الجمهورية الجزائرية الديمقراطية الشعبية
People's Democratic Republic of Algeria
وزارة التعليم العالي والبحث العلمي
Ministry of Higher Education and Scientific Research
المدرسة العليا للإعلام الآلي - 80 ماي 5491 – بسيدي بلعباس
Higher School of Computer Science -8 Mai 1945- Sidi Bel Abbès
(ESI-SBA)

END OF STUDY THESIS

To obtain a **state engineer's** diploma
Stream: **Computer Science**
Speciality: **Information Systems Engineering (ISI)**

Static Analysis For Early Detection Of Vulnerabilities in Source Code Based on Artificial Intelligence

---

Presented by:
Mr Remmane Mohamed

Presented on: **30/09/2025** In front of a committee composed of :

Dr. Belfedhal Alaa Eddine	Supervisor
Dr. Serhane Oussama	Co-Supervisor
Dr. Chikh Asma	President
Dr. Neggaz Imene	Examiner

Academic year
2024/2025

Acknowledgements

First and foremost, Alhamdulillah, all praise and gratitude are due to Allah (SWT) for granting me the strength, patience, and perseverance to undertake and complete this thesis. Without His blessings and guidance, this achievement would not have been possible.

I would like to express my deepest gratitude to my parents, whose unconditional love, encouragement, and sacrifices have been the driving force behind all of my academic pursuits. Their constant prayers and support have given me the motivation to persevere through every challenge.

My sincere appreciation goes to my supervisor, Mr. Belfedhal Alaa Eddine, for his invaluable guidance, patience, and support throughout the course of this work. His insightful feedback, constructive criticism, and encouragement have greatly contributed to shaping this thesis. It has been a privilege to benefit from his expertise and dedication.

I would also like to extend my heartfelt thanks to my co-supervisor, Mr. Serhane Oussama, for his valuable support and guidance throughout this work. In addition, I sincerely thank the jury members for taking the time to evaluate my work. Their insightful remarks, critical observations, and valuable suggestions have greatly contributed to improving the quality of this thesis.

A special thanks is extended to my colleagues and friends, with whom I shared countless hours of discussions, collaboration, and encouragement. Their companionship and moral support made this journey less difficult and more enjoyable.

Finally, I would like to acknowledge my own effort, determination, and dedication throughout this journey. Completing this thesis has been both a challenge and a rewarding experience that has allowed me to grow academically and personally.

Abstract

Software security has become a critical concern as modern applications grow increasingly complex and interconnected. Traditional static analysis tools, while widely adopted for detecting vulnerabilities in source code, often suffer from high false-positive rates and limited ability to capture deep semantic relationships in programs. To address these limitations, this project replicates and adapts a deep learning architecture from a state-of-art Research paper which achieved a great results relatively to other reseach papers for classifying Java functions as safe or vulnerable. The model leverages program representations to learn semantic and structural patterns in source code, enabling more accurate vulnerability detection.

Beyond model development, this work focuses on practical integration. A web application was implemented to provide an accessible interface for testing and analyzing Java code, while a dedicated Visual Studio Code extension was developed to bring vulnerability detection directly into the developer workflow. Together, these tools demonstrate how deep learning can be effectively applied in real-world development environments to assist programmers in identifying and mitigating security risks.

This project highlights the potential of combining deep learning with engineering solutions to advance automated vulnerability detection, bridging the gap between research prototypes and practical software security tools.

Keywords: Static Code Analysis, Vulnerability Detection, AST, DFG, CFG, DL, ML, Software Vulnerability Detection, Graph Neural Networks (GNNs), Java Security, Web Application, VS Code Extension.

Contents

List of Figures

List of Tables

General Introduction

Software security has become a major concern as applications grow more complex and interconnected. Vulnerabilities in source code can be exploited by attackers, causing significant financial and social damage. Detecting these flaws early in the development process is therefore essential.

Traditional approaches such as static and dynamic analysis have been widely adopted. While static analysis is scalable and useful for scanning code without execution, it often produces a high number of false positives and struggles to capture semantic dependencies. Dynamic analysis, meanwhile, is more accurate but limited by runtime overhead and scalability issues. These challenges have motivated the exploration of machine learning (ML) and deep learning (DL) to improve vulnerability detection.

Problem statement

Software vulnerability detection remains a persistent challenge despite significant progress in both academic research and industrial practice. Traditional static analysis tools are widely used due to their scalability and ability to analyze code without execution. However, they often generate a large number of false positives, overwhelming developers with alerts and reducing trust in their results. Additionally, static analysis is limited in its ability to capture deeper semantic relationships within code, which are often essential for detecting complex vulnerabilities.

In recent years, machine learning (ML) and deep learning (DL) approaches have been proposed as promising alternatives. These methods aim to automatically learn patterns of vulnerable code from large datasets, improving detection accuracy and adaptability. However, they face several unresolved challenges. First, **the quality of datasets** remains problematic: many benchmark datasets are small, synthetic, or suffer from noisy and mislabeled samples, which limits generalization to real-world scenarios. Second, **the reproducibility of ML/DL** research in this domain is often hindered by inconsistent preprocessing steps, lack of standardized evaluation metrics, and absence of open-source implementations. Third, **model generalization** remains an obstacle, as models trained on specific datasets often fail to adapt to different programming styles, frameworks, or unseen vulnerability types.

Objective

The aim of this work is twofold: first, title to replicate a deep learning architecture capable of classifying Java functions as either safe or vulnerable, this architecture that leverages multiple code representations to enhance vulnerability detection in Java source code including Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs). The architecture also employs advanced deep learning concepts such as multi-head attention and transformers. And second, to integrate this model into a practical environment by embedding it within a web application and developing a Visual Studio Code extension. This integration allows developers to detect potential vulnerabilities directly during the coding process, thereby providing real-time assistance and contributing to more secure software development practices.

Thesis Organization

- **Chapter 01**: introduces Java programming and common vulnerabilities affecting it.
- **Chapter 02**: this chapter provides background on static code analysis.
- **Chapter 03**: this chapter provides background on machine learning, and deep learning concepts relevant to this work.
- **Chapter 04**: this chapter provides detailed the imp lementation of the proposed solution.
- **Chapter 05**: this chapter showcases the overview of the experimental evaluation and results of our solution.

Part I

Background

Chapter 1

Java and Its Vulnerabilities

1.1 Introduction

Java is one of the world's most widely used programming languages and platforms, known for its **portability**, **object-oriented features**, and **security**. Its security architecture is a layered one, built into the language and the runtime environment to provide a robust framework against various threats. This post delves into the core aspects of Java's Security Architecture, exploring its components, how they function, and why they are essential in today's digital world.

Java [33] was designed by James Gosling at Sun Microsystems. It was released in May 1995 as a core component of Sun's Java platform. The original and reference implementation Java compilers, virtual machines, and class libraries were released by Sun under proprietary licenses. As of May 2007, in compliance with the specifications of the Java Community Process, Sun had relicensed most of its Java technologies under the GPL-2.0-only license. Oracle, which bought Sun in 2010, offers its own HotSpot Java Virtual Machine. However, the official reference implementation is the OpenJDK JVM, which is open-source software used by most developers and is the default JVM for almost all Linux distributions.

Java 24 is the version current as of **March 2025**. Java 8, 11, 17, and 21 are long-term support versions still under maintenance.

1.2 Java Security Model

Since its inception, **Java** has prioritized security, establishing itself as one of the most robust programming languages in this domain.[30] Its architecture is built around a comprehensive security model, making it suitable for a different types of applications—from lightweight mobile apps to complex enterprise-level systems. It ensures that:

- Untrusted code cannot harm the system.
- Sensitive data remains protected.
- Only authorized actions will be executed.

