SUMMARY

Introduction

This project aims to develop an artificial intelligence (AI)-powered software system for detecting cybersecurity vulnerabilities in critical software systems before deployment. By integrating AI with existing static and dynamic code analysis tools, the project seeks to enhance the identification and mitigation of potential security, ultimately strengthening the defenses of critical systems.

Requirements

Functional Requirements:

Code Scanning and Vulnerability Detection: The system should scan source code during compilation and runtime to identify potential security threats like DDoS attacks and ransomware.

Vulnerability Analysis and Prioritization: The system should analyze each vulnerability based on severity, impact, and likelihood of occurrence to prioritize remediation efforts.

Remediation Recommendations: The system should recommend solutions to address the detected vulnerabilities based on a predefined mapping.

User Interface: The system should provide a user interface for initiating scans, viewing vulnerabilities, and accessing recommendations.

Integrations: The system should integrate with existing development tools and support third-party plugin integration for extended functionality.

AI Model Integration: The system should allow developers to integrate and train AI models for vulnerability detection.

User Management and Permissions: The system should implement user authentication and authorization mechanisms to control access and permissions.

Non-Functional Requirements:

Performance: The system should deliver code analysis results within a reasonable timeframe (e.g., static analysis under 5 minutes, dynamic analysis under 1 minute).

Scalability: The system should be able to handle an increasing number of users and code submissions without performance degradation.

Security: The system should ensure the confidentiality and integrity of sensitive code data throughout the analysis process.

Usability: The system should be user-friendly and provide clear error messages and documentation.

Reliability: The system should be resilient to failures and recover quickly from disruptions.

Design

Project Architecture:

Microservices Architecture: The system is designed as a collection of independent services that communicate through APIs. This promotes modularity, scalability, and ease of development/deployment. Key services include:

Data Management Service: Handles data collection, transformation, and preparation for AI analysis.

AI Prediction Service: Leverages a trained AI model to analyze data and detect vulnerabilities.

Orchestrator Service: Coordinates interactions between other services and external tools (Sonarqube, Datadog).

Design Strategies:

Separation of Concerns: Code within each service is organized based on specific functionalities (data access, application logic, machine learning tasks) for better maintainability.

Single Responsibility Principle: Each component ideally focuses on a single task (e.g., data preprocessing, model training) to improve code clarity and reduce complexity.

Machine Learning Pipeline Design: Data is processed through a well-defined pipeline involving data extraction, cleaning, transformation, training, and evaluation stages.

Integration with External Tools: The system leverages existing tools like Sonarqube and Datadog for static and dynamic code analysis, respectively.

Chosen Architecture and Design Strategies:

Microservices Architecture: The system is designed as independent, modular services (e.g., Data Management, AI Engine) that communicate via APIs. This promotes easier development, deployment, and scaling of each component.

Separation of Concerns: Code within the AI Engine Service is organized based on functionalities (data handling, machine learning tasks) for better maintainability.

Data Preprocessing and Model Training: Data is carefully prepared and transformed before training a machine learning model to identify vulnerabilities in code or system behavior.

Integration with External Tools: The system leverages existing tools like SonarQube and Datadog for static and dynamic analysis, respectively. Custom plugins can be developed for integration.

Code Skeleton and Design Patterns: The code is organized with a clear structure using established design patterns like the Repository Pattern for data access abstraction.

Implementation

Implementation

This section details the key steps involved in implementing the vulnerability classifier model:

Data Preparation:

Data Loading:

Training and validation datasets are loaded from CSV files using pandas (pd.read\_csv).

The "one-hot" encoded vulnerability labels are converted from strings to lists using literal\_eval.

Preprocessing:

Code snippets are tokenized and potentially truncated to a maximum length using the transformers.AutoTokenizer.from\_pretrained function with the pre-trained JavaBERT model.

Batching:

Training and validation data are shuffled and divided into batches of a specified size (batch\_size).

Padding is applied within each batch to ensure sequences have the same length.

Functions like smart\_batching handle this process efficiently.

Model Training:

Model Definition:

The vulnerability classifier model inherits from nn.Module and utilizes a pre-trained JavaBERT model (transformers.AutoModel.from\_pretrained) for feature extraction.

A linear layer is added on top of the pre-trained model's output for classification.

Optimizer and Scheduler:

The AdamW optimizer (torch.optim.AdamW) is used for model training with hyperparameters like learning rate adjusted as needed.

