CHAPTER 1
INTRODUCTION

1.1 VULNERABILITY PREDICTION OF OPEN SOURCE SOFTWARES:

Open-Source Software (OSS) plays a critical role in modern software development. With the widespread adoption of OSS libraries, tools, and frameworks, ensuring the security and stability of these components is paramount. However, open-source projects are particularly susceptible to vulnerabilities due to their open nature, which allows anyone to view, use, and modify the source code. As a result, vulnerability prediction in OSS has become an important area of research and practice, helping developers and security experts anticipate, identify, and mitigate security risks in OSS before they can be exploited. Here's a detailed overview of vulnerability prediction in OSS:

What is Vulnerability Prediction?

Vulnerability prediction refers to the process of using analytical and machine learning methods to identify sections of code that are more likely to contain security flaws. By predicting vulnerabilities early in the development lifecycle, organizations can focus their resources on high-risk areas, reducing the time and cost required to secure their software. In OSS, vulnerability prediction tools can proactively assess the likelihood of vulnerabilities emerging in a given project, codebase, or version.

1.2. Challenges in Vulnerability Prediction for OSS

- **Code Complexity**: OSS projects can be highly complex, with intricate dependencies that increase the chances of vulnerabilities appearing across the software stack.
- **Lack of Uniform Security Standards**: OSS projects are often developed by a diverse set of contributors with different levels of security expertise and varying adherence to security best practices.
- **Frequent Updates and Patches**: OSS is frequently updated, which introduces new functionality and bug fixes but can also inadvertently introduce new vulnerabilities.

- **Limited Data on Vulnerabilities**: Not all vulnerabilities are reported immediately. For OSS projects, data on historical vulnerabilities may be inconsistent or incomplete, complicating prediction efforts.

1.3. Data Sources for Vulnerability Prediction

To predict vulnerabilities effectively, vulnerability prediction models rely on multiple data sources:

- **Code Repositories**: Version control systems like GitHub or GitLab contain historical data about changes in code, commits, contributors, and issues. Mining these repositories provides valuable insights into code structure, modification history, and collaboration patterns.
- **Issue Tracking Systems**: Platforms like Jira or Bugzilla track issues, bugs, and security vulnerabilities over time. This data helps in identifying common patterns that may lead to vulnerabilities.
- **Static Code Analysis Tools**: Tools like SonarQube, Bandit, or CodeQL can analyse the code for common vulnerability patterns, generating useful features for prediction models.
- **Dependency Management Systems**: Dependency data from tools like Maven, npm, and PyPI allow vulnerability prediction models to analyse external dependencies and alert users to potential risks.

1.4. Approaches to Vulnerability Prediction

Vulnerability prediction methods can broadly be classified into two categories: **Static Analysis** and **Machine Learning/AI-Based Prediction**.

- **Static Analysis**: This involves analysing the code structure, syntax, and logic without executing it. Static analysis tools can identify coding practices known to introduce vulnerabilities, such as SQL injection risks or buffer overflows.
- **Machine Learning and AI-Based Prediction**: Machine learning models, including supervised, unsupervised, and semi-supervised learning, can be used to predict vulnerabilities based on historical code patterns, developer activity, and previous vulnerabilities.
    - **Supervised Learning**: Models are trained on labelled datasets containing code sections marked as vulnerable or non-vulnerable.

Techniques include decision trees, support vector machines, and neural networks.

- **Unsupervised Learning**: Clustering algorithms group similar code structures and identify outliers that may represent vulnerabilities.
- **Deep Learning**: Techniques such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) analyse complex code patterns. Transformer models have also been applied to vulnerability prediction, given their success in natural language processing tasks.

1.5. Motivation and Problem Statement

Motivation

Open-Source Software (OSS) has become a vital part of modern software development, powering everything from individual applications to large enterprise systems. While OSS offers benefits like reduced costs, community-driven improvements, and greater flexibility, it also poses unique security challenges. Vulnerabilities in OSS can be exploited by malicious actors, potentially leading to severe security breaches, data theft, and operational disruptions. Predicting vulnerabilities in OSS helps in proactive mitigation, making systems more secure and reliable. The primary motivation behind vulnerability prediction in OSS is to identify high-risk components early, thereby reducing potential exploitation and improving overall software resilience.

Problem Statement

The objective of this project is to develop a predictive model that can accurately identify vulnerabilities in open-source software components before they are exploited. Vulnerabilities in OSS are difficult to track due to the diversity in development practices, the varying levels of maintenance, and the complexity of dependency chains. This problem is exacerbated by the fact that many organizations rely on third-party libraries, over which they have limited control and visibility.

By leveraging techniques such as machine learning, static and dynamic code analysis, and mining software repositories, this research aims to analyse patterns in open-source repositories to predict potential vulnerabilities. The model should be able to detect possible flaws and alert users or developers, enabling timely remediation efforts. This solution could reduce the number of security

incidents, lower costs associated with security patching, and enhance trust in OSS.

1.6. Objectives Of the Project

The objective of vulnerability prediction in open-source software (OSS) is to proactively identify potential security weaknesses or defects in OSS components before they can be exploited. By predicting vulnerabilities, organizations can enhance their software security posture, reduce risk, and allocate resources more effectively. Here are some key objectives:

- **Proactive Threat Mitigation:**
  Detect potential vulnerabilities early to minimize the risk of security breaches, data loss, or service disruptions.

- **Efficient Resource Allocation:**
  Prioritize security resources and testing efforts on the most vulnerable parts of the codebase, improving efficiency in addressing potential threats.

- **Cost Reduction:**
  Avoid the high costs associated with reactive vulnerability management, such as patching after a breach or dealing with compliance issues.

- **Improved Code Quality:**
  Identify patterns or characteristics in code that commonly lead to vulnerabilities, thereby improving the overall quality and robustness of OSS projects.

- **Support for Secure Development Practices:**
  Encourage secure coding practices by integrating vulnerability prediction tools within the development lifecycle, so developers can avoid common pitfalls that may introduce vulnerabilities.

- Enhanced Trust in OSS:
  Build trust and increase the adoption of OSS by demonstrating a

proactive approach to security, making OSS more reliable for end-users.

- **Data-Driven Insights:**
  Gather data on historical vulnerabilities to better understand trends, risk factors, and patterns in software vulnerabilities, which can improve future predictions and security measures.

- **Compliance and Governance:**
  Ensure OSS projects meet security compliance standards, such as OWASP, NIST, or ISO, by identifying vulnerabilities and addressing them in a timely manner.