Java uses a combination of security policies, classloaders, the sandbox model, and cryptographic APIs to enforce security in applications.

1.2.1 The Sandbox Security Model

The Java Security Model provide a controlled execution environment for the purpose of testing the code's trustworthiness by limiting code's access to system resources such as system memory, file system , preventing any unauthorized actions on the host system[3].
Example: If a Java applet tries to read a local file, Java's security manager blocks it unless explicitly allowed.
While applets are no longer common, the same principles apply to modern applications that need to run untrusted code safely.

1.2.2 The ClassLoader Mechanism

The **ClassLoader**[3] in java is a subsystem of **Java Virtual Machine (JVM)** that safely load class files. Java enhances security by dividing class loaders into two types: **system class loader** and **user-defined class loader**. The system class loader loads classes from the local file system, which are typically trusted. On the other hand, **user-defined** class loaders may load classes from remote locations (like over the network), which are potentially untrusted.

The class loader mechanism enforces a strong namespace separation, which means that even if a class from an untrusted source tries to spoof a core Java class **(Loading malicious classes with the same name as standard Java class)**, the JVM can differentiate them by their class loader namespace. This prevents many

types of attacks that rely on overriding the standard library classes with malicious ones.
Exemple: Java does not allow an untrusted class to override system classes like **java.lang.String**.

1.2.3 Bytecode Verifier

the **Bytecode Verifier**[3] is a critical component of the sandbox model. Before the JVM execute the bytecode, the verifier goes through the bytecode and checks that it follows Java's safety rules such as: the bytecode structure, type safety, type confusion attacks. Also **The verifier** simulates how the bytecode would behave at runtime and checks that the operand stack and local variables remain consistent. This prevents illegal operations from happening and keeps the JVM from crashing due to unexpected behavior.

1.2.4 The Security Manager and Access Controller

The **SecurityManager** (now deprecated)[3] is what controls access to system resources. Whenever an application tries to perform a sensitive operation, such as reading a file or opening a network connection, the Security Manager decides whether this should be allowed by checking the permissions of that specific resource.

The **Access Controller** provides a more granular level of security by complementing the Security Manager. Rather than making decisions solely based on the type of operation, it also considers the execution context—such as the origin of the code and the class loader involved. Using a stack inspection algorithm, it evaluates each method in the call stack, granting permission only if all methods in the chain have the necessary privileges.

1.2.5 Java Permissions and Policy Files

Java Security Model[30] uses **policy files** to define permissions that specify which actions are permitted based on where the code originates.The **AccessController** checks these permissions at runtime and throws a **SecurityException** if the requested operation is not allowed.

Java's permission model is flexible and allows fine-grained control over what actions are permitted. Developers can define custom policies that restrict file I/O, network connections, and access to system properties.

1.2.6 Java Cryptography API

Java provides built-in security libraries for cryptographic operations and secure communications:

- **Java Cryptography Architecture (JCA)**[3] – For encryption (AES, RSA), hashing (SHA-256) and key management.
- **Java Secure Socket Extension (JSSE)**[30] – Implements TLS/SSL for secure communication over networks.
- **Java Authentication and Authorization Service (JAAS)**[20] – Role-based access control.

Java Cryptography Architecture (JCA): consists of a set of APIs and implementations that offers variety of functionalities: **Provider Architecture, Key Management, Encryption and Decryption**.
Provider Architecture is defined as a set of packages that a set of cryptographic services, algorithms, and keys. **Key Management** simply provides classes and interfaces for key generation, key agreement, KeyFactory for converting keys into a key specification. Also **JCA** provides a set of APIs for **Encryption Decryption** and **Digital Signatures** for integrating cryptographic functionalities into Java applications and ensuring the integrity and authenticity of the data.

Java Secure Socket Extension (JSSE) for Network Security: provides a set of packages that support and implement the SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols which are widely used for securing data transmitted over networks, particularly the internet. **JSSE** is used in a wide array of applications such as web browsers, servers, HTTPS, and in any Java application that requires secure communication.

Java Authentication and Authorization Service (JAAS): is a Java's built-in framework for handling **authentication** (verifying user identity) and **authorization** (controlling user access). It separates these concerns into two phases:

- **Authentication**: by confirming or denying a subject's claimed identity. A subject typically represents an entity like a user, a process, or a device attempting to gain access to a system.
- **Authorization**: Once authentication is successfully completed, the authorization phase begins. JAAS relies on the authenticated subject's principals—system-recognized identities like usernames or group memberships—to determine the actions the subject is permitted to perform. These principals are linked to specific permissions as defined in the application's security policy and at runtime, whenever a subject tries to execute a protected operation, the policy is consulted to verify whether the subject has the necessary rights to proceed.

1.3 Common Java Vulnerabilities

Java is one of the most widely used programming languages in the world, particularly in enterprise, mobile, and web development. Like any programming language, however, it is not immune to security vulnerabilities.[21] The following is an overview of common techniques attackers use to exploit weaknesses in Java applications. While not exhaustive, understanding how these vulnerabilities arise is essential for developing and maintaining secure software.

1. **SQL Injection:** attacks are the process of injecting malicious SQL within data requests and gaining access to the database through them and this could result in the server giving back sensitive data or executing malicious scripting content to modify data on the database for different purposes from stealing data to data loss/corruption.
2. **Command Injections:** attacks come from user inputs that execute shell commands on the server running your application which can give the attackers the ability to execute system commands that can harm your operating system or lead to sensitive data leaks ranging from user data to administration-related information.

3. **Cross-site scripting (XSS)**: vulnerabilities occur when attackers inject malicious scripts into web applications, targeting users of the site. The two main types are:

    - **Reflected XSS:** The malicious script is included in a URL and executed when the user clicks it. It relies on immediate reflection of user input by the server and can trick users into running harmful scripts on trusted sites.
    - **Stored XSS:** More dangerous, this involves malicious scripts being saved on the server (e.g., in comments or profiles) and automatically executed when other users load the affected page.

4. **XPath injections:** attacks are similar to SQL injection but targets XML data. It occurs when attackers exploit improperly validated user inputs to inject malicious XPath queries, potentially gaining access to sensitive information like usernames and passwords. This type of attack is very dangerous, and it's becoming more prevalent as more organizations adopt microservices architecture. Because microservices involve many small applications accessing XML resources, organizations using this infrastructure are especially vulnerable to this sort of attack.

5. **Remote code execution (RCE):** is considered one of the most dangerous vulnerabilities where an attacker can remotely execute commands on someone else's device or run malicious code on the target system for the purpose of deploying additional malware or stealing sensitive data. The famous **log4j** exploit is a disastrous example of such an attack.

6. **LDAP Injections:** The Lightweight Directory Access Protocol **(LDAP)** is a popular software protocol for directory service authentication.If you don't have parameters around your LDAP query interfaces, hackers may insert LDAP queries in user input forms, which allow them to look up otherwise private information within your database. LDAP injections may also allow attackers to change users' permissions, status, and privileges.

7. **Resource injections:** occur when the attacker successfully changes the resource identifiers used by the Java application to execute scripts. This might be changing the port number, modifying the file name, and gaining the ability to execute or access other resources and this usually occurs when the application defines a resource via user input.

8. **Connection String Injection:** occurs when attackers manipulate connection strings—used to link an application to data sources like databases or LDAP directories by injecting malicious parameters. By exploiting database systems that accept duplicate parameters and follow a ”last one wins” logic, attackers can bypass authentication and gain unauthorized access to sensitive data. This type of attack often involves inserting semicolon-separated values such as data source, catalog, user ID, and password into the connection string.

