgets were then converted into vector representations and fed into the BLSTM for training. VulDeePecker shows promising results in terms of the 'zero-one loss' criterion and precision.

In another research paper, Louis Russell and Hamilton used deep learning to automate vulnerability detection by using open-source C and C++ code to develop a large-scale, function-level vulnerability detection system with machine learning [2]. After generating useful features from the raw source code of each function provided—using a lexer design—they removed the duplicates during the data accumulation process. Then, Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) were deployed for function-level source vulnerability classification and decision trees were applied to classify the CWE labels. Their design using CNN models outperformed the RNN models and overall showed promising results. The use of a lexer and machine learning models reduced their code compilation time. Additionally, the flexibility of machine learning methods allowed the models to be tuned for desired recall and precision.

B. Graph Neural Networks

In another paper, Vul-LMGNN is introduced, a novel vulnerability detection approach that combines the strengths of both Code Pre-trained Language Models (codePLM) and Graph Neural Networks [3]. Here, the source code is first converted into Program Dependency Graphs (PDG), Control Flow Graphs (CFG), and Abstract Syntax Trees (AST). The approach emphasizes combining both sequence-based and graph-based methods. To achieve this, Vul-LMGNN constructs a Code Property Graph (CPG) that merges the AST, CFG, and PDG representations.

Graphs were initialized with node embedding using a pre-trained model, specifically CodeBERT, which helps capture semantic information from source code. A Gated Graph Neural Network (GGNN) is then deployed for vulnerability detection. During pre-processing, the source code was transformed into various graph representations, such as an AST, CFG, PDG, and CPG. Subsequently, the nodes and edges in these graphs were converted into vectors, which were then fed into the GNN model. Since GNNs often struggle to capture contextual relationships between distantly connected nodes, this limitation is mitigated by utilizing a pre-trained code language model to initialize the embedding of nodes in the code graph—these bolster the model's ability to recognize these relationships.

For the experiments, the new release of the DiverseVul dataset and the Devign dataset, both of which comprise real-world function examples from GitHub are used. These datasets were filtered by specific CWE categories and the model was evaluated by classifying these CWE categories in the test set. Results showed that the model outperformed all baseline models, achieving an accuracy of 93.06% and an F1-score of 23.54%.

III. WIMFG-VAN

For this project, we introduce WIMFG-Van (WIMFG-Van is Matt's Favorite Group's Vulnerability ANalyzer) which leverages program semantics in a graph representation as input into a neural network. Our aim is to detect various CWE's potentially present in a program.

A. Questions

1. Given only compiled binary files, can we correctly and accurately identify vulnerabilities in binaries?
2. How do our methods performance metrics compare to the performance metrics in the Source Code Vulnerability Detection Paper [3]?
3. How will changes in the input graphs affect the performance metrics of the machine learning model?

B. Methodology

In this section, we will discuss the main components of WIMFG-Van that we developed. A high-level architecture of the model is presented in Figure 1

1) D: ata Acquisition For this project, we used the Juliet dataset. We looked at a few different candidate datasets. We decided against using datasets that were unlabeled; these included datasets that included vulnerable code units that did not have CWE labels and datasets that listed CWE's but pointed to a pinned commit of a project with no specifics about which functions were vulnerable. We found a few datasets that fell outside of these criteria, including NIST's Juliet SARD, Draper VDISC, MegaVul, DiverseVul, VulinOSS, VulDeePecker, and Wild-C datasets. We chose Juliet as we deemed it large enough for our analysis and easy to ingest into our tooling. We excluded the other datasets due to time