1.7. Scope and Limitations

The scope and limitations of vulnerability prediction in open-source software (OSS) are influenced by factors like the diversity of OSS projects, the complexity of codebases, and the availability of historical vulnerability data. Here's a breakdown:

Scope:

- Predicting Vulnerabilities Across OSS Projects:
  Vulnerability prediction aims to cover a wide range of OSS projects, from small libraries to large-scale frameworks. By analysing code patterns, libraries, and dependencies, prediction models can flag potential vulnerabilities.
- Identifying Risky Code Patterns:
  Vulnerability prediction models often target specific code patterns, configurations, or libraries that have historically been associated with security flaws (e.g., SQL injection, buffer overflow).
- Using Machine Learning for Prediction:
  Machine learning models are frequently used to analyse code and past vulnerabilities to predict potential issues in similar code segments. These models are trained using features like code complexity, commit history, or known bug patterns.
- Assisting Security and Development Teams:
  Vulnerability predictions can be integrated into the development pipeline,

providing insights that help prioritize code reviews, testing, or patching efforts, and enabling a proactive security stance.

- Improving Security Awareness in the OSS Community:
  By publicly sharing vulnerability data and predictive insights, OSS communities can become more aware of common vulnerabilities, leading to better code quality and more secure OSS development practices.

Limitations:

- Dependence on Historical Data Quality:
  Prediction models rely heavily on historical vulnerability data. For lesser-known or recently developed OSS projects with limited data, predictions may be less reliable or even impossible.
- False Positives and False Negatives:
  Prediction models are prone to false positives (flagging non-vulnerable code as vulnerable) and false negatives (failing to detect actual vulnerabilities), which can lead to wasted resources or undetected risks.
- Limited Understanding of Complex, Contextual Vulnerabilities:
  Vulnerabilities that rely on complex interactions, business logic, or unique configurations may be difficult to predict, as machine learning models typically analyse code patterns rather than broader system contexts.
- Dynamic vs. Static Analysis Challenges:
  Static analysis, commonly used in vulnerability prediction, cannot always capture issues that only emerge at runtime. Dynamic factors like specific user inputs or environment configurations may reveal vulnerabilities that prediction models overlook.
- Rapid Changes in OSS Codebases:
  Many OSS projects evolve quickly, with frequent updates and new features. A model that correctly identifies vulnerabilities today may need constant retraining to remain relevant as the codebase changes.
- Resource Constraints in OSS Projects:
  Some OSS projects lack the resources to implement comprehensive security practices or integrate advanced vulnerability prediction models, particularly smaller projects maintained by a few developers.
- Lack of Standardization:
  OSS projects vary greatly in structure, language, and coding practices, making it challenging to develop a one-size-fits-all prediction model that can handle such diversity effectively.

- Privacy and Legal Concerns:
  Sharing predictive insights and vulnerability data across OSS projects can raise concerns about data privacy and legal liability, especially for vulnerabilities that could be exploited if publicly disclosed.

1.8. Relevance of ML and Hybrid Model in vulnerability Prediction:

Machine learning (ML) and hybrid models are increasingly relevant in predicting vulnerabilities in open-source software (OSS) due to their ability to analyze complex patterns in code, automate detection, and improve prediction accuracy. Here's how they contribute:

1. Improved Accuracy with Pattern Recognition

- ML Models: Machine learning models, especially deep learning models, can identify patterns in code and historical vulnerability data that would be difficult or time-consuming for humans to spot. They can detect nuanced correlations between certain coding practices, dependencies, and vulnerability-prone areas, enhancing prediction accuracy.
- Hybrid Models: Combining different ML approaches (e.g., supervised and unsupervised learning) or blending traditional rule-based systems with ML can further improve accuracy. These hybrid models can balance the flexibility and learning ability of ML with the precision and structure of rule-based systems.

2. Ability to Process Large, Dynamic Data

- ML: Machine learning algorithms can process large datasets, such as vast OSS codebases and repositories, enabling continuous learning as new vulnerabilities are reported.
- Hybrid Models: Hybrid models allow the integration of multiple data sources and learning methods. They can handle both structured (e.g., code structure, dependencies) and unstructured data (e.g., commit messages, issue reports), which provides a more comprehensive approach to vulnerability prediction.

3. Enhanced Adaptability to New Threats

- ML Models: By continuously training on new data, ML models can adapt to new security threats and evolving coding patterns, making them suitable for handling the fast-paced updates in OSS.
- Hybrid Models: Hybrid approaches can combine ML with other methodologies, such as static analysis and heuristics, to adaptively address vulnerabilities from multiple perspectives, ensuring that prediction models are resilient against diverse attack methods.

4. Reduced False Positives and False Negatives

- ML: Models trained with sufficient data can learn to minimize false positives (reporting vulnerabilities where there are none) and false negatives (missing actual vulnerabilities). This reduces alert fatigue among developers and ensures that real threats are more likely to be addressed.
- Hybrid Models: By leveraging both machine learning and traditional rule-based methods, hybrid models can further refine the results. For instance, ML might flag potential vulnerabilities, while rules help validate or weigh their relevance, reducing unnecessary or incorrect alerts.

5. Feature Engineering for Better Predictions

- ML: Machine learning models can automatically learn important features from data, but they often benefit from specific domain knowledge, such as understanding coding conventions and known vulnerable patterns in OSS.
- Hybrid Models: Hybrid models can integrate expert knowledge and ML-generated insights, combining the benefits of both for more relevant feature selection, which improves model performance. For instance, features from static analysis, code complexity, and dependency scanning can be incorporated along with historical ML-learned patterns.

6. Early Detection and Proactive Mitigation

- ML and Hybrid Models: These models can help identify risky components during development and before deployment, which is crucial for open-source projects that rely on the community for maintenance. Early detection is particularly valuable in OSS, as contributors and users are often decentralized, and rapid issue resolution may be challenging.