1.3.1 CVE (Common Vulnerabilities and Exposures)

Common Vulnerabilities and Exposures (CVE) is a publicly accessible database that identifies and catalogs known security vulnerabilities in software and hardware. Each vulnerability is assigned a unique ID, making it easier for organizations to share information, prioritize fixes, and protect their systems.CVE helps organizations identify and prioritize security issues with documented CVE numbers and **CVSS**[1] scores to plan and prioritize their **vulnerability management programs**[2].

In 1999, the CVE system was established to make it easier to identify security problems. This system is managed by **MITRE** Corporation and supported by the U.S. government. It provides a common way to share and compare information about security issues. Before CVE, different companies had their own databases with different ways of identifying problems, making it hard to compare information. **CVE** fixed this by providing a standard system that everyone can use.

1.3.2 Some Java CVEs

Theses some examples of high-profile Java CVE, you can find all these CVEs here **NVD**[3] :

CVE-2021-44228 – Log4Shell

One of the most critical Java vulnerabilities discovered in recent years is **CVE-2021-44228**, also known as *Log4Shell*. Log4j2, a widely used logging library in Java applications, includes support for Java Naming and Directory Interface (JNDI) lookups within log messages. When message lookup substitution is enabled, attackers can exploit this feature by injecting malicious log messages that reference external JNDI

[1]https://www.balbix.com/insights/understanding-cvss-scores/
[2]https://www.balbix.com/insights/what-is-vulnerability-management/
[3]https://nvd.nist.gov/search

resources (e.g., via LDAP). This allows arbitrary code execution when the application retrieves and executes code from a remote LDAP server.
Impact: Remote Code Execution (RCE) **Severity**: **CVSS** v3 Score of 10.0 (Critical).

CVE-2022-22947 (Spring Cloud Gateway)

A code injection vulnerability in **Spring Cloud Gateway** versions prior to 3.1.1+ and 3.0.7+ that leads to Remote code Execution on the remote host when the Gateway Actuator endpoint is unsecured.
Severity: CVSS v3 Score of 10.0 (Critical)

CVE-2022-36067

Spring MVC controller methods with an **@RequestBody byte[] method parameter** are vulnerable to a DoS attack.
Severity: **CVSS** v3 Score of 5.3 (Medium).

CVE-2023-43192

an Improper input validation in the **SpringbootCMS 1.0** background allows for SQL injection, where attackers can exploit this by submitting malicious input with special characters to execute any SQL Statement.
Severity: CVSS v3 Score of 8.8 (High)

CVE-2020-6309

Certain versions of **SAP NetWeaver AS JAVA** lack authentication checks for a specific web service, allowing attackers to send malicious payloads and cause a Denial of Service (DoS).
Severity: CVSS v3 Score of 7.5 (High)

CVE-2018-17865

A Cross-Site Scripting (XSS) vulnerability in SAP **J2EE** Engine 7.01 allows remote attackers to inject arbitrary scripts through the **wsdlPath** parameter in the **/ctcprotocol/Protocol** endpoint.
Severity: CVSS v3 Score of 6.1 (Medium)

CVE-2018-17862

A Cross-Site Scripting (XSS) vulnerability in SAP **J2EE** Engine 7.01 (Fiori) allows remote attackers to inject arbitrary web scripts by manipulating the **sys_jdbc** parameter in the **/TestJDBC_Web/test2** endpoint.
Severity: CVSS v3 Score of 6.1 (Medium)

CVE-2024-45761

In 11.0.1.0 version and prior versions of Dell OpenManage Server Administrator contains an improper input validation vulnerability that allows a remote low-privileged malicious user to load any web plugins or Java class leading to the possibility of altering the behavior of certain apps/OS or Denial of Service attack.
Severity: CVSS v3 Score of 8.1 (High)

CVE-2018-14721

FasterXML jackson-databind 2.x before 2.9.7 might allow remote attackers to conduct server-side request forgery (SSRF) attacks by leveraging failure to block the axis2-jaxws class from polymorphic deserialization.
Severity: CVSS v3 Score of 10.0 (Critical)

1.3.3 CWE (Common Weakness Enumeration)

According to [18], **Common Weakness Enumeration (CWE™)** is a community-developed list of common software and hardware weaknesses. A "weakness" is a condition in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities. The CWE List and associated classification taxonomy identify and describe weaknesses in terms of CWEs. Knowing the weaknesses that result in vulnerabilities means software developers, hardware designers, and security architects can eliminate them before deployment, when it is much easier and cheaper to do so.

1.3.4 Some Java CWEs

Some of the most common **CWEs** relevant to Java From the Top 25 CWE 2024 **List**[4] include:

[4]https://cwe.mitre.org/top25/archive/2024/2024$_{c}we_{t}op25.html$

CWE-79:

Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'): Common in **Java Web Frameworks** and could lead to the execution of Unauthorized Code or Commands, and reading Application Data.

CWE-89:

Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'): heavily affects java because Popular Java Frameworks like **Spring JDBC, JPA** interact with databases making it a critical risk.

CWE-78:

Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'): The vulnerability allows attackers to execute unauthorized code, cause denial of service (DoS), read or modify files and application data, and potentially hide malicious activities.

CWE-22:

Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'): occur when attackers manipulate file paths using special elements like **"../"** or absolute paths to escape a restricted directory and access unauthorized files and Without proper input validation, attackers can exploit these features to read or write unauthorized files, leading to data breaches or system compromise.

CWE-20:

Improper Input Validation:a frequent root cause of many injection and logic errors attacks such as: **RCE, Command Injection, SQLi....** .

CWE-502:

Deserialization of Untrusted Data refers to insecure deserialization, where software processes data from untrusted sources without proper validation. This can allow attackers to manipulate objects and perform actions like remote code execution, denial of service, or access control bypass. A notable example is the **Log4Shell** vulnerability in Java's Log4j library, which exploited this weakness to execute arbitrary commands remotely.

CWE-476:

NULL Pointer Dereference: occurs when a program attempts to access or use a memory location and this vulnerability can lead to application crashes, denial-of-service (DoS), or in some cases, arbitrary code execution. It often results from improper Input Validation or making assumptions about object initialization without verification.

CWE-352:

Cross-Site Request Forgery (CSRF): occurs when a malicious website tricks a user's browser into performing an unwanted action on a web application in which the user is already authenticated. Built Web applications, with frameworks like **Spring MVC, Spring Boot, Java EE (Jakarta EE), and JSF** often involve session-based authentication, which is the primary condition that enables **CSRF attacks**.

CWE-789:

Use of Hard-coded Credentials: refers to the risky practice of embedding sensitive credentials **(passwords, API keys, tokens)** directly in source code, making them easily exploitable.

CWE-434:

Unrestricted Upload of File with Dangerous Type: Uploading Files without a proper validation in java web applications could lead to Uploading **JSP** files that contain malicious code that allow for Remote Code Execution attack on the target system.