A linear learning rate scheduler (transformers.get\_linear\_schedule\_with\_warmup) is employed to gradually decrease the learning rate during training.

Training Loop:

The model iterates through training batches.

For each batch, forward pass calculates loss using the binary cross-entropy function (nn.BCEWithLogitsLoss).

Backward pass propagates gradients for updating model weights using the optimizer.

A scheduler step might be included to adjust learning rate after each batch or epoch.

Evaluation:

Similar to the training loop, the model processes validation batches to assess performance metrics like accuracy.

The getAccuracy function calculates subset accuracy based on predicted probabilities and ground truth labels.

Checkpoint Saving:

The save\_checkpoint function periodically stores model state dictionaries, optimizer and scheduler states, along with training and validation losses for potential later use (resuming training, evaluating different checkpoints).

Testing:

Test Data Loading:

The test dataset is loaded and preprocessed similarly to the training and validation data.

Model Evaluation:

The trained model is used to predict vulnerabilities on the test set.

Evaluation metrics like accuracy, confusion matrix, and classification report are generated to assess model performance comprehensively.

Testing

Learning Outcomes

Data Preparation:

Loading Datasets (train\_dataset, val\_dataset):

CSV files containing code snippets and their corresponding vulnerability labels (one-hot encoded) are loaded using pandas.

The literal\_eval function is used to convert the string-formatted labels back to lists.

Preprocessing:

The transformers.AutoTokenizer.from\_pretrained function is used to tokenize the code snippets in both the training and validation datasets.

Tokenization breaks down the code into smaller units (words or sub-words) that the model can understand.

The code might be truncated to a maximum length to ensure consistent input size for the model.

Batching:

Training and validation data are shuffled to avoid overfitting and then divided into batches of a specified size (batch\_size).

Padding is applied within each batch to ensure all sequences have the same length.

Functions like smart\_batching handle shuffling, batching, and padding efficiently.

Model Definition (vulnerabilityClassifier class):

Inheritance and Base Model:

The vulnerabilityClassifier class inherits from nn.Module, a core building block for neural networks in PyTorch.

It utilizes a pre-trained JavaBERT model (transformers.AutoModel.from\_pretrained) for feature extraction. The JavaBERT model is trained on a massive dataset of Java code and learns general representations of code elements.

Dropout Layer:

A dropout layer (nn.Dropout) with a probability of 0.1 (DROPOUT\_PROB) is introduced to prevent overfitting.

Dropout randomly drops a certain percentage of activations during training, forcing the model to learn more robust features.

Linear Layer:

A linear layer (nn.Linear) is added on top of the pre-trained model's output. This layer transforms the extracted features from JavaBERT into a probability distribution over the number of vulnerability classes (N\_CLASSES).

Learning Rate Scheduler:

The model uses a learning rate scheduler (step\_scheduler\_after) to adjust the learning rate during training.

A common strategy is to gradually decrease the learning rate as training progresses, allowing the model to fine-tune its weights.

Model Training (train\_fn function):

Forward Pass:

The model takes code snippet (ids) and attention mask (mask) as input during the forward pass.

The attention mask helps the model focus on relevant parts of the code snippet.

The pre-trained JavaBERT model extracts features, and the dropout layer and linear layer transform them into vulnerability probabilities.

Loss Calculation:

The binary cross-entropy loss function (nn.BCEWithLogitsLoss) is used to compare the predicted probabilities with the actual one-hot encoded vulnerability labels.

This loss function measures the difference between the model's predictions and the ground truth labels.

Backward Pass and Optimization:

The calculated loss is backpropagated through the network using the backward pass.

The optimizer (torch.optim.AdamW) updates the model's weights based on the gradients to minimize the loss.

Evaluation (accuracy):

The getAccuracy function calculates the subset accuracy based on the predicted probabilities and ground truth labels.

Subset accuracy refers to the proportion of samples where the model correctly predicts all assigned vulnerabilities (or none).

Checkpoint Saving:

The save\_checkpoint function periodically stores the model's state dictionary, optimizer and scheduler states, and training/validation losses for potential later use (resuming training, evaluating different checkpoints).

Model Evaluation (eval\_fn function):

Similar to Training:

The model follows a similar process as training, processing batches of validation data to assess performance metrics like accuracy.