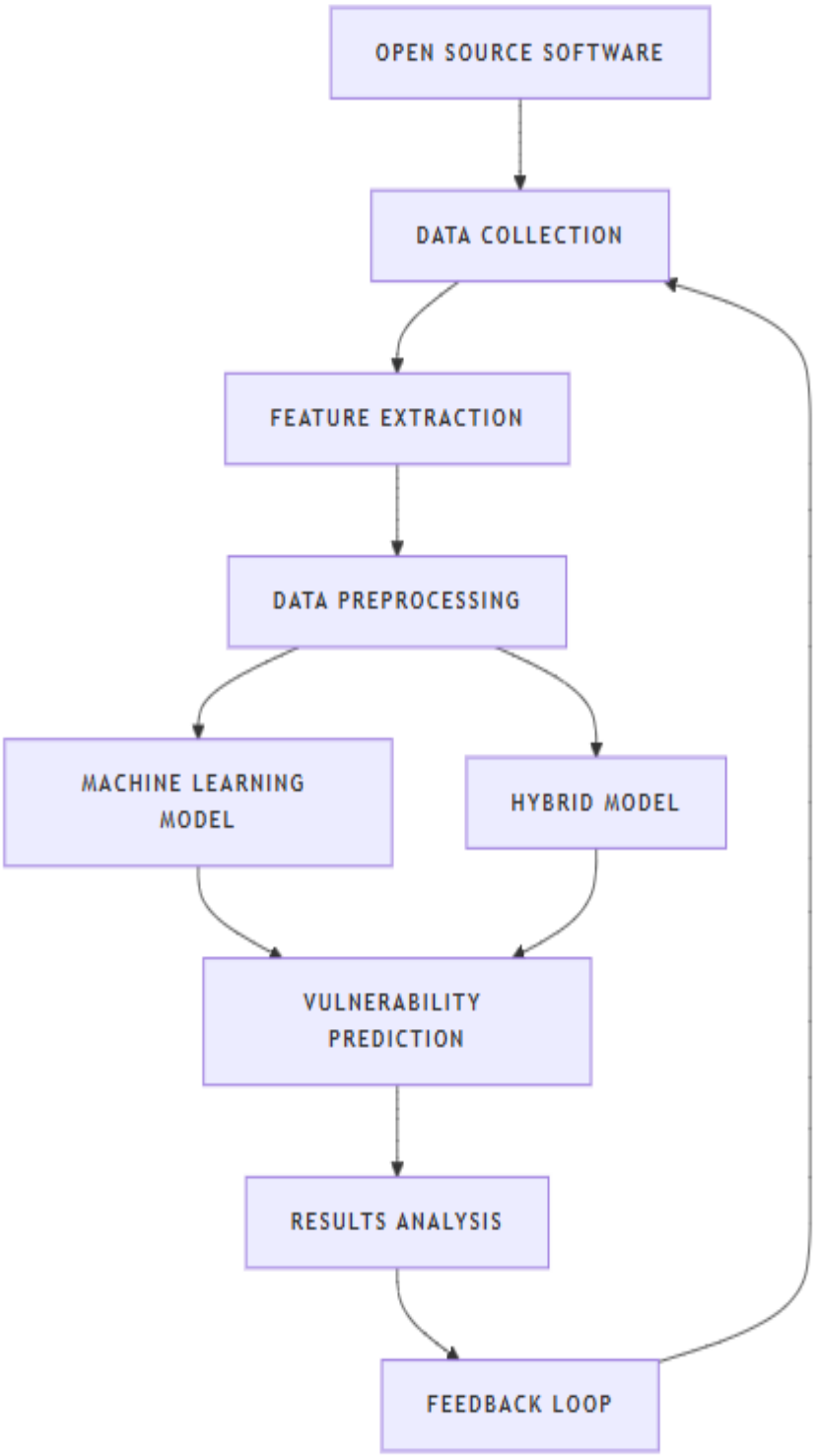
7. Increased Trustworthiness and Adoption of OSS

- ML and Hybrid Models: By helping predict and address vulnerabilities, these models increase the reliability of OSS for enterprises. Knowing that OSS has a robust predictive layer for vulnerability management builds confidence, encouraging wider adoption and contributing to software resilience.

8. Continuous Improvement through Feedback Loops

- ML: Machine learning models can improve continuously through feedback loops from real-world usage data. They learn from false positives and negatives, allowing OSS projects to fine-tune vulnerability prediction over time.
- Hybrid Models: Hybrid models can integrate feedback more efficiently, where rule-based components can be updated or replaced without affecting the learning aspects of the model. This enables a more modular and iterative improvement process.

1.9. ARCHITECTURAL DIAGRAM



CHAPTER 2
LITERATURE REVIEW

A scholarly, which includes the current knowledge including substantive findings, as well as theoretical and methodological contributions to a particular topic. Literature reviews are secondary sources, and do not report new or original experimental work. Most often associated with academic-oriented literature, such reviews are found in academic journals, and are not to be confused with book reviews that may also appear in the same publication. Literature reviews are a basis for research in nearly every academic field. A narrow-scope literature review may be included as part of a peer-reviewed journal article presenting new research, serving to situate the current study within the body of the relevant literature and to provide context for the reader. In such a case, the review usually precedes the methodology and results sections of the work.

2.1 Toward More Effective Deep Learning-based Automated Software Vulnerability Prediction, Classification, and Repair

Software vulnerabilities are prevalent in software systems and the unresolved vulnerable code may cause system failures or serious data breaches. To enhance security and prevent potential cyberattacks on software systems, it is critical to (1) early detect vulnerable code, (2) identify its vulnerability type, and (3) suggest corresponding repairs. Recently, deep learning-based approaches have been proposed to predict those tasks based on source code. In particular, software vulnerability prediction (SVP) detects vulnerable source code; software vulnerability classification (SVC) identifies vulnerability types to explain detected vulnerable programs; neural machine translation (NMT)-based automated vulnerability repair (AVR) generates patches to repair detected vulnerable programs. However, existing SVPs require much effort to inspect their coarse-grained predictions; SVCs encounter an unresolved data imbalance issue; AVRs are still inaccurate.

I hypothesize that by addressing the limitations of existing SVPs, SVCs and AVRs, we can improve the accuracy and effectiveness of DL-based approaches for the aforementioned three prediction tasks. To test this hypothesis, I will propose (1) a finer-grained SVP approach that can point out vulnerabilities at the line level; (2) an SVC approach that mitigates the data imbalance issue; (3) NMT-based AVR approaches to address limitations of previous NMT-based approaches. Finally, I propose integrating these novel

approaches into an open-source software security framework to promote the adoption of the DL-powered security tool in the industry.

Author: Michael Fu, Year:2023

2.2. Identifying Evolution of Software Metrics by Analysing Vulnerability History in Open-Source Projects

Software developers often prioritize functional code over security concerns during development. However, security has recently gained importance across all phases of the Software Development Life Cycle (SDLC), from requirements specification to maintenance. As a result, research has increasingly focused on addressing security issues throughout these phases, with a particular emphasis on developing predictive models for vulnerability detection. Most current prediction models rely on software metrics, achieving high precision but low recall rates, and rarely examining the specific impact of individual metrics on vulnerability occurrences.

In this study, we investigate the evolution of software metrics across the lifecycle of vulnerabilities, from their initial introduction to the final fixed version, using 250 files from three major Apache Tomcat releases (versions 8, 9, and 10). Our findings indicate that four key metrics—AvgCyclomatic, AvgCyclomaticStrict, CountDeclMethod, and CountLineCodeExe—undergo significant changes throughout a vulnerability's history in Tomcat. Additionally, we observe that the Tomcat team prioritizes fixing critical vulnerabilities, such as those causing Denial of Service, over less severe issues.