1.3.5 Mitigation and Defense Strategies

Java is a solid choice for developing robust and scalable applications, but No programming language is completely secure by default due to Insecure coding practices by developers. This subsection outlines practical mitigation techniques and secure coding practices to mitigate vulnerabilities and increase the Security status of Java applications[21]:

Input Validation and Sanitization

Proper validation helps prevent security threats such as SQL injection and Cross-Site Scripting (XSS). One essential technique is query parameterization, which ensures

user inputs do not alter the intended structure of queries.
Code Exemple: Using parameterized queries to prevent From SQL injection.

```
public void prepStatmentExample(String parameter) throws SQLException {
   Connection connection = DriverManager.getConnection(DB_URL, USER, PASS);
   String query = "SELECT * FROM USERS WHERE lastname = ?";
   PreparedStatement statement = connection.prepareStatement(query);
   statement.setString(1, parameter);
   System.out.println(statement);
   ResultSet result = statement.executeQuery();
   printResult(result);
}
```

also it is recommended to use input validation libraries or frameworks, such as OWASP ESAPI.

```
// in your artifacts file
    <dependency>
   <groupId>org.owasp.encoder</groupId>
   <artifactId>encoder</artifactId>
   <version>1.2.2</version>
</dependency>
// Code Example
String untrusted = "<script> alert(1); </script>";
System.out.println(Encode.forHtml(untrusted));

// output: <script> alert(1); </script>
```

Using Security libraries and frameworks

It is Highly recommended to use the **Built-in Java** Security libraries and APIs since they provide a tested and maintained security features such as: **Encryption and Decryption** libraries for protecting sensitive data during transmission or storage such as **Passwords, users data....**, or **input/output** validation that uses prepared statements for database queries and escaping special characters to prevent user data from executing as code.

Authentication and Authorization

Authentication:

Java frameworks like **Spring Security** and **Jakarta EE** enable robust user authentication mechanisms, ensuring only authorized users access sensitive resources.
Code Example: of using **JAAS** authentication Modules

```
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

public class JAASAuthenticationExample {
    public static void main(String[] args) {
        try {
            // Create a LoginContext with a callback handler
            LoginContext lc = new LoginContext("Sample", new MyCallbackHandler());

            // Attempt to authenticate the user
            lc.login();

            // If authentication succeeds, proceed with the application logic
        } catch (LoginException le) {
            System.err.println("Authentication failed: " + le.getMessage());
        }
    }
//The MyCallbackHandler class would implement CallbackHandler and handle prompts
}
```

Also Setting up OpenID Connect OIDC is a straightforward and solid option as it works well with frameworks like **Spring Security** and it allows you to use third party clients like **Google, Github** to handle your OIDC Authentication.

Authorization:

Access control, driven by roles or attributes, dictates who can perform specific actions or access particular data. It guarantees that users only access authorized resources.

Code Example: of Using **Spring Security** for defining rules that that control access to different parts of an application based on user roles:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            .and()
            .httpBasic();
    }
}
```

Here only users with **"ADMIN"** roles can access the **/admin** endpoint

Security Monitoring and Network Protection in Java Application

Java enhances application security through monitoring, logging, and network protection. It supports logging of security events and integration with log analysis and intrusion detection systems to identify threats. Java also enables vulnerability scanning and penetration testing. On the network side, it facilitates the configuration of firewalls, traffic filters, and supports SSL/TLS encryption for secure data transmission.

Avoid hard-coding sensitive data:

It is not recommended to store sensitive data such as **API keys, passwords, or encryption keys** directly in the source code. A more secure approach is to use environment variables or protected configuration files with proper access controls to manage such information.

Code Example:

```
String dbPassword = System.getenv("THE_DB_PASSWORD");
```

Regularly Update Dependencies:

Keeping your libraries and frameworks up to date to avoid vulnerabilities in outdated versions. Using a dependency management tool like **Maven or Gradle** helps you do that.

Security Testing and Maintenance:

It is crucial to regularly conduct **VAPT**[5]**, Code reviews and Vulnerability Scanning** for detecting and mitigating vulnerabilities in Java applications before potential exploitation by attackers. Also maintaining up-to-date security patches and secure error handling are essential practices to prevent exploitation and enhance overall application security.
Examples:

- Integrate static code analysis tools like **SonarQube** into your CI/CD pipeline.
- vulnerability scanning tools such as: **Synk**

Education and Training:

Java developers should receive thorough training in secure coding practices and cybersecurity awareness to effectively understand potential threats and the best strategies for mitigating them. By leveraging Java's security features and integrating robust security practices throughout the software development lifecycle, organizations can proactively detect, prevent, and respond to cyberattacks, thereby strengthening the overall security of their Java-based applications and systems.

[5]VAPT stands for Vulnerability Assessment and Penetration Testing

1.4 Conclusion

This chapter provided a comprehensive overview of the Java security model and common vulnerabilities affecting Java applications. By analyzing real-world **CVEs and CWE** patterns, we highlighted the importance of early detection and mitigation using various tools and best practices. Understanding these fundamentals is essential before exploring how **Machine and Deep Learning** can enhance the process of **Java vulnerability detection and Java Static Code Analysis**, which will be addressed in the following chapters.

Chapter 2

Static Code Analysis

2.1 Introduction

Static Code Analysis (SCA), is defined as the automatic process that attempt to highlight possible vulnerabilities within 'static' (non-running) source code by using techniques such as **Taint Analysis and Data Flow Analysis**. It plays a crucial role in detecting vulnerabilities early in the **software development life cycle (SDLC)**, before they manifest during execution.

By systematically analyzing the structural and semantic properties of code, static analysis tools can uncover a broad range of issues such as insecure API usage, improper input validation, and logic vulnerabilities. In Java, a language frequently used for building enterprise-grade and security-sensitive applications, SCA offers an effective means to enforce security best practices and coding standards. However, despite its utility, traditional static analysis techniques face notable challenges, including limited contextual awareness and high false-positive rates.

This chapter presents an overview of static code analysis, explores its different techniques, compares **SCA** with the **Dynamic Approach (DPA)**[1], mentions some static analysis tools for java programming language, and highlights the potential of integrating artificial intelligence to overcome current shortcomings.

[1] In the process of the DPA, the program is evaluated by executing it

2.2 SCA Techniques

To understand the mechanics of static analysis, it is important to explore the key techniques and internal representations used in modern tools:

2.2.1 Control flow Analysis

is a technique used to understand all the possible execution paths a program might take during runtime — without actually executing it. It helps identify how the control moves through the program's statements, loops, conditionals, and function calls and helps identify bugs like **infinite loops and unreachable code**. The end goal of this technique is to construct the **Control Flow Graph (CFG)** which will be employed within **data-flow analysis and Taint Analysis**.

2.2.2 Data Flow Analysis

This technique involves tracking the **flow of data** through the code, in order to identify potential issues such as **uninitialized variables, null pointers, and data race conditions**.

Data Flow Analysis operates on the Code's **control flow graph (CFG)**, which represents all possible paths that the execution might take through the code. Each **node**(aka a Block of Code) in the CFG corresponds to a program statement or instruction, and edges represent possible execution paths.
Code Example:

```
int x;
if (userInput != null) {
    x = 5;
}
System.out.println(x);
```

A data flow analyzer might flag this because **x** could be used without being initialized if **userInput** is null. Even though Java will throw a compile-time error, **DFA** can help detect similar issues in more complex cases, especially in languages or configurations where such checking is weaker.

Taint Analysis:

Taint Analysis is a **type of Data Flow Analysis** used to track the flow of potentially harmful or untrusted data **("Tainted")** through a program to assess its

security. It involves three main stages: identifying **taint** sources (e.g., user input), analyzing how tainted data propagates, and detecting where it is used without proper **sanitization**. Widely applied in detecting vulnerabilities like information leakage and command injection. Some Concept in **Taint Analysis** to be defined:

- **Source:** a point where data enters the program (e.g., request.getParameter() in Java).
- **Sink:** a dangerous operation where tainted data could cause harm (e.g., SQL queries, file I/O).
- **Taint:** Data derived from an untrusted source (e.g., user input).
- **Sanitizer:** Functions that clean tainted data (e.g., input validation, escaping).

Taint Analysis attempts to identify variables that have been **'tainted'** with user controllable input and traces them to possible vulnerable functions also known as a **'sink'**. If the tainted variable gets passed to a sink without first being **sanitized** it is flagged as a vulnerability. **Code Example of SQLi:**

