Static Analysis vs. Machine Learning: A Study of Source Code Vulnerabilitiy Detection Models

Mohammad Almasi
M.Sc. Computer Science University of Passau
Passau, Germany
almasi01@ads.uni-passau.de

***Abstract*—This thesis focuses on using two different approaches to finding security problems in Python code: traditional static analysis methods and modern machine learning techniques. The main goal is to detect common vulnerabilities like SQL injection, XSS vulnerabilities, command injection, and CSRF issues in Python applications. The system I plan to build will consist of a traditional static analysis scanner using Python's Abstract Syntax Tree (AST) [1] and a machine learning-based scanner. It will be integrated into a Python Flask backend with a React frontend that provides a user-friendly interface for comparing results. Users can upload files, paste code directly, or scan GitHub repositories to test both methods. The expected outcome is a working security scanner that can help Python developers find and fix security issues in their code. This tool will be especially useful for students and small development teams who can't afford expensive commercial security tools.**

***Index Terms*—Web Security, Vulnerability Scanner, SQL Injection, XSS, Static Code Analysis, Cybersecurity**

I. Introduction

Web application security is a big problem today. Hackers are constantly looking for ways to break into websites and steal data. Some of the most common attacks include SQL injection (where attackers can access databases), XSS attacks (where malicious scripts run in users' browsers) [3], [4], and command injection (where attackers can run system commands). Most security tools available today have several problems. First, they're expensive and many students or small companies can't afford them. Second, many tools don't work well with Python or don't understand Python-specific security issues. Python is becoming very popular for web development, especially with frameworks like Django and Flask. However, there aren't many good security tools specifically designed for Python applications. This creates a gap that my thesis aims to fill by combining traditional static analysis with cutting-edge machine learning approaches.

Recent advances in deep learning for code analysis [5] and the success of the VulnerabilityDetection-master project [6] demonstrate the potential for machine learning-based vulnerability detection. By combining these approaches with traditional static analysis, we can potentially achieve better accuracy and coverage than either method alone.

II. Problem Statement

After researching existing security tools, I found several issues:

- Cost: Professional security scanners are expensive. Tools like Veracode or Checkmarx cost thousands of dollars per year, which most students and small teams can't afford.
- Hard to Use: Many security tools are complex to set up and use. They require special training or expertise that many developers don't have.

A. Dataset Statistics

The VulnerabilityDetection-master project successfully created datasets for 4 vulnerability types:

- SQL Injection: 33,600 samples from 336 repositories
- XSS: 3,900 samples from 39 repositories
- Command Injection: 8,500 samples from 85 repositories
- CSRF: 8,800 samples from 88 repositories

III. Research Questions

The main questions I want to answer in this thesis are:

- How can I build both static analysis and machine learning approaches for Python vulnerability detection?
- Which approach (static analysis vs. machine learning) works better for detecting different types of vulnerabilities in Python code?

IV. Static Analysis Methodology

A. Abstract Syntax Tree (AST) Analysis

- Code Parsing: Converts Python source code into structured AST representation [1]
- Pattern Matching: Identifies vulnerability signatures through rule-based pattern recognition
- Context Analysis: Examines surrounding code structure to reduce false positives
- Framework Awareness: Implements specific rules for Django, Flask, and other Python web frameworks

B. Vulnerability Detection Rules

- SQL Injection: Detects direct string concatenation in database queries, unparameterized queries [8]
- XSS: Identifies unfiltered user input in HTML output, unsafe string handling

- Command Injection: Finds direct user input in system command execution
- CSRF: Detects missing CSRF tokens, improper form protection mechanisms

V. Machine Learning Methodology

A. Data Collection and Labeling Strategy

- GitHub Mining: The system mines security-related commits from GitHub repositories using keywords like "fix sql injection issue", "prevent XSS", etc.
- Commit-based Labeling: Code that was changed in security-related commits is labeled as vulnerable, while unchanged code is labeled as non-vulnerable
- Natural Codebase: Uses real-world Python code from production repositories rather than synthetic datasets
- Data Filtering: Implements sophisticated filtering to exclude demonstration repositories, capture-the-flag challenges, and attack showcases
- Dataset Scale: Successfully collected 25,040 commits from 14,686 different repositories, creating specialized datasets for each vulnerability type
- Quality Assurance: Filters out duplicates, low-quality data, and non-Python files to ensure dataset integrity

B. Code Representation and Embedding

- Token-level Analysis: Works at the individual code token level (e.g., "if", "return", "+", "x") rather than file-level analysis
- Word2Vec Embedding: Pre-trains Word2Vec models on large Python code corpora to create numerical vector representations of code tokens [7]
- Context-aware Processing: Uses moving window approach with focus areas (length n) and context windows (length m) to capture sequential dependencies

C. Deep Learning Architecture

- LSTM Networks: Long Short-Term Memory networks [2] for modeling sequential code patterns and long-term dependencies
- Memory-based Learning: Internal state management to remember context from previous tokens
- Gradient Stability: Addresses vanishing/exploding gradient problems common in standard RNNs
- Context Modeling: Each token's vulnerability status depends on its surrounding context, not just the token itself