Our results aim to encourage further research into more accurate vulnerability prediction models that leverage relevant software metrics, while also providing insights into developers' prioritization strategies for addressing vulnerabilities in open-source projects.

Author: Erik Maza and Kazi Zakia Sultana, Year: 2022

2.3. Function-Level Software Metrics for Predicting Vulnerable Code

Limited experience or hard deadlines often lead developers to technical Q&A websites, e.g., Stack Overflow, to find quick answers to their coding problems. Research, however, suggests that a large share of the answers on Stack Overflow include vulnerable code and their use in a software system may cause major security implications. Researchers have suggested several approaches, e.g., leveraging software metrics, to predict vulnerability in the source code. The majority of these approaches require the whole code base or a big chunk of it to perform the prediction. The answers on those Q&A websites, however, are code

snippets typically with a single function, or, at most a couple of related functions. Predicting vulnerabilities in such code snippets has received limited attention. Few examples present in the literature suggest leveraging function-level software metrics to predict vulnerable code. Such work, however, provides incomplete analysis with heavily imbalanced datasets. This paper examines whether function-level code metrics can differentiate the vulnerable and fixed versions of a given function. In particular, we develop a "near-balanced" dataset with a total of 132,754 function instances, calculate 18 function-level code metrics for both vulnerable (49.6% of the total) and fixed versions of the functions, and conduct binary logistic regression with the Wald test. The results suggest that function-level code metrics may not be useful in predicting if a given function contains vulnerability. Index Terms—software metrics; function-level software metrics; software vulnerabilities; logistic regression

Author: Md Rayhan Amin, Daniel Tanner, Yashita Sharma, Kollin Napier, and Tanmay Bhowmik, Year:2024

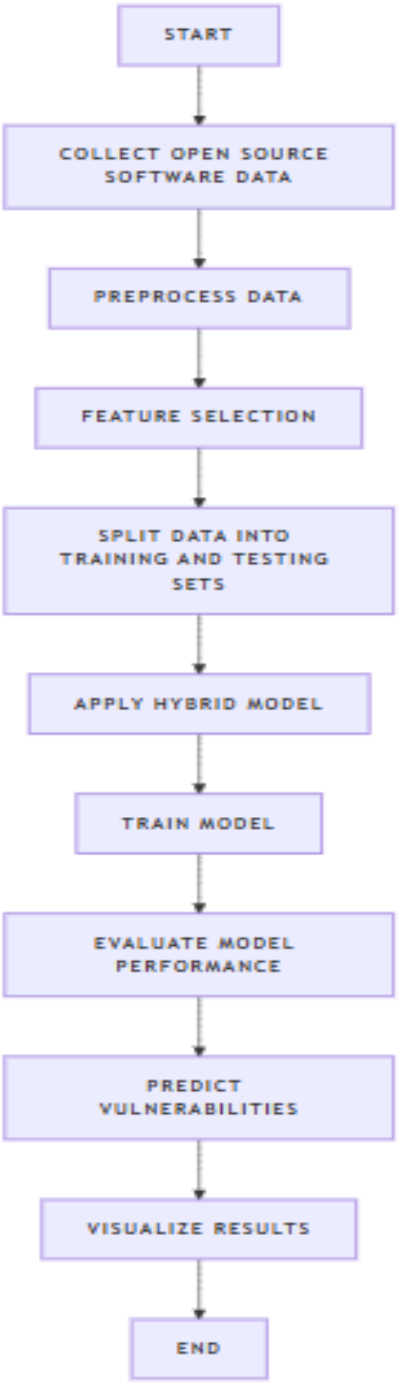
2.4. A Comparative Study on Hyperparameter Optimization Methods in Software Vulnerability Prediction

Hyperparameter optimization (HPO) is the procedure to find the optimal hyperparameters for predictive modelling. In order to build an efficient software vulnerability prediction (SVP) model, effective HPO method is required. Recent studies have explored the area of machine learning for SVP models' construction. Different machine learning algorithms have distinct hyperparameters to be tuned. In this paper, we have compared four HPO methods for six machine learning based SVP models using an open-source public dataset 'Drupal'. The performance of the models is evaluated by the accuracy of the model and the computational time. Experiments are performed using various python libraries such as sklearn, skopt, hyperopt, and tpot. After comparative analysis, we found that Bayesian optimization and random search has shown the performance improvement for SVP models with effective computational time.

Author: Deepali Bassi and Hardeep Singh, Year:2021

CHAPTER 3

SYSTEM DIAGRAM



These are stages involved in the vulnerability prediction now we shall breakdown each step to know further about it:

1. **Collect open source software data:**
   Collecting data for vulnerability prediction in open-source software involves gathering information from a variety of sources that provide insights into past vulnerabilities, code patterns, and historical security incidents. Here are key steps and resources to collect relevant data effectively:

   Gather Vulnerability Databases

   - **National Vulnerability Database (NVD)**
     The NVD is a comprehensive repository that includes details about publicly reported vulnerabilities in various software, including open-source projects. It provides data on CVEs (Common Vulnerabilities and Exposures), vulnerability severity, and mitigation recommendations.
   - **Common Vulnerabilities and Exposures (CVE) List**
     The CVE list, maintained by MITRE, is a standard list of publicly disclosed vulnerabilities. It can be used to identify vulnerabilities in open-source software components.
   - **Open Source Vulnerability (OSV) Database**
     OSV is a Google-led initiative that catalogs vulnerabilities in open-source libraries and packages. It focuses on ecosystems like Python, JavaScript, and others, and provides structured vulnerability data.

   Extract Version Control Data

   - **GitHub, GitLab, Bitbucket**
     Version control repositories for open-source projects provide rich data on code commits, bug reports, and issue histories. Tools like GitHub's REST API or GraphQL API can be used to pull data on pull requests, issues, commit messages, and release notes.
   - **Commit History Analysis**
     Use commit messages to identify instances where developers have fixed vulnerabilities. Terms like "security fix," "CVE," or "bug fix" can help filter relevant commits.
   - **Software Metrics (Code Quality)**
     Tools such as SonarQube or CodeQL can analyze OSS code repositories for code smells, technical debt, and security

issues, generating code quality metrics that help identify potential vulnerabilities.

2. Preprocess Data

Data preprocessing for vulnerability prediction in open-source software involves cleaning, transforming, and organizing data to create a reliable dataset for training predictive models. The quality of this preprocessing step is crucial, as it directly affects the accuracy of the predictions. Here are the main steps and considerations for data preprocessing in this context:

1. Data Collection and Integration

- Source Identification: Collect data from reliable sources, such as version control systems (e.g., GitHub, GitLab), issue trackers (e.g., JIRA), vulnerability databases (e.g., CVE, NVD), and static code analysis tools.
- Data Types: Gather a variety of data, including source code, change history, commit messages, bug reports, and metadata related to files or components.