```
public class SQLInjectionExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter username: ");
        // the Tainted Variable
        String username = scanner.nextLine();

        try {
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3
            Statement stmt = conn.createStatement();

            // Vulnerable to SQL Injection
            String query = "SELECT * FROM users WHERE username = '" + username + "'
            // Sink
            ResultSet result = stmt.executeQuery(query);

            if (result.next()) {
                System.out.println("User found: " + result.getString("username"));
            } else {
```

```
                System.out.println("User not found.");
            }

            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Note: there are also other types of DFA techniques such as **Reaching Definitions, Live Variable Analysis, Use-definition Analysis**

2.2.3 Lexical Analysis/Pattern Matching

Lexical analysis, also known as syntactic analysis or pattern matching, is a technique used in static analysis tools like **ITS4, Flawfinder, and RATS**. It works by tokenizing the source code and searching for predefined patterns or insecure functions (e.g., strcat, gets in C).[26].
simple Tokenization Example:

```
String name = "John";
```

TYPE_IDENTIFIER: (String)
VARIABLE_IDENTIFIER: (name)
SYMBOL: (=)
STRING_LITERAL: ("John")
SYMBOL: (;)

2.3 Code Representation

In this section, we will showcase some of the most used **Source Code Representation graphs** with a Code example for demonstration purposes:

2.3.1 Control Flow Graph (CFG)

is a graphical representation of the control flow or sequence of computations during a program's execution. It is structured as a directed graph, where **nodes** represent

basic blocks—sequences of instructions without branches—and edges indicate possible transitions between these blocks. A CFG typically includes two special nodes: an **entry block**, where execution begins, and an **exit block**, where it ends. This structure helps illustrate how various parts of a program or system manage the flow of information from start to finish.[8]

Example of a CFG from this simple java code:

```
public class FlowGraphExample {
    public static void main(String[] args) {

        int w, x = 1, y = 2, z = 3;

        // Block B1
        w = 0;
        x = x + y;
        y = 0;

        // Conditional branch
        if (x > z) {
            // Block B2
            y = x;
            x++;
        } else {
            // Block B3
            y = z;
            z++;
        }

        // Block B4 (executed after either B2 or B3)
        w = x + z;
    }
}
```

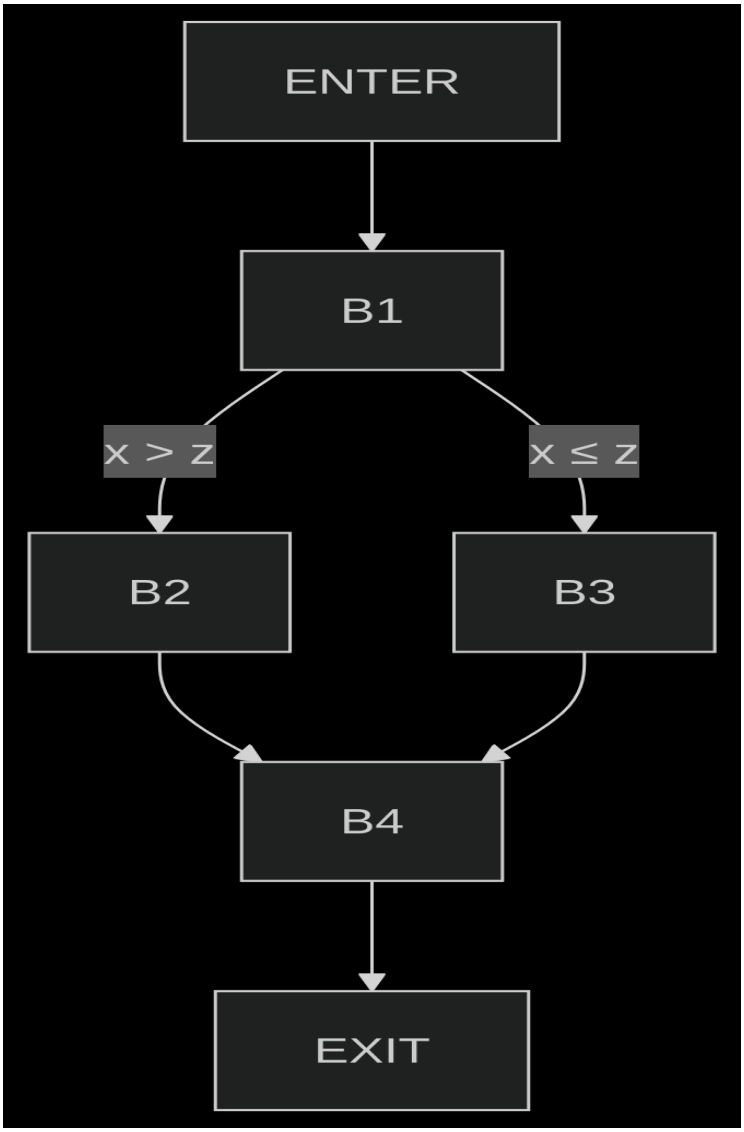


Figure 2.1: Control Flow Graph example

Figure2.1 illustrates an example of a **CFG**. Block B1 represents an if statement and branches into two distinct blocks based on the condition (x > z): B2 if the condition is true, or B3 otherwise.
Some tools that generate **CFG** from Java Source Code: WALA framework[31], SOOT[28], Joern[16].

2.3.2 Abstract Syntax Tree (AST)

is a tree-like representation of the syntactic structure of source code. Unlike raw source code or tokens, an AST abstracts away unnecessary details (like parentheses or semicolons) and focuses on the hierarchical relationships between language constructs (e.g., classes, methods, loops, conditions).It is widely used in compilers,

interpreters, and static analysis tools because it captures both the structure and semantics of a program in a machine-readable form.

Example of an AST from this simple java code:

```
public class HelloWorld {
    public static void main(String[] args) {
        int x = 5;
        if (x > 0) {
            System.out.println("Positive");
        }
    }
}
```