The evaluation metrics are calculated on the validation set, which helps to avoid overfitting and provides an estimate of how well the model generalizes to unseen data.

Testing (after training):

Test Data Preparation:

The test dataset is loaded and preprocessed similarly to the training and validation datasets.

Model Evaluation:

The trained model is used to predict vulnerabilities on the unseen test set.

Evaluation metrics like accuracy, confusion matrix, and classification report are generated to provide a comprehensive assessment of the model's performance on unseen data.

This Python script defines a Flask web API for vulnerability prediction in Java code. It leverages a pre-trained vulnerability classifier model and a multi-label binarizer.

Functionality:

Accepts code snippets as input: The API takes code sections or entire files in JSON format through POST requests.

Preprocesses code: The code is cleaned (removing comments and whitespaces) before being fed to the model.

Tokenizes code: The code is broken down into smaller units (tokens) suitable for the model's processing.

Predicts vulnerabilities: The vulnerability classifier model predicts the presence of various vulnerabilities based on the code's content.

Returns predictions: The API responds with a JSON object containing the predicted vulnerabilities associated with the code sections or files.

Key Points:

The script utilizes pre-trained models:

CAUKiel/JavaBERT: A transformer model trained for understanding Java code.

A multi-label binarizer (mlb): This model maps vulnerability labels to binary vectors for efficient prediction.

The code is chunked and processed in batches to handle sequences exceeding a maximum length.

A sigmoid function is applied to the model's output to convert it into probabilities between 0 and 1.

A threshold of 0.5 is used for filtering the predicted vulnerabilities (probabilities above 0.5 are considered vulnerabilities).

Training Phase of the Vulnerability Classifier Model

The training phase involves using the prepared data to train the vulnerability classifier model. Here's a breakdown of the key steps:

1. Data Loading:

The preprocessed and formatted training data (code snippets, labels) are loaded into memory or accessed from a data loader.

2. Model Definition:

The vulnerabilityClassifier class likely defines the model architecture.

It specifies:

The pre-trained JavaBERT model (loaded using transformers.AutoModel.from\_pretrained).

The number of hidden layers and neurons in the MLP.

Dropout layer configuration (probability for dropping activations).

Output layer with a sigmoid activation function.

3. Optimizer and Loss Function:

An optimizer (e.g., AdamW) is chosen to update the model's weights based on the calculated loss during training.

A loss function (e.g., binary cross-entropy for binary classification or multi-class cross-entropy for multi-label classification) is used to measure the difference between the model's predicted probabilities and the actual vulnerability labels.

4. Forward Pass:

A batch of training data (code snippets and labels) is fed into the model.

The JavaBERT model extracts features from the code.

The features are passed through the MLP layers.

The output layer generates probabilities for each vulnerability class.

5. Backward Pass and Optimization:

The loss function calculates the difference between the predicted probabilities and the actual labels.

The calculated loss is backpropagated through the network.

The optimizer uses the gradients to update the weights of the model in a direction that minimizes the loss.

6. Epochs and Evaluation:

The training process iterates over the entire dataset multiple times (epochs).

After each epoch, the model might be evaluated on a validation set to monitor its performance and prevent overfitting.

Overfitting occurs when the model memorizes the training data and performs poorly on unseen data.

7. Saving the Model:

Once training is complete, the model's state dictionary containing the trained weights and biases is saved for later use (prediction or further training).

Model Prediction

The trained model can be used to predict vulnerabilities in new, unseen Java code snippets:

1. Preprocessing New Code:

The new code snippet is preprocessed similarly to the training data (cleaning and tokenization).

2. Forward Pass:

The preprocessed code is fed into the trained model.

JavaBERT extracts features.

The features are passed through the MLP layers.

The output layer generates probabilities for each vulnerability class.

3. Classification (Thresholding):

Depending on the chosen classification strategy:

For binary classification (vulnerable vs. non-vulnerable), a threshold (e.g., 0.5) is applied to the predicted probability. Values above the threshold are classified as vulnerable.

For multi-label classification, a threshold might be applied to each class probability independently, or a more sophisticated approach like selecting the top-k most likely vulnerabilities might be used.

4. Output:

The model outputs the predicted vulnerability class(es) or a list of probabilities for each class, depending on the configuration.

Overall, the training phase involves iteratively refining the model's weights to learn from the labeled data. The trained model can then be used to predict vulnerabilities in new code based on the patterns it has learned from the training data.