2. Data Cleaning

- Remove Duplicates: Ensure there are no duplicate entries (e.g., duplicate commits or identical code snippets) that could bias the model.
- Handle Missing Values: Fill or impute missing data points, such as blank fields in bug reports, commit messages, or security classifications.
- Normalize Text Data: Clean textual data, like commit messages or bug descriptions, by removing irrelevant information, special characters, and stop words, making it easier for text-based models to process.

3. Labelling Data

- Classify Vulnerable vs. Non-Vulnerable Code: Label code segments, commits, or files as "vulnerable" or "non-vulnerable" based on previous bug reports, CVE references, or static code analysis results.
- Categorize Vulnerability Types: Where applicable, categorize vulnerabilities by type (e.g., SQL injection, buffer overflow) based on standard taxonomies (e.g., OWASP or MITRE CWE).
- Time-Based Labels: For temporal analyses, create time-based labels to track vulnerability occurrences over specific intervals.

4. Feature Extraction and Engineering

- Code Features: Extract features from code, such as function length, cyclomatic complexity, number of lines of code, or the use of certain libraries or functions known to be risky.
- Textual Features: For commit messages or issue descriptions, use Natural Language Processing (NLP) techniques like TF-IDF, word embeddings, or n-grams to capture the content.

- Metadata Features: Extract features like author details, number of contributors, frequency of changes, and file ownership.
- Historical Features: Add information on past vulnerabilities or defect density, such as the number of times a file has been patched or the frequency of security-related issues.
- Social Features: Use project-related data like the size of the community, user ratings, and social metrics (e.g., number of stars or forks in GitHub), which can correlate with code quality.

5. Data Transformation

- Normalization/Standardization: Scale numerical features to a standard range, such as [0, 1] or a standard normal distribution, which helps many machine learning algorithms perform better.
- Tokenization and Encoding: Convert text data into tokens and apply encoding techniques such as one-hot encoding or word embeddings (e.g., Word2Vec, GloVe) for textual features.
- Handling Imbalanced Data: Use techniques like SMOTE (Synthetic Minority Over-sampling Technique) or undersampling to balance classes if vulnerable code samples are much rarer than non-vulnerable samples.

3. Feature Selection:

Feature selection plays a critical role in vulnerability prediction for open-source software (OSS), as it helps identify the most relevant attributes for predicting potential vulnerabilities. By selecting important features, prediction models can achieve higher accuracy, efficiency, and interpretability. Here are some common types of features used in vulnerability prediction for OSS:

1. Code-Based Features

These features capture attributes of the code itself and are often derived from code analysis tools.

- Complexity Metrics: Measures of code complexity, such as cyclomatic complexity, code length, depth of inheritance, and coupling between objects.
- Code Churn: Frequency of changes to a piece of code, often correlated with the likelihood of introducing vulnerabilities.
- Code Structure Metrics: Number of classes, methods, lines of code, comments, and loops.
- Defect Density: Historical density of defects in a module, indicating potentially risky areas of code.
- Code Dependencies: Dependency on external libraries or other modules, which might increase the risk of vulnerabilities.

2. Process-Based Features

These are derived from the software development and maintenance processes.

- Commit Frequency and Volume: High commit frequency or large volumes of commits may indicate volatile areas of code.
- Developer Experience: Experience level of the developers working on a module, as less experienced developers may inadvertently introduce vulnerabilities.
- Bug Reports and Issue Tracker Data: Frequency and severity of bug reports, particularly security-related issues, may help predict vulnerability.
- Time Since Last Update: Code that hasn't been updated recently may have unaddressed vulnerabilities, especially if security practices have evolved.

3. Textual and Semantic Features

These leverage the actual content and context of the code or commit messages.

- Commit Message Analysis: Specific keywords or phrases in commit messages (e.g., "fix," "security," "vulnerability") can indicate potentially risky changes.
- Natural Language Processing (NLP) Features: NLP techniques applied to comments and documentation can help identify areas with low clarity or potentially error-prone code.
- File Names and Directory Structure: Certain files, like configuration files, may be more prone to vulnerabilities.

4. Historical Features

These capture past trends and patterns associated with vulnerabilities.

- Historical Vulnerabilities: Previous vulnerabilities or security flaws in certain modules indicate a higher risk of future issues.
- Patch Frequency and Pattern: Frequency and nature of patches applied to a module over time.
- Time Lag Between Discovery and Patch: Time it took to patch a previous vulnerability; a larger time lag may correlate with higher vulnerability.

4. Split Data into Training And Testing Sets:

Splitting data into training and testing sets is a crucial step in building a reliable vulnerability prediction model. In vulnerability prediction for open-source software (OSS), this process ensures that the model is both trained effectively and evaluated properly on unseen data, which helps improve its generalization capability. Here's how to approach this split:

1. Define the Dataset

- Your dataset may include various OSS repositories, files, or code snippets labeled with vulnerability indicators (e.g., vulnerable or not).

- It could also include specific features like code complexity, commit history, code churn, dependencies, contributor information, and historical vulnerability data.

2. Choose an Appropriate Split Ratio

- Typical Ratios: A common approach is to split 70–80% of the data for training and 20–30% for testing. This ensures enough data for the model to learn while retaining a portion for evaluation.
- Considerations for Small Datasets: If the dataset is small, a 90-10 split may be necessary, or even a cross-validation approach, to ensure enough data for both training and testing.

3. Stratified Sampling (if applicable)

- Balanced Distribution: If the dataset is imbalanced (i.e., many more non-vulnerable samples than vulnerable ones), use stratified sampling. This technique maintains the proportion of vulnerable and non-vulnerable samples in both training and testing sets, ensuring the model has balanced data to learn from.
- Handling Imbalance: Another approach for imbalance is to use techniques like oversampling or undersampling, especially for training data, to make sure the model learns effectively from vulnerable cases.

4. Temporal Split (for Sequential Data)

- When Using Time-Based Features: If the dataset includes time-based features (e.g., commit history or version updates), you might consider a temporal split where older data is used for training and newer data for testing. This simulates real-world scenarios where predictions are made based on historical data.
- Benefits for Generalization: Temporal splits help the model generalize to newer code or repositories that it hasn't "seen" during training, which is closer to how vulnerability prediction would work in practice.

5. Feature Selection and Engineering

- Select and engineer features (e.g., complexity metrics, dependency counts, author experience) before the split to avoid data leakage, where information from the test set could inadvertently influence the training process.