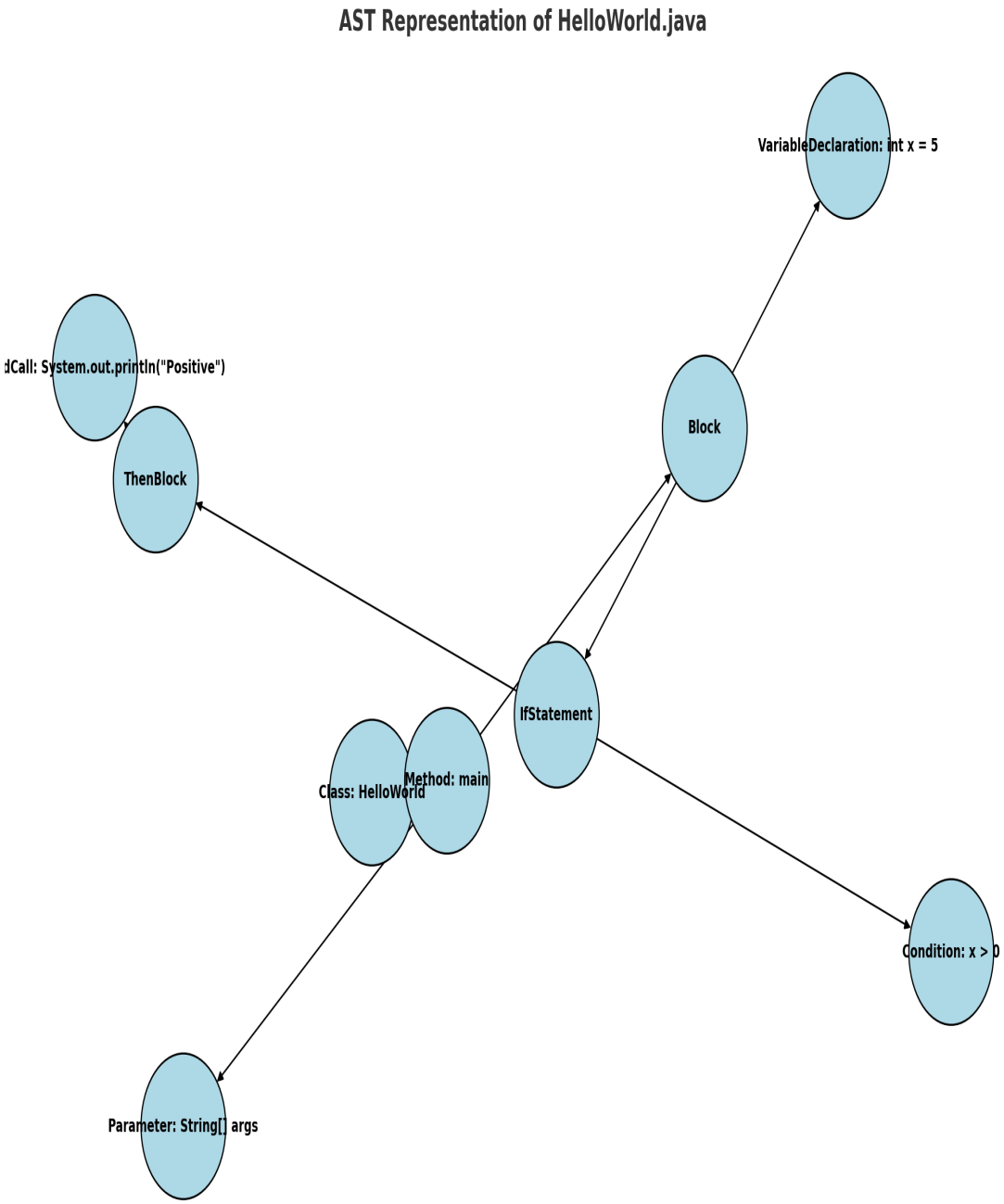


Figure 2.2: Abstract Syntax Tree example

Figure2.2 illustrates an example of a **AST**.The root is a ClassDeclaration node for HelloWorld. Inside it, a MethodDeclaration node represents the main method.The method's block has two child nodes: A VariableDeclaration for int x = 5 and an IfStatement with a condition (x > 0) then a block containing a method call (System.out.println("Positive")).

the most recommended tool For generating **AST** from Java Source Code by research is: JavaParser[15]

2.3.3 Data Flow Graph (DFG)

is a representation of the flow of data within a program. Nodes represent operations or statements, and edges represent the flow of data between them. Unlike a CFG, DFG focuses on the dependencies of data rather than control flow.
Example of a DFG from this simple java code:

```
public class Example {
    public static void main(String[] args) {
        int a = 2;
        int b = 3;
        int c = a + b;
        int d = c * b;
        System.out.println(d);
    }
}
```

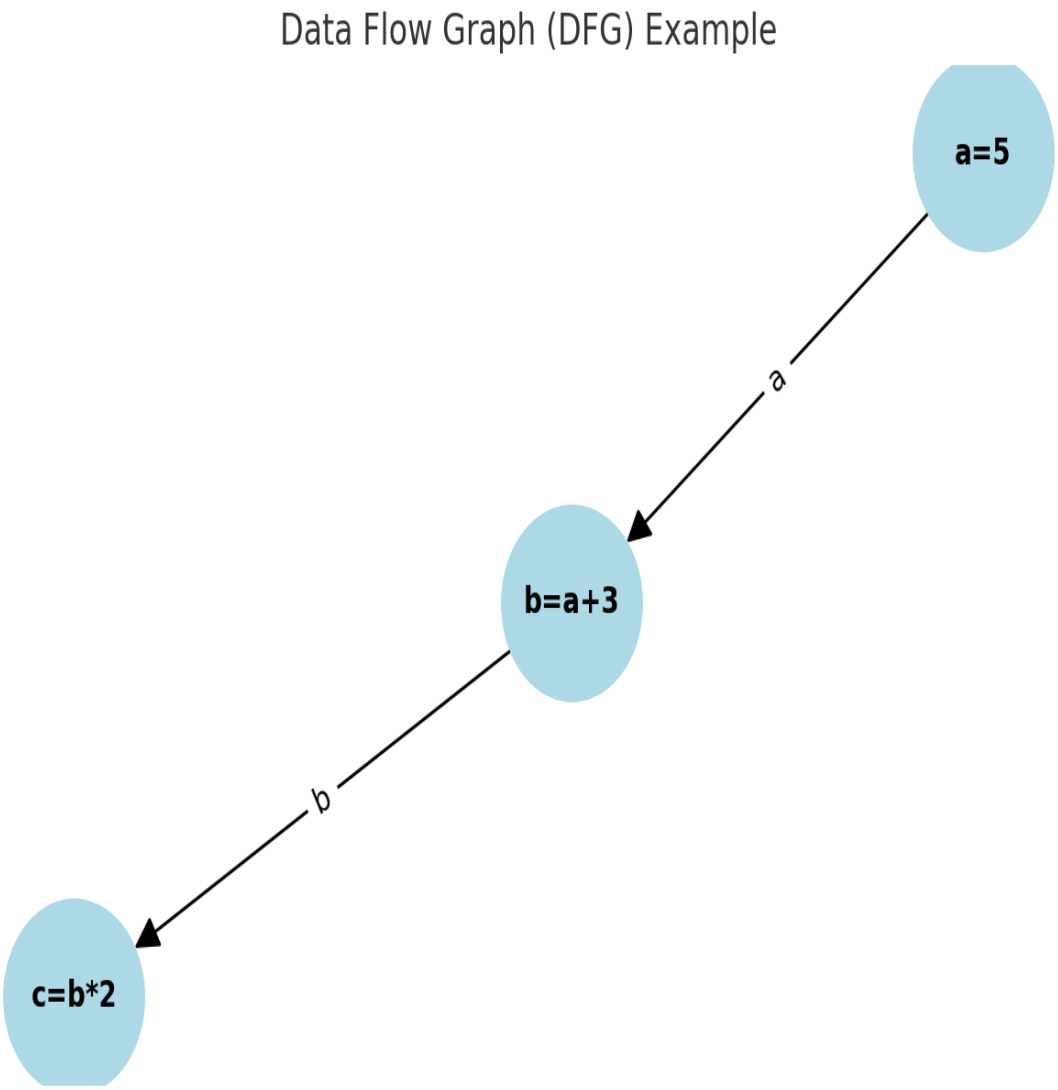


Figure 2.3: Data flow graph example

Figure2.3 illustrates an example of a **DFG**.The graph shows how data flows: a is defined and used in computing b. b is then used in computing c.
Some tools that generate **DFG** from Java Source Code: WALA framework[31], SOOT[28], Joern[16].