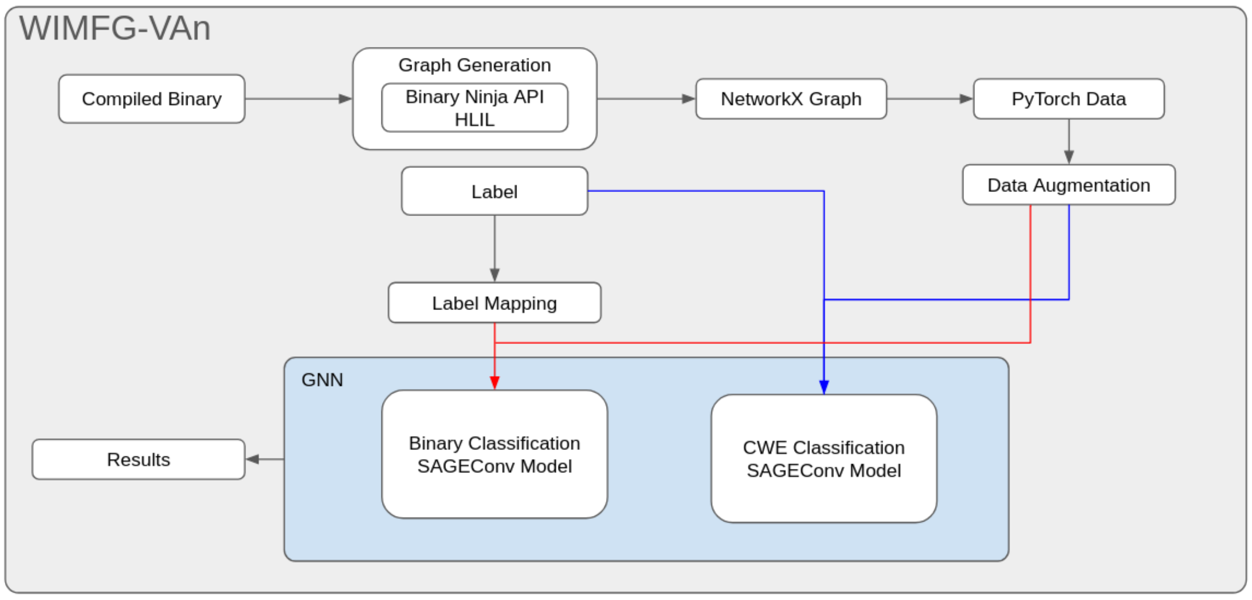


Fig. 1: Flowchart of WIMFG-VAn Architecture

constraints for training and/or limited CWE enumeration in each respective dataset.

2) Graph Generation: The first component of WIMFG-Van is Graph Generation. This component utilizes binaries organized into folders based on their respective CWE class labels. We used the Binary Ninja API to generate high-level intermediate language (HLIL) CFGs and DDGs. These graphs are simplified abstract syntax representations that include function nodes, but reduce the importance of these nodes. Instead, they consist of HLIL statements as nodes, with CFG and DDG edges defining their structure. Using NetworkX, these graphs were combined into a single graph and saved in the same folder structure as the binary source code. Figure 2 illustrates an example of graph generation from source code using Binary Ninja.

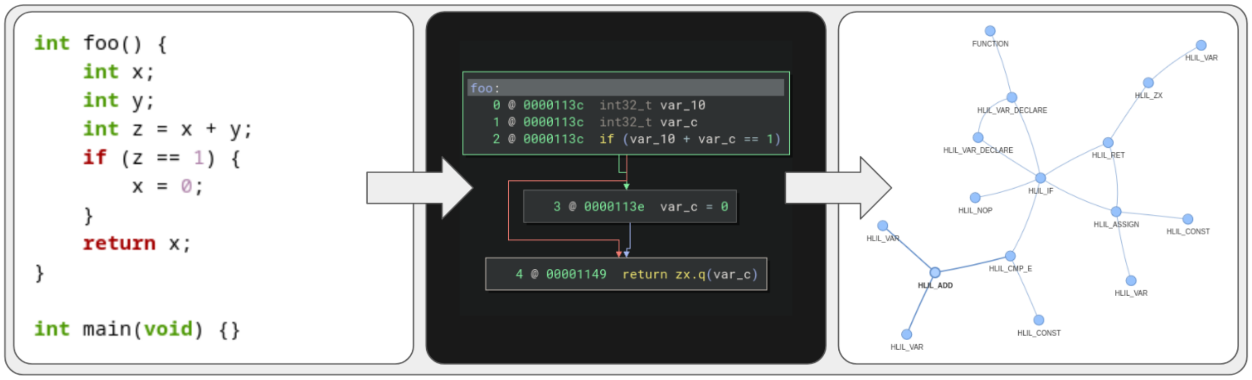
3) Data PreProcessing: The graphs generated in the previous component were then imported into tensors using the PyTorch Geometric library [4]. The data is split into training, testing, and validation partitions—we used a 70/20/10 split. A data augmentation step is introduced to enhance the model's robustness and generalizability. In this step, noise is added to the training and validation datasets, specifically to the x-attributes of the tensor graphs. Additionally, masking is performed on the graph data to improve further robustness, where certain parts of the graph (e.g., nodes, edges, or features) are selectively hidden. This combined approach of adding noise and masking helped the model learn to handle variations and missing information effectively, resulting in improved performance on unseen data.