5. Apply Hybrid Model:

A hybrid model for vulnerability prediction in open-source software (OSS) combines various techniques—typically a mix of machine learning, statistical, and rule-based approaches—to improve accuracy in detecting potential vulnerabilities. By integrating multiple models or methods, a hybrid approach leverages the strengths of each to address the complexities

of vulnerability prediction in OSS projects. Here's a breakdown of how a hybrid model could be applied:

1. Combine Static and Dynamic Analysis

- Static Analysis Models: Use static code analysis tools to evaluate the source code without executing it. These tools identify common security flaws, like buffer overflows, SQL injection, or uninitialized variables, by inspecting code patterns.
- Dynamic Analysis Models: Complement static analysis with dynamic testing (e.g., fuzz testing or sandbox execution) to evaluate how the code behaves at runtime. This can reveal vulnerabilities that may not be apparent in static analysis alone, such as memory leaks or runtime configuration issues.
- Hybrid Approach: Integrate both static and dynamic analysis insights into a model that learns from both, providing a richer dataset for predicting vulnerabilities.

2. Leverage Ensemble Machine Learning Models

- Use ensemble techniques like Random Forests, Gradient Boosting, or Voting Classifiers to combine multiple machine learning algorithms (e.g., decision trees, support vector machines, neural networks).
- Each model captures different aspects of the vulnerability data, and combining their predictions often results in more robust and generalized predictions.

3. Integrate Supervised and Unsupervised Learning

- Supervised Learning: Train models on labeled datasets of known vulnerabilities. Techniques like logistic regression, decision trees, or deep learning models can detect patterns associated with vulnerabilities.
- Unsupervised Learning: Use clustering or anomaly detection methods to identify unusual or suspicious code patterns in the OSS codebase, which might indicate new or previously unknown vulnerabilities.
- Hybrid Learning: Use the unsupervised model to flag anomalous regions, and then apply a supervised model to those flagged regions for further evaluation, increasing detection accuracy.

4. Augment with Natural Language Processing (NLP) for Textual Data

- Use NLP models to analyze commit messages, code comments, and issue-tracking systems, which often provide insights into potential vulnerabilities.
- This information can be integrated into the predictive model to detect vulnerability-prone areas based on language patterns associated with security-related commits or patches.

5. Time-Series and Trend Analysis for Vulnerability Patterns

- Use time-series analysis on historical vulnerability data to predict when and where vulnerabilities are likely to emerge in OSS.
- This can be especially useful in OSS projects with extensive version histories, where vulnerability patterns or cycles may appear over time.

6. Rule-Based and Heuristic Models for Known Vulnerability Patterns

- Apply rule-based methods to detect known patterns of vulnerabilities (e.g., hard-coded credentials, unvalidated input). These rules can act as a first filter.
- The results from rule-based detection can be passed to machine learning models for further refinement, focusing on patterns that are harder to generalize with rules alone.

6. Evaluate Model Performance:

Evaluating the performance of a model for vulnerability prediction in open-source software involves several key metrics and processes to determine how accurately and effectively the model identifies potential vulnerabilities. Here are the primary methods and metrics for evaluating such models:

1. Accuracy and Precision Metrics

- Precision: Measures the proportion of correctly identified vulnerabilities out of all identified vulnerabilities. High precision indicates fewer false positives, which is important in vulnerability prediction to avoid unnecessary patching or resource allocation.
- Recall: Measures the proportion of correctly identified vulnerabilities out of all actual vulnerabilities in the software. High recall is essential to ensure the model captures as many vulnerabilities as possible.
- F1 Score: Combines precision and recall to provide a balanced measure, especially useful when there's an imbalance between positive (vulnerabilities) and negative classes.
- Accuracy: Measures the proportion of true results (both true positives and true negatives) among the total number of cases examined. While accuracy can give a broad sense of model performance, it's less informative if vulnerabilities are rare (class imbalance).

2. Class Imbalance Handling

- In many OSS datasets, vulnerabilities are rare compared to safe code segments, creating an imbalance in classes. Techniques like SMOTE

(Synthetic Minority Over-sampling Technique) or undersampling can help the model learn effectively from both vulnerable and non-vulnerable cases.

3. ROC-AUC (Receiver Operating Characteristic - Area Under Curve)

- ROC-AUC measures the model's ability to distinguish between vulnerable and non-vulnerable components. A higher AUC value indicates better discriminatory power of the model. This metric is valuable for understanding the trade-off between true positive and false positive rates.

4. Confusion Matrix Analysis

- The confusion matrix (TP, TN, FP, FN) provides insights into the distribution of predictions. Analyzing this helps identify patterns, such as whether the model is too conservative (many false negatives) or too lenient (many false positives).

5. Precision-Recall Curve

- For datasets with class imbalance, the precision-recall curve can be more informative than ROC-AUC. It shows how precision and recall vary as the decision threshold changes, which is useful for setting a threshold that balances the cost of false positives and false negatives.

6. Time-Based Evaluation

- Since vulnerabilities evolve, it's important to assess model performance over time. Time-based cross-validation (training on past data and testing on future data) can help evaluate whether the model remains effective at predicting vulnerabilities in newer software versions.

7. Predict Vulnerabilities:

Predicting vulnerabilities in open-source software (OSS) involves using data-driven methods and machine learning to assess the likelihood that specific code elements or components may contain security weaknesses. The key elements of vulnerability prediction include analyzing various factors, patterns, and attributes in the codebase to preemptively identify risky areas.

Here's an outline of approaches and considerations for predicting vulnerabilities in OSS:

1. Data Collection and Feature Extraction

- Historical Data: Gather data from known vulnerabilities, such as Common Vulnerabilities and Exposures (CVE) databases or repositories like GitHub and GitLab, to see past patterns in OSS vulnerabilities.
- Code Metrics: Analyze code complexity, size, and dependencies. Highly complex or large code files, or those with many dependencies, are often more vulnerable.

- Commit and Change History: Track changes in the codebase, frequency of commits, developer activity, and types of changes. Frequent updates or changes by multiple developers may indicate potential weaknesses.
- Contributor Metrics: Evaluate developer experience and familiarity with the codebase, as inexperienced contributions can sometimes introduce vulnerabilities.

2. Machine Learning Models and Algorithms

- Supervised Learning Models: Train models (e.g., decision trees, random forests, or support vector machines) on labeled data (vulnerable vs. non-vulnerable code) to predict vulnerabilities.
- Unsupervised Learning: Use clustering or anomaly detection to identify unusual or potentially risky patterns in the code that could signify vulnerabilities.
- Deep Learning: Apply deep learning models, such as neural networks, to complex code patterns, enabling a more granular analysis, especially useful for large OSS projects with extensive codebases.
- Natural Language Processing (NLP): Use NLP techniques on commit messages, bug reports, or comments to identify terms or phrases commonly associated with vulnerabilities.