2.4 Static vs Dynamic Analysis

While static analysis examines code without execution, **dynamic analysis** monitors its behavior during runtime. Both approaches are complementary but serve different purposes. This is a simple Comparison table between both approaches:

Aspect	**Static Code Analysis**	**Dynamic Code Analysis**
Goal	Analyzes source code without executing it to identify maintainability issues, security flaws, and bad practices	Detects issues that appear during execution, such as memory leaks, logic errors, and performance bottlenecks
Output	Warnings and reports about potential flaws or risky patterns	The program's actual runtime behavior with logs, traces, and performance metrics
Performance Overhead	No runtime impact (code is not executed). Large codebases may slow down analysis time	Can impose a significant performance cost, since it runs within or alongside the application
Use Cases	Detecting coding standard violations, security flaws, and maintainability problems early in development	Finding issues that only manifest during real execution with real inputs and runtime conditions
Scope	All code paths (potentially)	Only executed paths
False Positives	More common	Less common

Table 2.1: Comparison of Static and Dynamic Code Analysis

2.5 Existing Static Analysis Tools for Java

There are several tools that support static code analysis for Java. These vary in complexity, coverage, and ease of integration:

2.5.1 Coverity

Coverity Static Analysis[6] is a commercial product made by Synopsys, works with multiple Programming languages such as: **C/C++, Java, C#**. **Coverity** offers a robust static code analysis, aiming to pinpoint software defects and security bugs. during the development phase. Recognized for its precision, it aligns with the commitment to reduce coding errors and improve overall software robustness.
Coverity Static Analysis[4] works in three steps to achieve software integrity:

1. Map the Software DNA
2. Indentify Critical Defects
3. Resolve Defects

Pros:

- Smooth integration with DevOps pipelines CI/CD, SCM, and issue-tracking integrations, allowing analysis and code improvement automation.
- Deep source code analysis helps reduce complex bugs and code vulnerabilities.

Cons:

- Some configurations can be intricate, requiring in-depth knowledge.
- As a commercial tool, the pricing might be on the higher end for some teams.

2.5.2 FindBugs

FindBugs[10] is a free static analysis tool designed to detect bugs in Java programs. It is distributed under the Lesser GNU Public License (LGPL) and is a trademark of the University of Maryland, which also owns the **FindBugs™** name and logo. As of July 2008, the tool had been downloaded over 700,000 times.

While **FindBugs** requires JRE or JDK version **1.5.0** or higher to run, it is capable of analyzing Java programs compiled with any version of the language.[4]

2.5.3 SonarQube

SonarQube[27] is an enterprise-grade, self-managed platform for systematic code quality and security analysis. Through its capabilities, developers can identify and fix code smells, coding errors, and other potential pitfalls, aligning with the objective of consistent code quality assurance. Trusted by 400,000+ organizations, SonarQube serves as an industry standard for maintaining clean code through automated static analysis and team-wide quality standards enforcement. Its flexibility supports both technical and governance requirements in enterprise environments.

Pros:

- **Multi-language support:** Analyzes 30+ languages, frameworks, and infrastructure-as-code **(IaC)**.
- **DevOps integration:** Works effortlessly with platforms like GitHub, GitLab, Bitbucket, Jenkins, Azure to perform code quality checks into the CD pipeline.
- **Quality governance:** Enforces standards via configurable Quality Gates (go/no-go thresholds) and Provides unified code health metrics across organizations.
- **Enterprise capabilities:** it allows for Customizable deployments and Centralized deployments.
- **Performance:** Delivers fast analysis with actionable results and it integrates with IDEs through the **SonarLint** extension for on-the-fly code issue detection.

Cons:

- Managing complexity may be difficult for smaller teams lacking specialized DevOps support.
- **SonarQube** may generate false positives, requiring careful analysis and validation.
- Initial configuration can demand a certain level of technical proficiency.

2.5.4 PMD

PMD[22] is a free and open-source rule-based static analyzer that helps enforce code style and detect detects recurring programming mistakes, code styles, and

performance violations such as: unused variables, empty catch blocks, unnecessary object creation. It comes with 400+ built-in rules and can be extended with custom rules. It uses JavaCC and Antlr to parse source files into **abstract syntax trees (AST)** and runs rules against them to find violations. **Pros:**

- CPD[2] prevents duplicate code occurrence.
- Efficient integration with popular development tools that support automated performance.
- **Extensibility:** Developers can write custom rules in **Java or XPath** to target project-specific issues.
- **IDE integration:** PMD integrates seamlessly with popular **Java IDEs** such as IntelliJ IDEA, Eclipse, NetBeans, and build tools like Maven, Gradle, and Ant.

Cons:

- May miss complex logic errors or context-specific security issues.
- May generate false positives in large or complex projects.
- Limited built-in support for modern security testing compared to specialized tools.

2.5.5 SpotBugs

SpotBugs[29] is a free and open-source static analysis tool designed to inspect Java bytecode for potential bugs and code quality issues. **SpotBugs** is a fork of **FindBugs** which is no longer maintained and can be used as a standalone application, as well as through several integrations including Maven, Gradle, Eclipse, and IntelliJ. **SpotBugs** needs at least Java 8 to run, but you can analyze programs written with older versions of Java.

[2]CPD stands for the copy-paste-detector, which is a duplicate code detection tool integrated into PMD

Pros:

- a wide-ranging set of bug patterns for detecting mistakes in Java code.
- a plugin architecture that allows users to extend its capabilities.
- seamless integrations with popular build tools with automated scans and report generation options.

Cons:

- Analyzes bytecode, not source – may miss certain syntax-based issues
- May not detect high-level design or architectural flaws
- Plugin setup (like FindSecBugs) may require manual configuration

Note: There are exits a lot of tools for Java Static Code Analysis like: **Snyk, Fortify, Checkmarks, Codiga.....**

2.6 Challenges and Limitations of Traditional SCA

While static analysis is powerful, it comes with certain limitations:

- **False Positives and False Negatives:** SCA tools are prone to generating both false positives and false negatives. A **false positive** occurs when harmless code is incorrectly flagged as vulnerable, leading to wasted developer effort. Conversely, **false negatives** represent undetected genuine vulnerabilities, posing significant security risks. Striking a balance between reducing false positives without increasing false negatives remains a persistent challenge.

- **Complexity of Static Analysis in Modern Software Systems:** Performing accurate static analysis is inherently difficult, as it requires a deep understanding of the application's structure, logic, and data flow—without executing the code. As software becomes more complex, the likelihood of missed vulnerabilities or incorrect findings increases. SAST tools also struggle to analyze dynamic or data-driven applications, such as modern web systems where user interactions and input paths vary significantly at runtime.

- **Language and Ecosystem Dependence:** There's not a single static analysis tool that does it all. Many are specialized for different environments, filetypes, and types of scanning. An organization might need one tool for security scanning, a different tool for their Typescript frontend, a third tool to scan the Golang backend, and yet another static analyzer for their server configuration Terraform files. Each tool delivers value, but it can be onerous to set up and maintain all of them.

- **Limits scope to an extent:** Static analysis can't detect runtime errors, performance issues, or concurrency problems as it doesn't execute the code. Combine static analysis with dynamic analysis and manual code reviews for comprehensive debugging.

- **Configuration Required:** Although default rule sets provide a useful starting point, they are often either too general or overly restrictive. To improve accuracy and relevance, it is advisable to customize configurations by disabling irrelevant rules and fine-tuning checks based on project-specific needs. This helps reduce noise from unnecessary warnings while minimizing the risk of overlooking real issues. Leveraging pre-defined rule sets as a baseline and adapting them accordingly can also reduce setup time and improve overall efficiency.

