Project Title: AI-Based Vulnerability Prediction in Source Code

---

Short Project Description:

The system combines static code analysis with machine learning to identify insecure coding patterns and generate actionable reports.

Component	Role
Source Code Collector	Gathers code repositories for analysis
Static Feature Extractor	Extracts semantic code features (data flows, control flows, etc)
AI Prediction Engine	Predicts vulnerabilities from code patterns
False Positive Filter	Reduces noise in results
Report Generator	Compiles suspected vulnerabilities into structured findings

---

Components Details:

1. **Source Code Collector**:
    - Downloads code from GitHub, GitLab, internal repositories, etc.
2. **Static Feature Extractor**:
    - Parses code into ASTs (Abstract Syntax Trees).
    - Extracts features:
        - Function call graphs
        - Taint analysis paths
        - Etc
3. **AI Prediction Engine**:
    - Applies trained machine learning models:
        - Deep Learning (RNNs, Transformers, etc)
        - Classical ML (SVM, Random Forests, etc)
    - Predicts which code snippets are vulnerable.
4. **False Positive Filter**:
    - Applies thresholding, post-processing to remove unlikely results.

5. **Report Generator**:
    - Prepares lists of suspected vulnerabilities with location and severity.

Overall System Flow:

- Input: Source Code Repositories
- Output: List of probable vulnerabilities
- Focus on **predictive static code analysis using AI**.

Internal Functioning of Each Module:

1. Source Code Collector

- **Tools (e.g.)**: GitHub API, GitLab API, direct downloads, etc.
- **Function**:
    - Clone repositories or download zipped source code.
    - Normalize file formats: `.c`, `.cpp`, `.java`, `.py`, etc.
    - Metadata tagging (repo name, commit ID, authorship).

---

2. Static Feature Extractor

- **Key Techniques**:
    - **Abstract Syntax Tree (AST) Generation** using parsers like `tree-sitter`, `clang-ast`.
    - **Control Flow Graph (CFG)** generation for code behavior modeling.
- **Features extracted**:
    - Function calls (data sinks, data sources, etc)
    - Variable assignments (tainted flow analysis, etc)
    - API usage patterns (e.g., `strcpy()` risky usage, etc)
- **Advanced**: Use **Graph Neural Networks (GNNs)** to embed ASTs and CFGs into feature vectors.

---

3. AI Prediction Engine

- **Model Choices**:
    - **Traditional**: Random Forests, SVMs (good for small datasets), etc
    - **Deep Learning**:
        - RNNs (sequence modeling of code tokens, etc)
        - Transformers (e.g., CodeBERT, GraphCodeBERT for source code understanding. etc)
- **Input to Model**:

    - Feature vectors from AST/CFG.
- **Output**:
    - Probability score per function: `P(vulnerable)`

---

4. False Positive Filter

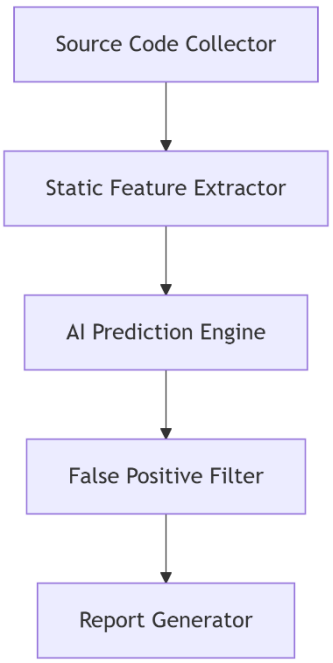
- **How (e.g.)**:
    - Post-processing thresholds:
        - Only flag if `P(vulnerable) > 0.7`
    - Context validation:
        - Flag only if sensitive API appears nearby (e.g., `exec()`, `system()`).