3. Static and Dynamic Analysis

- Static Analysis: Check for security flaws without executing the code. Static analysis tools can detect issues like unvalidated inputs, insecure APIs, and poor cryptographic practices.
- Dynamic Analysis: Run the code in a controlled environment to observe its behavior. Dynamic analysis can reveal runtime vulnerabilities, such as buffer overflows and memory leaks.

4. Predictive Factors and Patterns

- Code Smells and Anti-Patterns: Detect patterns known to be associated with vulnerabilities, like “magic values” (hardcoded values) or weak error handling.
- Dependency Vulnerability: Analyze the libraries or dependencies in use. Outdated or unmaintained libraries are often linked to vulnerabilities, as are dependencies with known CVEs.
- Community Factors: Measure the size and activity of the community around the OSS project, as less active projects may lack regular maintenance, leading to increased risk over time.

5. Model Evaluation and Validation

- Precision and Recall: Evaluate models for their accuracy in predicting vulnerabilities (e.g., precision in detecting true positives and recall for avoiding false negatives).

- Cross-Validation: Validate models using cross-validation techniques to ensure consistent performance across different segments of data.
- Continuous Learning: Update models continuously with new data and insights, as software and security landscapes evolve.

Chapter 4

Project Modules

A project on Vulnerability Prediction in Open-Source Software (OSS) using a Hybrid Machine Learning Model would typically be divided into several modules, each addressing a key aspect of the overall system. Here's an outline of the potential project modules:

1. Data Collection and Preprocessing

- Objective: Gather and prepare the data necessary for training the machine learning model.
- Tasks:
    - Collect data from open-source software repositories (e.g., GitHub, GitLab).
    - Gather historical vulnerability data, including reported vulnerabilities, their types, and affected software components.
    - Extract features like code complexity metrics (e.g., cyclomatic complexity, lines of code, etc.), commit history, developer activity, and dependency graphs.
    - Clean and preprocess the data (e.g., removing duplicates, handling missing values, normalizing data).
    - Feature engineering to create meaningful attributes that can be used by the machine learning models.

2. Exploratory Data Analysis (EDA)

- Objective: Understand the data, identify patterns, and determine which features are most relevant for vulnerability prediction.
- Tasks:
    - Visualize the distribution of vulnerabilities across the dataset.
    - Identify correlations between features and vulnerabilities.
    - Analyze the frequency of vulnerabilities across different OSS projects.
    - Perform a class imbalance analysis and determine the need for resampling techniques.

3. Hybrid Model Development

- Objective: Develop the core machine learning model using a hybrid approach (combining multiple models or techniques).
- Tasks:
    - Model Selection: Choose a combination of machine learning models (e.g., decision trees, random forests, support vector machines, neural networks) to create a hybrid model.
    - Ensemble Learning: Use ensemble methods (e.g., bagging, boosting) to combine multiple models for improved prediction accuracy.
    - Feature Fusion: Implement techniques to combine features from different sources (e.g., code metrics and developer activity data) into a unified model.
    - Model Training: Train the model using labeled data (vulnerabilities) and tune hyperparameters to optimize performance.

4. Model Evaluation and Performance Tuning

- Objective: Evaluate the performance of the model using various metrics and optimize its performance.
- Tasks:
    - Split the data into training, validation, and test sets.
    - Use evaluation metrics such as accuracy, precision, recall, F1-score, and area under the ROC curve (AUC) to assess model performance.
    - Perform cross-validation to ensure robustness and avoid overfitting.
    - Implement hyperparameter tuning (e.g., grid search, random search) to find the best model configuration.
    - Compare the performance of the hybrid model with single machine learning models.

5. Vulnerability Prediction and Analysis

- Objective: Use the trained model to predict vulnerabilities in new or unseen open-source software projects.
- Tasks:

- Apply the trained hybrid model to predict vulnerabilities in OSS projects that were not part of the training data.
- Analyze the predicted vulnerabilities and categorize them based on severity, type, and likelihood of occurrence.
- Provide recommendations for developers to address the predicted vulnerabilities.

6. Visualization and Reporting

- Objective: Create user-friendly visualizations to present the results of vulnerability predictions.
- Tasks:
  - Develop visual dashboards to show the prediction results, such as the likelihood of vulnerabilities in different parts of the codebase.
  - Generate heatmaps or graphs to illustrate code areas with the highest risk of vulnerabilities.
  - Provide detailed reports for developers, showcasing the predicted vulnerabilities and suggesting possible mitigation strategies.

7. Deployment and Integration

- Objective: Deploy the model in a real-world environment and integrate it with existing development tools.
- Tasks:
  - Develop a deployment pipeline that integrates the hybrid model into the software development lifecycle (e.g., CI/CD pipelines).
  - Implement an API for easy integration with GitHub, GitLab, or other repositories to automatically predict vulnerabilities during the development process.
  - Provide real-time vulnerability scanning for OSS projects as part of the build or commit process.

8. Monitoring and Maintenance

- Objective: Continuously monitor the model's performance and update it with new data.
- Tasks:

- Set up continuous monitoring to track the model's performance in real-world applications.
- Periodically retrain the model with new data to keep it up to date with emerging vulnerabilities.
- Handle model drift by identifying changes in patterns or distributions in the data.

9. User Feedback and Model Improvement

- Objective: Collect feedback from end-users (developers and security teams) to improve the model's accuracy.
- Tasks:
  - Provide a mechanism for users to report false positives/negatives in the predictions.
  - Use the feedback to refine the model, improving its predictions and handling edge cases.
  - Implement an active learning framework where the model can improve over time by incorporating user feedback.

CHAPTER 5

SYSTEM REQUIREMENTS

When building a system for vulnerability prediction in open-source software (OSS) using a hybrid model in machine learning (ML), several system requirements must be considered. These requirements span hardware, software, and operational aspects to ensure the model can effectively analyze OSS code and predict vulnerabilities. Below are the key system requirements for such a system:

1. Hardware Requirements

- Processor (CPU):
    - High-performance multi-core processors (e.g., Intel i7/i9, AMD Ryzen) are recommended for efficient parallel processing, especially during feature extraction and model training phases.
    - High-performance GPUs (e.g., NVIDIA A100, Tesla V100) may be used for large-scale data processing and deep learning model training, especially if the hybrid model includes deep learning components.
- Memory (RAM):
    - A minimum of 16 GB of RAM is recommended for basic models, while 32 GB or more may be needed for large-scale datasets.
    - RAM is crucial for handling large codebases and running complex machine learning algorithms.
- Storage:
    - Sufficient disk space (e.g., 1 TB SSD or more) to store OSS repositories, vulnerability data, and trained models.
    - Fast SSD storage is preferred for faster data retrieval and model training.
- Network:
    - High-speed internet connection for downloading OSS repositories, vulnerability data, and interacting with cloud services.
    - If using cloud-based training, ensure high-speed networking between local systems and cloud services.