4) Graph Neural Network (GNN) : We employed two GNN models to detect vulnerabilities in the source code. The first model was designed to identify the presence of vulnerabilities, while the second focused on classifying the types of vulnerabilities. Both models shared a similar architecture, utilizing a SAGE convolution network, which emphasizes node embeddings over edge features. Each model consisted of multiple SAGE convolution layers, followed by four dropout layers with a dropout rate of 0.5, ensuring robust training. The models were trained for 50 epochs each, using ReLU activation functions in the hidden layers and cross-entropy as the loss function. This architecture effectively captured the patterns needed for both detection and classification tasks.

Due to the resource constraints required to train neural network technologies, some form of high-performance computing was required to train our models efficiently. As such, computational efforts were performed on the Tempest High Performance Computing System, operated and supported by the University Information Technology Research Cyberinfrastructure at Montana State University.[5]

C. Experiment and Results

Model	F1 Score
Devign	57.71
ReVeal	57.86
CodeBERT	5.36
GraphCodeBERT	31.22
UniXcoder	61.36
*WIMFG-VAn	65



```
int foo() {
    int x;
    int y;
    int z = x + y;
    if (z == 1) {
        x = 0;
    }
    return x;
}

int main(void) {}
```

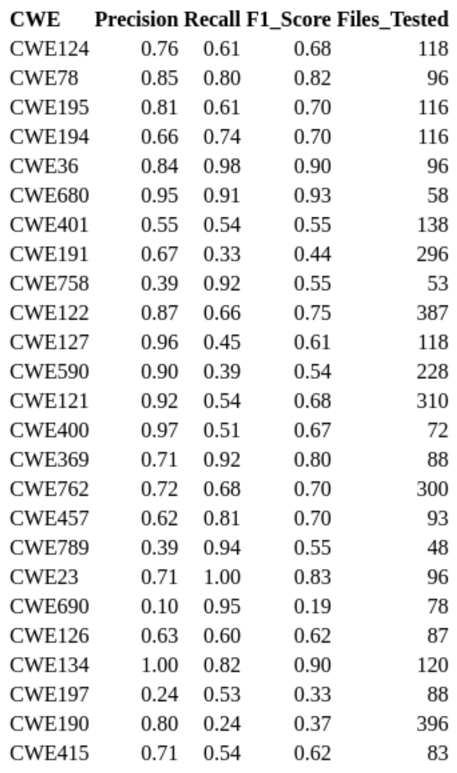
```
foo:
   0 @ 0000113c  int32_t var_10
   1 @ 0000113c  int32_t var_c
   2 @ 0000113c  if (var_10 + var_c == 1)

      3 @ 0000113e  var_c = 0

    4 @ 00001149  return zx.q(var_c)
```

Fig. 2: Example of Graph Generation with Binary Ninja and NetworkX

TABLE I: F1 score comparison on Juliet dataset. [6] $*$ Top 25 classes from the Juliet dataset according to the number of graphs. [Table II]

As discussed in the previous section, our experimental setup involved running the SAGEConv network for 50 epochs on the Juliet dataset. The results of our experiments are presented in Figure 3 as numerical data and Figure 4 provides a visual comparison of the F1 scores for each class. We designed our experiments to address the research questions of our project. Our results demonstrate the ability to identify vulnerabilities in binaries and show that our approach outperformed baseline models, including Devign, ReVeal, CodeBERT, GraphCodeBERT, and UniXcoder.