---

5. Report Generator

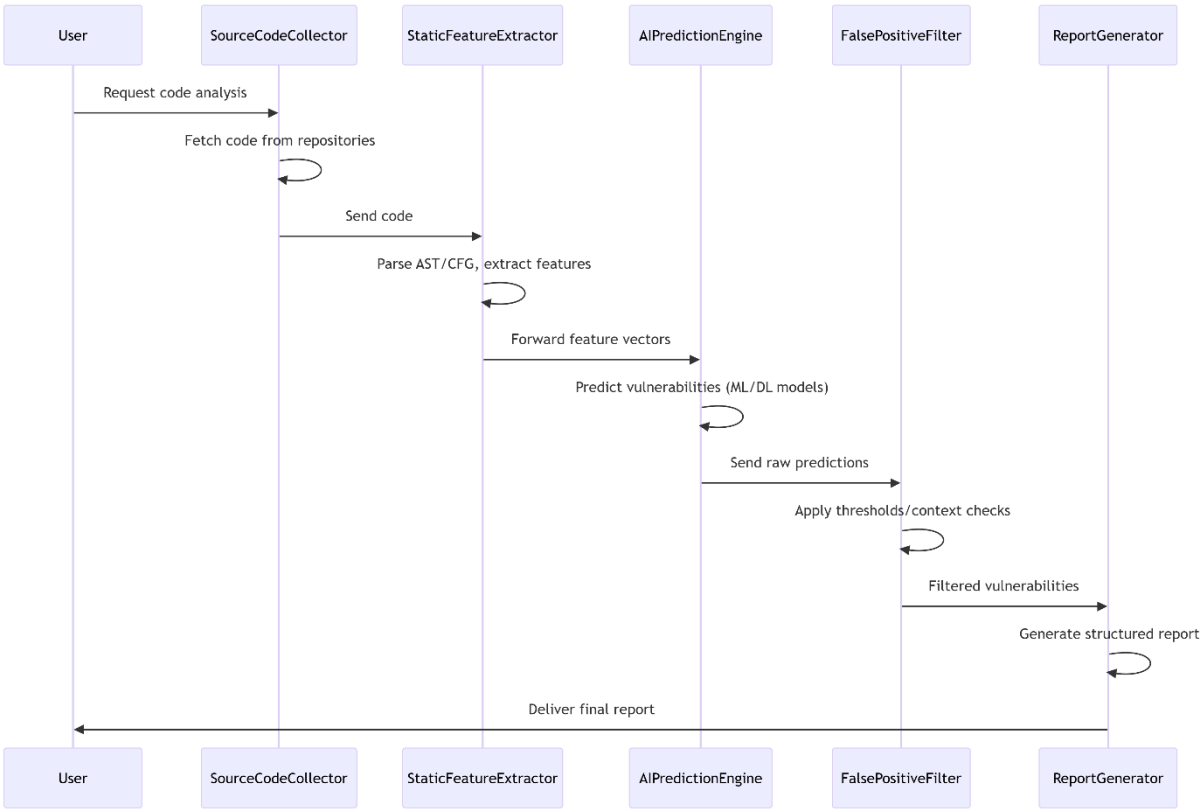
- **Structure**:
    - Lists:
        - File
        - Line number
        - Function name
        - Vulnerability type (prediction class)
        - Probability score
        - Suggested remediation (static recommendations)

Component Diagram



- The **Source Code Collector** retrieves code and sends it to the **Static Feature Extractor**.
- The **Static Feature Extractor** parses code into ASTs/CFGs and extracts vulnerability-relevant features.
- The **AI Prediction Engine** uses ML models to predict vulnerabilities.
- The **False Positive Filter** refines predictions by removing low-confidence results.
- The **Report Generator** compiles validated findings into a user-ready report.

Sequence Diagram



- The **User** initiates the process, triggering code collection and parsing.
- The **Static Feature Extractor** generates code representations and sends features to the **AI Prediction Engine**.
- Predictions are filtered to reduce noise, and the refined results are formatted into a report by the **Report Generator**.

Detailed Project Description: AI-Based Vulnerability Prediction in Source Code

An AI-driven tool for predicting vulnerabilities in source code. The tool leverages static code analysis and machine learning to identify insecure patterns, prioritize risks, and generate actionable reports.

---

1. System Overview

The tool analyzes source code repositories to predict vulnerabilities (e.g., buffer overflows, SQL injection, etc) using static code features and machine learning models. It focuses on **predictive analysis** without code execution, prioritizing high-risk findings while minimizing false positives.

Key Components

1. **Source Code Collector**
2. **Static Feature Extractor**
3. **AI Prediction Engine**
4. **False Positive Filter**
5. **Report Generator**

---

2. Component Design & Implementation

2.1 Source Code Collector

Functionality:
Retrieves code from public/private repositories and normalizes it for analysis.

Implementation Steps (e.g.):

1. **Repository Integration**:

    - **Public Repos**: Use GitHub API (`PyGithub`) and GitLab API (`python-gitlab`) to fetch code.
    - **Private Repos**: Authenticate via OAuth2 tokens or SSH keys.
2. **Code Download**:
    - Clone repositories using `GitPython` or download ZIP archives.
    - Example:
      ```
      from git import Repo
      Repo.clone_from("https://github.com/user/repo.git", "/local/path")
      ```
3. **Normalization**:
    - Organize files by language (C, Java, Python, etc) and filter non-code files (e.g., images).
    - Extract metadata (commit history, author, branch) using `git log`.

Output:

- Local directory with structured code and metadata (JSON).

Tools:

- `PyGithub`, `python-gitlab`, `GitPython`.

2.2 Static Feature Extractor

Functionality:
Parses code into structured representations (ASTs, CFGs, etc) and extracts vulnerability-relevant features.

Implementation Steps (e.g.):

1. **Parsing**:
    - **AST Generation**: Use `tree-sitter` (multi-language support) or `Clang` (C/C++).
      ```
      from tree_sitter import Parser, Language
      parser = Parser()
      parser.set_language(Language('build/my-languages.so', 'python'))
      tree = parser.parse(bytes(source_code, "utf8"))
      ```

    - **CFG Generation**: Use `Code2Flow` or derive from ASTs using graph traversal.
2. **Feature Extraction**:
    - **Data Flow Analysis**: Track tainted inputs (e.g., user input → `strcpy()`).
    - **Control Flow Analysis**: Identify risky loops or unreachable code.
    - **API Usage**: Flag insecure functions (`exec()`, `system()`, `strcpy`).
3. **Embedding**:
    - Convert ASTs/CFGs into feature vectors using **Graph Neural Networks (GNNs)** (e.g., `PyTorch Geometric`).