D. Training Process

- Dataset Creation: 70% training, 15% validation, 15% testing split with random shuffling
- Token Encoding: Uses pre-trained Word2Vec models to convert code tokens to numerical vectors
- Model Persistence: Saves trained models as .h5 files for each vulnerability type
- Performance Metrics: Evaluates using precision, accuracy, recall, and F1 score
- Hyperparameter Optimization: Systematically tested and optimized:

  Word2Vec: 200 dimensions, 100 iterations, min-count=10, keeping original strings

  LSTM: 100 neurons, batch size 128, dropout 20%, Adam optimizer

  Context Parameters: step size n=5, context window m=200 characters

VI. System Design

A. Backend (Python Flask)

- REST API that handles scanning requests
- Separate modules for each type of vulnerability (SQL injection, XSS, etc.)
- User authentication system
- Report generation functionality

B. Frontend (React with TypeScript)

- Web interface where users can upload code or paste it directly
- Real-time progress updates while scanning
- Clear display of found vulnerabilities
- Report download functionality

VII. Expected Results

A. Working Security Scanner

- A tool that can find security problems in Python code using both static analysis and machine learning
- Support for major Python frameworks

B. Existing Implementation

- I have already developed and deployed a working SQL injection vulnerability scanner at https://sql-scanner-thesis.de.r.appspot.com. This system demonstrates my practical experience in building vulnerability detection tools and provides a foundation for the more advanced hybrid approach proposed in this thesis.

C. Access Credentials

- https://sql-scanner-thesis.de.r.appspot.com
- Username: admin
- Password: a

D. Current System Features

- Web-based interface for SQL injection vulnerability detection
- Real-time scanning capabilities
- User authentication system (admin access)
- Deployed on Google Cloud Platform (App Engine)
- Demonstrates practical implementation of security scanning concepts

E. User-Friendly Interface

- Easy-to-use web interface
- Clear reports that explain problems and solutions
- Export options for different report formats

F. Scalable System

- Can handle multiple scan requests at the same time
- Easy to deploy on cloud platforms
- API that other tools can use
- Modular design for adding new features

VIII. Timeline

A. Static Analysis Methodology

- Read papers about vulnerability detection techniques
- Study the VulnerabilityDetection-master project architecture and models
- Set up development environment and tools
- Design the overall system architecture integrating both approaches
- Start building basic prototypes

B. August 2025: Core Development - Static Analysis

- Build the four main vulnerability scanners using static analysis
- Implement AST-based code parsing and pattern matching
- Create framework-specific detection rules (Django, Flask, etc.)
- Develop confidence scoring algorithms
- Test static analysis components individually

C. September 2025: Core Development - Machine Learning Integration

- Integrate and adapt VulnerabilityDetection-master models for Python code
- Set up Word2Vec embedding pipeline
- Implement LSTM model integration
- Develop the Flask API backend
- Set up database and user authentication

D. October 2025: Frontend Development and Integration

- Build the React user interface with TypeScript
- Connect frontend to backend API
- Add real-time scanning features and progress indicators
- Create comprehensive report generation system
- Implement user authentication and session management

E. November 2025: Hybrid System Integration and Testing

- Integrate static analysis and machine learning approaches
- Implement ensemble methods for result combination
- Test the complete system end-to-end
- Compare performance of individual vs. hybrid approaches
- Optimize performance and fix bugs
- Test security of the system itself

F. December 2025: Final Work and Documentation

- Comprehensive system testing and validation
- Performance benchmarking and optimization
- Write complete technical documentation
- Deploy to production environment
- Final testing and quality checks
- Write thesis and prepare for defense

References

[1] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pp. 359–368, ACM, Dec. 2012, doi: 10.1145/2420950.2421003.

[2] M. Sundermeyer, R. Schlüter, and H. Ney, "LSTM Neural Networks for Language Modeling," in *Proc. Interspeech 2012*, Portland, OR, USA, Sept. 9–13, 2012, pp. 194–197, doi: 10.21437/Interspeech.2012-65.

[3] L. K. Shar, H. B. K. Tan, and L. C. Briand, "Mining SQL Injection and Cross Site Scripting Vulnerabilities using Hybrid Program Analysis," Nanyang Technological University and SnT Centre, Univ. of Luxembourg, Tech. Report, 2012.

[4] L. K. Shar and H. B. K. Tan, "Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns," *Information and Software Technology*, vol. 55, no. 10, pp. 1767–1780, 2013, doi: 10.1016/j.infsof.2013.04.002.

[5] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–37, 2018, doi: 10.1145/3212695.

[6] Wartschinski, L. (2019). Detecting Software Vulnerabilities with Deep Learning. Master's thesis, Humboldt University of Berlin.

[7] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[8] L. K. Shar and H. B. K. Tan, "Mining input sanitization patterns for predicting SQL injection and cross-site scripting vulnerabilities," in *Proc. 34th Int. Conf. on Software Engineering (ICSE)*, San Francisco, CA, USA, May 2013, pp. 1293–1296, doi: 10.1109/ICSE.2013.6606688.