Table I compares our results with other baseline models, highlighting improved performance in terms of the average F1 score. Our findings suggest a structural difference between vulnerable and non-vulnerable programs, addressing our third research question that states structural changes and number of input graphs affect performance metrics. Figure 5 shows the comparison of precision and recall for our project, helping to assess performance in terms of vulnerability identification while minimizing false positives and false negatives.



CWE	Precision	Recall	F1_Score	Files_Tested
CWE124	0.76	0.61	0.68	118
CWE78	0.85	0.80	0.82	96
CWE195	0.81	0.61	0.70	116
CWE194	0.66	0.74	0.70	116
CWE36	0.84	0.98	0.90	96
CWE680	0.95	0.91	0.93	58
CWE401	0.55	0.54	0.55	138
CWE191	0.67	0.33	0.44	296
CWE758	0.39	0.92	0.55	53
CWE122	0.87	0.66	0.75	387
CWE127	0.96	0.45	0.61	118
CWE590	0.90	0.39	0.54	228
CWE121	0.92	0.54	0.68	310
CWE400	0.97	0.51	0.67	72
CWE369	0.71	0.92	0.80	88
CWE762	0.72	0.68	0.70	300
CWE457	0.62	0.81	0.70	93
CWE789	0.39	0.94	0.55	48
CWE23	0.71	1.00	0.83	96
CWE690	0.10	0.95	0.19	78
CWE126	0.63	0.60	0.62	87
CWE134	1.00	0.82	0.90	120
CWE197	0.24	0.53	0.33	88
CWE190	0.80	0.24	0.37	396
CWE415	0.71	0.54	0.62	83

Fig. 3: CWE Classifier Evaluation Metrics

IV. Discussion

WIMFG-VAn has proven to be an important milestone in our efforts to develop a generalized automated vulnerability detection tool. Past research in this area has paved the way for our project, enhanced our understanding, and helped us identify areas for improvement. We thoroughly enjoyed working on this project and learned new concepts and tools that will be valuable for future work. One advantage of using graphs, instead of tokens from source code is that it provides flexibility to scale models by adding more layers of neighboring nodes in graph neural networks (such as SAGE Convolution). However, these models are computationally expensive. To address

this, we utilized Tempest—the HPC cluster at MSU—for training and experimentation. [5]. Additionally, we gained experience using the Binary Ninja API and Graph Neural Networks.
A key finding, which is also a focus of our future work, is that deep learning models require rich and diverse data for accurate predictions. In this project, we focused on reduced ASTs, where function-node edges were reduced. Moving forward, we aim to address this limitation by incorporating more details regarding program graphs and edge types into the graphs and working with more extensive datasets.

V. Conclusion

In this project, we introduced WIMFG-VAn, a novel tool for code vulnerability detection that leverages code graphs and graph neural networks. By integrating CFGs and DDGs into ASTs and employing data augmentation strategies, we developed an effective binary analysis tool for detecting vulnerabilities in binary programs. Our approach to CWE identification also showed promising results.
Our models were trained and validated on the Juliet dataset. In future work, we plan to use other datasets and further refine the representation of ASTs. Our model achieved 100% accuracy in vulnerability detection and demonstrated improved performance in CWE detection.

VI. Limitations

Due to time constraints and limited data available, our experiments focused on only 25 classes from the Juliet dataset.[Table II] These classes were selected based on the number of graphs, where only classes containing at least 1,000 graphs were included in the training. The other limitation related to this project is that we used reduced ASTs, these did not properly contain all AST information as it would emphasize the importance of semantically unimportant nodes.

