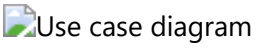
VulWitch: Machine Learning Based Static Code Analyzer

Static code analysis is a powerful technique for helping programmers identify bugs and vulnerabilities in their software. Without executing the program, static analysis unearths useful information by solely inspecting code in structured representations, such as abstract syntax trees (ASTs). Traditional static analyzers typically follow two main approaches. One approach is to exercise rule-based pattern matching against ASTs, as in Semgrep. Another one is to perform more complicated control-flow and data-flow analyses, like Meta's Infer does. Given the pattern-matching nature of static analysis tasks, we believe that artificial intelligence (AI) is a natural fit for this domain. To achieve this goal, we propose VulWitch, a machine-learning-based code analyzer. By learning from a large amount of past and future bugs and vulnerabilities, we believe that machine learning methods can facilitate the development and continuous improvement of a code analyzer. Moreover, we do not anticipate that AI-powered code analyzers will entirely replace traditional ones. Instead, we suppose that they can work together in a complementary manner to deliver more accurate results by cross-validating outcomes of each other.

Design of VulWitch

Use case diagram



Use Case Name	**View abstract syntax trees of a source code file**
Actors	User
Preconditions	User gives a source file containing syntactically correct code.
Goal	Let a user view abstract syntax trees of a source file.
Scenario	1. User give a source code file. 2. The file is parsed, and abtract syntax trees of are created. 3. ASTs are serialized and printed.
Exceptions	The source file has code with invalid syntaxes.

Use Case Name	**Parse code and create abstract syntax trees**
Actors	User
Preconditions	User inputs a source file which does not have syntactical errors.
Goal	Parse the file according to syntaxes of the language specification, and create abstract syntax trees.
Scenario	1. User inputs a code file. 2. VulWitch parses the file and generate abstract syntax trees for it.
Exceptions	VulWitch should raise errors if the file has syntax errors.

Use Case Name	**Serialize abstract syntax trees**
Actors	User
Preconditions	A valid source file is parsed, and abtract syntax trees are created.
Goal	Use strings to represent abtract syntax trees.
Scenario	1. User gives a code file. 2. VulWitch parses the file and create abstract syntax trees. 3. VulWitch serializes ASTs into strings to present them to User.
Exceptions	Abstract syntax trees are invalid.

Use Case Name	**Analyze a source code file for common vulnerabilities**
Actors	User
Preconditions	User inputs an exitings source file which contains well-formed code.
Goal	Detect all bugs and vulnerabilities in a source file.
Scenario	1. User gives a source file. 2. VulWitch parse the file and create ASTs. 3. AI Model analyzes function ASTs and reports all detected vulnerabilities. 4. VulWitch generates a detailed report of these vulnerabilities.
Exceptions	1. The source file can not be found at the specific path. 2. The file contains invalid code.

Use Case Name	**Detect common vulnerabilites in the AST of a function**
Actors	AI Model
Preconditions	VulWitch gives the AST of a syntactically valid function.
Goal	Detect vulnerabilities in a function AST.
Scenario	1. VulWitch feeds a function AST into AI Model. AI Model analyzes the AST and reports any detected vulnerabilities along with their categories.
Exceptions	The abstract syntax tree is ill-formed.

You can view the above use case diagram at Lucidcart.

Class diagram

 Class diagram

You can view the above class diagram at Lucidcart.

Security plan

Security goals

1. Vulnerability Detection Performance

- Detect common security weaknesses, including injection attacks, buffer overflows, insecure cryptographic practices, hardcoded secrets, and insecure API usage.
- Map detected vulnerabilities to recognized security standards such as OWASP Top 10, CERT Secure Coding Standards, and CWE identifiers, to support compliance and reporting.
- Minimize false positives and false negatives to maintain high usability.

2. Runtime Performance

- Efficiently analyze large-scale codebases without introducing significant delays into development pipelines.
- Provide fast analysis for individual source files.
- Support parallel analysis of multiple files to scale across multi-core environments.

Security metrics

1. Vulnerability Detection Performance

- False positive rate, percentage of incorrectly flagged non-issues, measured against public vulnerability datasets (e.g., Juliet Test Suite, SARD).
- False Negative Rate, percentage of missed vulnerabilities in benchmark datasets.
- Precision, recall, and F1-score, which are used to evaluate the accuracy of vulnerability classification of our AI model.
- Standards mapping accuracy, percentage of correctly classified vulnerabilities according to OWASP, CERT, and CWE categories.

2. Runtime Performance

- Single file analysis time, average and maximum time to analyze one file of typical size (e.g., <1 MB).
- Large project analysis time, total analysis time for a representative large project (e.g., 1M+ lines of code).
- Parallelization efficiency, speedup ratio when increasing number of processing threads or cores.