2. Software Requirements

- Operating System:
    - Linux-based operating systems (e.g., Ubuntu, CentOS) are often preferred for development due to better support for machine learning libraries and OSS tooling.
    - Windows or macOS can also be used, but some tools and libraries are optimized for Linux.
- Programming Languages:
    - Python is the most common language for machine learning and data analysis, and it provides libraries like TensorFlow, PyTorch, scikit-learn, and Keras.
    - R may also be used for statistical analysis and predictive modeling.
    - Other scripting languages like Bash or Shell for automation may be needed for data preprocessing.
- Machine Learning Libraries:
    - scikit-learn for traditional machine learning algorithms (e.g., decision trees, random forests, SVM).
    - TensorFlow or PyTorch for deep learning models, if your hybrid model includes neural networks.
    - XGBoost, LightGBM, or CatBoost for gradient boosting techniques in the hybrid model.
    - NLTK, spaCy, or Gensim for natural language processing (NLP) tasks if analyzing code comments, documentation, or other textual data for vulnerability prediction.
- Data Management Tools:
    - Git/GitHub for accessing and managing OSS code repositories.
    - Docker or Kubernetes for containerization and deployment of models, especially in a cloud-based environment.
    - Hadoop/Spark for distributed data processing, especially for large-scale code analysis.
- Database:
    - Relational Databases (e.g., MySQL, PostgreSQL) for storing vulnerability data, model results, and logs.

- NoSQL databases (e.g., MongoDB) may also be used for unstructured data or if flexibility in schema is needed.

3. Data Requirements

- OSS Repositories:
  - A large corpus of open-source repositories (e.g., from GitHub, GitLab, SourceForge) that contain code, version history, and metadata.
  - Access to OSS code with associated vulnerability reports (e.g., Common Vulnerabilities and Exposures - CVE data).
- Vulnerability Data:
  - Historical vulnerability data such as CVE records, security advisories, and patch information.
  - Public datasets (e.g., NVD, OSS-Fuzz, or GitHub Security Advisory API) may be used for training and validation.
- Feature Data:
  - Features might include code metrics (e.g., cyclomatic complexity, lines of code), code review comments, version history, commit logs, dependency analysis, and prior vulnerability occurrences.
  - Textual data such as bug reports, commit messages, and documentation might require NLP processing.

4. Hybrid Model Design Requirements

- Model Architecture:
  - Traditional Machine Learning Algorithms (e.g., Random Forest, SVM) can be used for initial vulnerability prediction using code metrics and historical data.
  - Deep Learning Models (e.g., CNNs, RNNs, LSTMs) can be used for sequential data like commit logs or complex pattern recognition in code structure.
  - Ensemble Models (e.g., stacking, boosting) can combine the strengths of multiple models to improve predictive performance.
- Model Integration:

- A hybrid model often combines several techniques to leverage both structured and unstructured data (e.g., combining code analysis and text data).
- A feature fusion approach may be used where features from different models are merged before final prediction.

- Model Evaluation:
  - Use metrics like accuracy, precision, recall, F1-score, ROC AUC, and confusion matrix to assess the model's performance in predicting vulnerabilities.
  - Cross-validation techniques should be used to ensure model robustness.

5. Operational Requirements

- Automation & CI/CD Pipeline:
  - Integrate vulnerability prediction into CI/CD pipelines to run predictions during code integration and testing.
  - Automation tools like Jenkins or GitLab CI can be used to run tests on new OSS code commits or pull requests.
- Scalability:
  - The system should be scalable to handle an increasing amount of OSS data or growing complexity in vulnerability prediction models.
  - Cloud-based environments (e.g., AWS, Oracle Cloud, Google Cloud) can provide scalable compute and storage resources.
- Monitoring & Logging:
  - Set up logging systems to track model performance and to monitor any data issues.
  - Use monitoring tools (e.g., Prometheus, Grafana) to keep track of the health and performance of the system.
- Security Considerations:
  - Ensure secure access control to the OSS code, data, and prediction models.
  - Protect the system from adversarial attacks that may attempt to manipulate the predictions or codebase.

CHAPTER 6

CONCLUDING REMARKS

In conclusion, leveraging hybrid machine learning models for vulnerability prediction in open-source software presents a powerful approach to enhancing software security. By combining multiple machine learning techniques, such as ensemble learning, deep learning, and natural language processing, these hybrid models can more accurately identify complex patterns and potential vulnerabilities in code. This predictive capability allows developers to proactively address security risks, prioritize remediation efforts, and improve the overall quality of open-source projects. The result is a more secure software development process that reduces the likelihood of exploitation and builds greater trust in open-source ecosystems.

REFERENCES

[1] E. Aghaei, W. Shadid, and E. Al-Shaer, “Threatzoom: Hierarchical neu ral network for cves to cwes classification,” in International Conference on Security and Privacy in Communication Systems. Springer, 2020, pp. 23–41.

[2] ASIC, “Guidance for consumers impacted by the optus data breach,” https://asic.gov.au/about-asic/news-centre/news-items/ guidance-for-consumers-impacted-by-the-optus-data-breach/, 2022.

[3] I. Babalau, D. Corlatescu, O. Grigorescu, C. Sandescu, and M. Dascalu, “Severity prediction of software vulnerabilities based on their text description,” in 2021 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). IEEE, 2021, pp. 171–177.

[4] G. Bhandari, A. Naseer, and L. Moonen, “Cvefixes: automated collec tion of vulnerabilities and their fixes from open-source software,” in Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, 2021, pp. 30–39.

[5] T. Britton, L. Jeng, G. Carver, and P. Cheak, “Reversible debugging software “quantify the time and cost saved using reversible debuggers”,” 2013.

[6] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet,” IEEE Transactions on Software Engineering (TSE), 2021.

[7] Checkmarx, “Checkmarx,” https://checkmarx.com/, 2006.

[8] Z. Chen, S. Kommrusch, and M. Monperrus, “Neural transfer learning for repairing security vulnerabilities in c code,” IEEE Transactions on Software Engineering (TSE), 2021.

[9] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, “Sequencer: Sequence-to-sequence learning for end-to end program repair,” IEEE Transactions on Software Engineering (TSE) , vol. 47, no. 9, pp. 1943–1959, 2019.

[10] R. Croft, D. Newlands, Z. Chen, and M. A. Babar, “An empirical study of rule-based and learning-based approaches for static application security testing,” in Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2021, pp. 1–12.