CWE Label	**Description**
CWE-124	Buffer Underwrite
CWE-78	OS Command Injection
CWE-195	Signed to Unsigned Conversion Error
CWE-194	Unexpected Sign Extension
CWE-36	Relative Path Traversal
CWE-680	Integer Overflow to Buffer Overflow
CWE-401	Memory Leak
CWE-191	Integer Underflow
CWE-758	Reliance on Undefined Behavior
CWE-122	Heap-based Buffer Overflow
CWE-127	Buffer Over-read
CWE-590	Free of Invalid Pointer
CWE-121	Stack-based Buffer Overflow
CWE-400	Uncontrolled Resource Consumption
CWE-369	Divide by Zero
CWE-762	Mismatched Memory Management Routines
CWE-457	Use of Uninitialized Variable
CWE-789	Uncontrolled Memory Allocation
CWE-23	Path Traversal
CWE-690	NULL Dereference
CWE-126	Buffer Over-read
CWE-134	Format String Vulnerability
CWE-197	Numeric Truncation Error
CWE-190	Integer Overflow or Wraparound
CWE-415	Double Free

TABLE II: A list of CWEs and their descriptions.

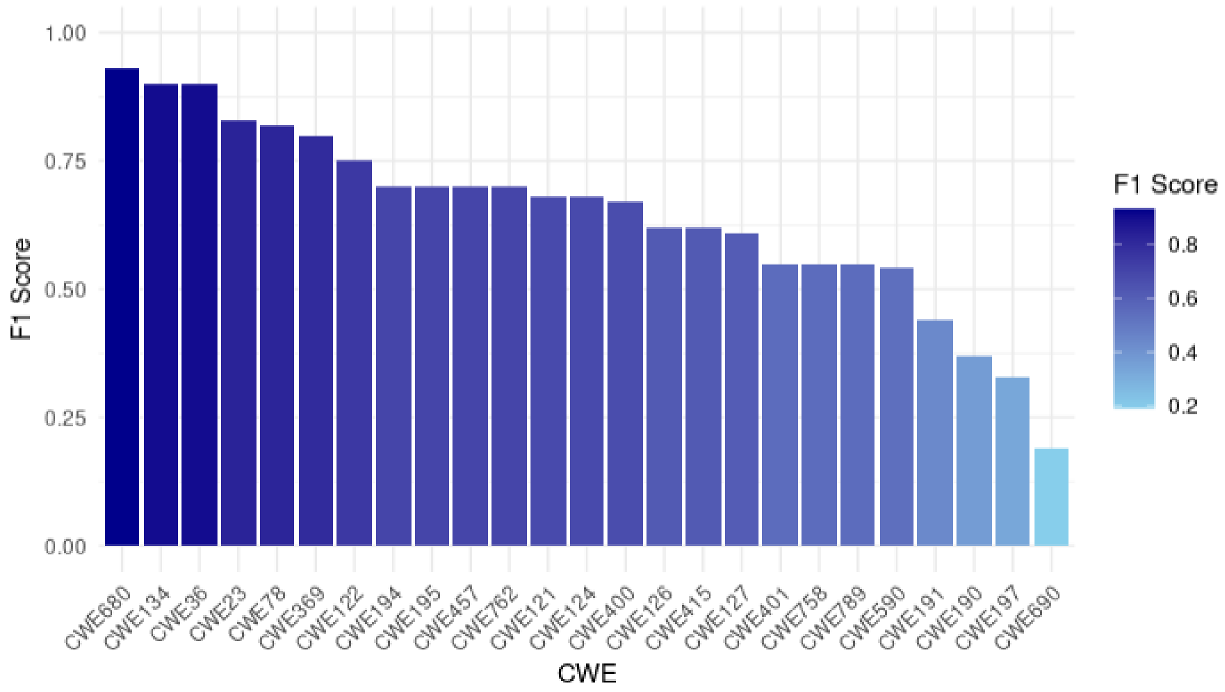


Fig. 4: F1 Scores for Each CWE

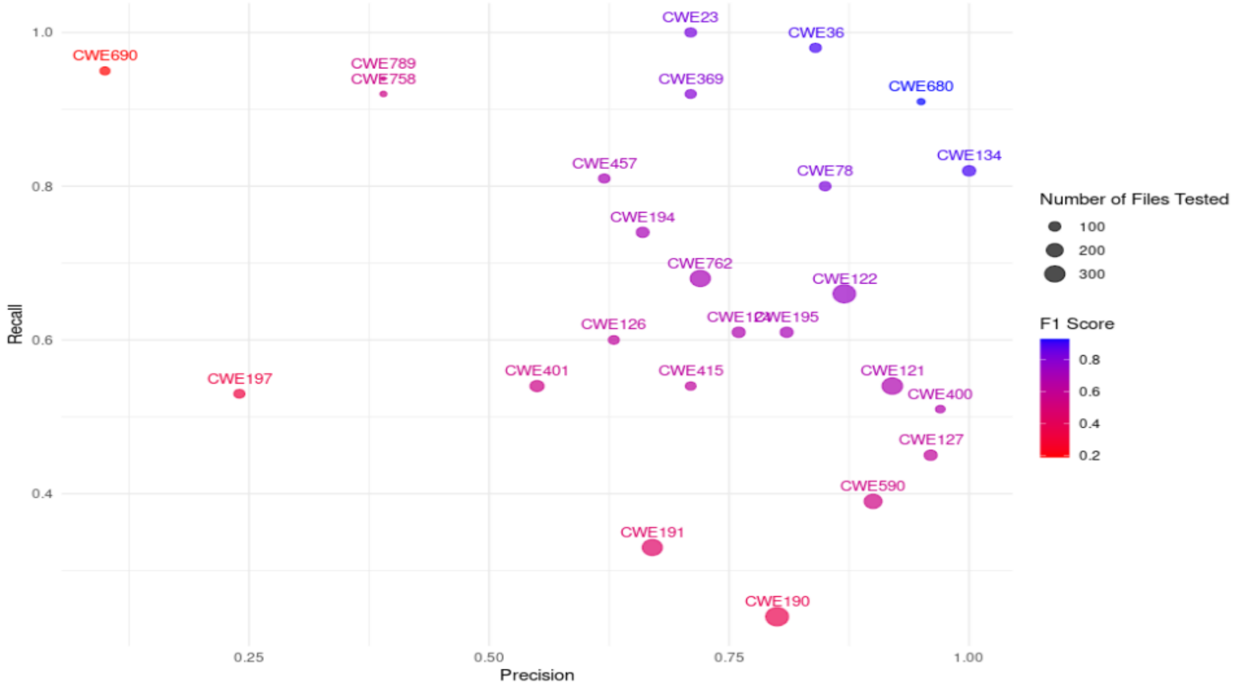


Fig. 5: CWE Precision vs Recall

References

[1] Zhen Li et al. “VulDeePecker: A Deep Learning-Based System for Vulnerability Detection”. In: *Proceedings 2018 Network and Distributed System Security Symposium*. NDSS 2018. Internet Society, 2018. DOI: 10.14722/ndss.2018.23158. URL: http://dx.doi.org/10.14722/ndss.2018.23158.