Output:

- Feature vectors (JSON/CSV) for ML model input.

Tools:

- `tree-sitter`, `Clang`, `NetworkX`, `PyTorch Geometric, etc`.

2.3 AI Prediction Engine

Functionality:
Predicts vulnerabilities using ML models trained on code patterns.

Implementation Steps (e.g.):

1. **Model Selection**:
    - **Traditional ML**: Scikit-learn's `RandomForestClassifier` or `SVM` for small datasets, etc.
    - **Deep Learning**:
        - **RNNs/Transformers**: Use `HuggingFace Transformers` with pre-trained models like `CodeBERT`.
        - **GNNs**: Train on AST/CFG embeddings using `PyTorch`.
        - **Etc**.
2. **Training Data**:
    - Use labeled datasets:

        - **SARD**: Synthetic C/C++ vulnerabilities.
        - **BigVul**: Real-world vulnerabilities from GitHub.
        - **Etc**.
    - Augment data by injecting synthetic vulnerabilities into clean code.
3. **Training Pipeline**:
    - Preprocess data: Tokenize code, split into training/validation (80/20).
    - Train models using frameworks like `TensorFlow` or `PyTorch`.
    - Optimize hyperparameters (learning rate, batch size) via grid search.

Output:

- Vulnerability probability scores (0–1) per code snippet.

Tools:

- `Scikit-learn`, `TensorFlow`, `PyTorch`, `HuggingFace Transformers, etc`.

2.4 False Positive Filter

Functionality:
Reduces noise by filtering low-confidence predictions.

Implementation Steps (e.g.):

1. **Thresholding**:
    - Discard predictions with scores < 0.7.
2. **Context Validation**:
    - Check proximity to sensitive APIs (e.g., `exec()` near user input).
    - Ignore test files and comments using regex (`^#` or `//`).
3. **Secondary Validation Model**:
    - Train a logistic regression classifier to validate predictions.

Output:

- High-confidence vulnerabilities (e.g., P(vulnerable) ≥ 0.7).

Tools:

- `Scikit-learn` for logistic regression, regex for pattern matching.

2.5 Report Generator

Functionality:
Compiles findings into structured reports with remediation guidance.

Implementation Steps (e.g.):

1. **Data Aggregation**:
    - Merge predictions with code metadata (file, line number, etc).
2. **Template Design**:
    - **HTML**: Use `Jinja2` for dynamic tables and filters.
    - **SARIF**: Generate standardized output for CI/CD tools (GitHub Code Scanning).
    - **PDF**: Convert HTML to PDF using `WeasyPrint`.
3. **Remediation Suggestions**:
    - Provide code fixes (e.g., "Replace `strcpy` with `strncpy`").

Output:

- Final report with vulnerability type, location, severity (CVSS-based), and fixes.

Tools:

- `Jinja2`, `SARIF SDK`, `WeasyPrint`.

---

3. Technology Stack (e.g.)

- **Code Parsing**: `tree-sitter`, `Clang`, `Code2Flow, etc`.
- **ML Frameworks**: `Scikit-learn`, `TensorFlow`, `PyTorch`, `HuggingFace, etc`.

- **Feature Extraction**: `NetworkX`, `PyTorch Geometric, etc`.
- **Reporting**: `Jinja2`, `SARIF SDK`, `WeasyPrint, etc`.

---

4. Evaluation & Validation

1. **Performance Metrics**:
    - **Precision/Recall**: Measure against labeled datasets.
    - **F1-Score**: Balance precision and recall.
    - **False Positive Rate**: Calculate invalid findings post-filtering.
    - **Etc**.
2. **Baseline Comparison (e.g.)**:
    - Compare with tools like **SonarQube** or **Checkmarx**.
3. **User Testing**:
    - Validate reports with developers on open-source projects (e.g., Apache HTTP Server, etc).

---

5. Development Roadmap (e.g.)

1. **Phase 1**: Build Source Code Collector and Static Feature Extractor.
2. **Phase 2**: Train and validate ML models.
3. **Phase 3**: Implement False Positive Filter and reporting.
4. **Phase 4 (optional)**: Test on real-world codebases and refine.

---

6. Challenges & Mitigations (optional)

- **Multi-Language Support**: Start with C/Python; extend using `tree-sitter`.
- **Model Bias**: Use diverse training data (multiple repos/languages).
- **Scalability**: Parallelize AST parsing with multiprocessing.

7. Glossary

- **AST**: Abstract Syntax Tree
- **CFG**: Control Flow Graph
- **GNN**: Graph Neural Network
- **SARIF**: Static Analysis Results Interchange Format
- **CVSS**: Common Vulnerability Scoring System