[2] Rebecca L. Russell et al. *Automated Vulnerability Detection in Source Code Using Deep Representation Learning*. 2018. arXiv: 1807.04320 [cs.LG]. URL: https://arxiv.org/abs/1807.04320.

[3] Ruitong Liu et al. *Source Code Vulnerability Detection: Combining Code Language Models and Code Property Graphs*. 2024. arXiv: 2404.14719 [cs.CR]. URL: https://arxiv.org/abs/2404.14719.

[4] Matthias Fey and Jan Eric Lenssen. *Fast Graph Representation Learning with PyTorch Geometric*. 2019. arXiv: 1903.02428 [cs.LG]. URL: https://arxiv.org/abs/1903.02428.

[5] Matthias Fey and Jan Eric Lenssen. *Computational efforts were performed on the Tempest High Performance Computing System, operated and supported by University Information Technology Research Cyberinfrastructure at Montana State University.* URL: https://www.montana.edu/uit/rci/tempest/.

[6] Xiaohu Du et al. “Generalization-Enhanced Code Vulnerability Detection via Multi-Task Instruction Fine-Tuning”. In: *Findings of the Association for Computational Linguistics: ACL 2024*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 10507–10521. DOI: 10.18653/v1/2024.findings-acl.625. URL: https://aclanthology.org/2024.findings-acl.625.