2.7 The Role of AI in Enhancing SCA

Artificial Intelligence (AI), particularly **machine learning (ML)** and **deep learning (DL)**, offers promising solutions to many challenges in traditional SCA. By learning from labeled datasets of vulnerable and safe code, AI models can detect patterns beyond the capabilities of hand-crafted rules. Here are some key advantages of incorporating AI into code analysis:

1. **Pattern Recognition:** AI-powered tools are highly effective at identifying complex patterns and trends in code, allowing them to detect subtle issues that might be missed by traditional analysis methods.

2. **Contextual Understanding:** AI is known to understand the context of the code, making it more better to reduce False Positives and provide a more accurate analysis.

3. **Self-Learning Capabilities:** Some AI-based tools are designed to learn continuously from the code they process, enabling them to adapt and improve their detection capabilities over time.

4. **Advanced Security Analysis:** AI can identify security vulnerabilities, such as zero-day threats and advanced malware, by learning from a vast database of historical threat data.

5. **Efficiency and Cost savings:** AI tools can efficiently scan large codebases, significantly reducing the time and effort required for manual code reviews and vulnerability assessments and also they save money by speeding up the security testing process.

6. **Code Quality and Refactoring:** AI tools play a vital role in improving overall code quality by enforcing consistent coding styles and suggesting intelligent refactoring options such as suitable design patterns or architectural enhancements.

7. **Make query creation more efficient:** AI-enabled SCA tools can easily generate queries based on simple prompts, even users with limited knowledge of query syntax can contribute to security testing using AI-enabled static analysis tools.

8. **Enhancing Developer Productivity:** AI can boost developers efficiency enabling faster identification of vulnerabilities and suggested fixes which will help devs to resolve issues faster and make them adapt **SCA** tools and solutions.

2.8 Conclusion

Static code analysis remains a corner stone of secure software development, offering proactive insights into code quality and potential vulnerabilities. In Java, where security risks are amplified by its widespread use and rich feature set, SCA provides a first line of defense. However, traditional approaches are limited in precision and adaptability. The integration of AI techniques presents a new frontier, enabling smarter, more context-aware analysis that enhances both detection accuracy and developer experience.

The next chapter explores the fundamentals and Foundations of **Machine Learning & Deep Learning**, their key concepts, learning paradigms and Some Known Algorithms/Architectures.

When applied to Java code, **AI-enhanced SCA** can adapt to evolving threats, detect previously unseen patterns, and operate at scale with improved accuracy. This fusion of traditional techniques and AI forms the foundation of the proposed solution, which will be detailed in the implementation phase of this thesis.

Chapter 3

Machine Learning & Deep Learning

3.1 Introduction

Machine learning (ML) and deep learning (DL) have become essential tools in modern **software engineering and Cyber Security**. Their ability to model complex patterns, learn from data, and adapt to evolving threats makes them particularly valuable in static code analysis and vulnerability detection. This chapter provides an overview of ML and DL, their key concepts, learning paradigms, and some of their techniques.

3.2 Machine Learning (ML)

Machine Learning (ML) is a branch of artificial intelligence (AI) that enables computers to learn and make predictions or decisions without being explicitly programmed for each task. Instead, ML uses data to "train" to understand patterns, make predictions, and improve performance over time on a task without being explicitly programmed. As illustrated in Figure 3.1, a typical machine learning workflow consists of several stages, from problem definition and data collection to deployment and monitoring:

1. **Problem Definition:** Clearly define the **objective** by understating the desired outcome and success criteria.

2. **Data Collection:** This phase involves **relevant** collection of datasets/datasources that can be used as raw data to train model. The quality and diversity of the collected data directly impact the robustness and generalization of the model.

3. **Data Cleaning and Preproccessing:** Prepare the raw collected data to be usable by machine learning models for better the accuracy and insuring the reliability of the machine learning model.

4. **Exploratory Data Analysis (EDA):** Analyze and visualize data to understand distributions, patterns, correlations, and outliers and this making choices in feature engineering, model selection and other critical aspects.

5. **Feature Engineering and Selection:** selecting the most relevant features to enhance model efficiency and effectiveness.

6. **Model Selection:** Choosing the appropriate machine learning algorithm based on the nature of the data, the complexity of the problem and the desired outcomes (e.g., Random Forest, SVM, Neural Networks).

7. **Model Training:** This is where the model **learns** from the input data

8. **Model Evaluation and Tuning:** involves testing the model on validation or test datasets using metrics like accuracy, precision, recall, and F1 score to assess its effectiveness.

9. **Model Deployment:** integrating the trained model into a production environment where it can start making real predictions for users or systems.

10. **Model Monitoring and Maintenance:** Update or retrain the model when necessary to handle data drift, new requirements, or degraded accuracy over time.

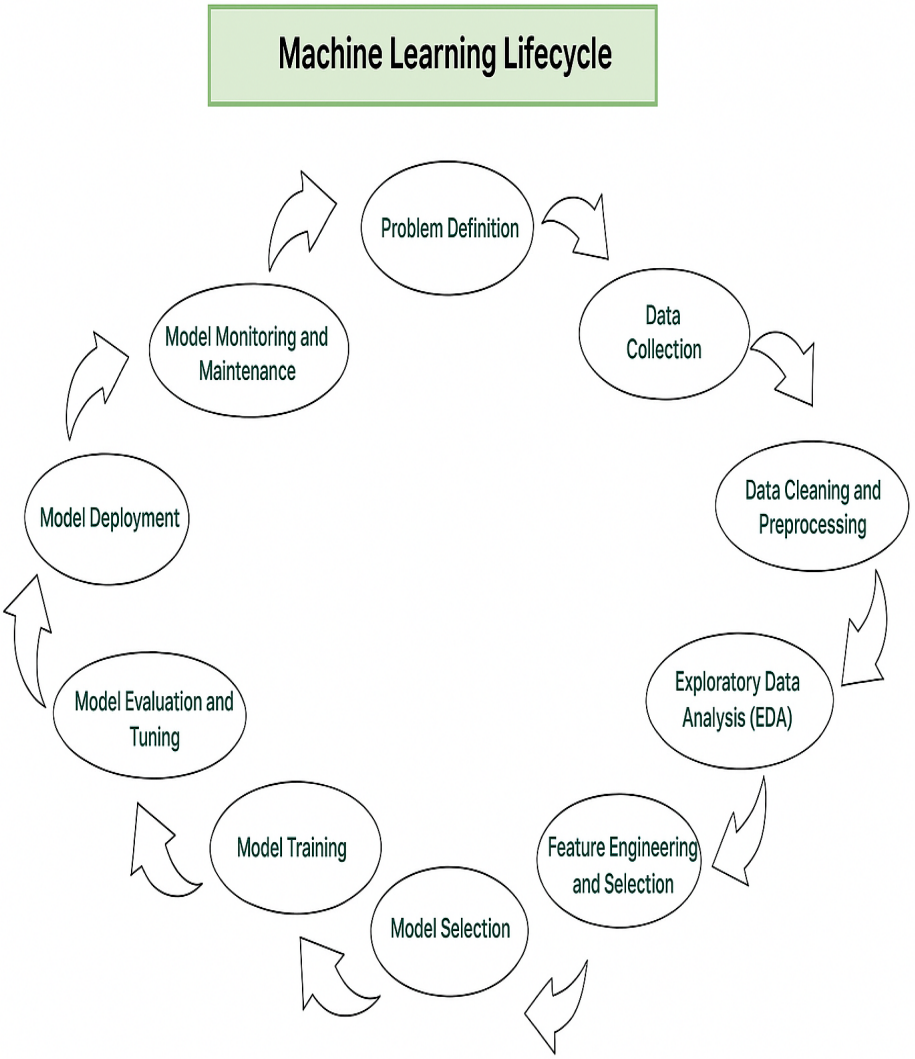


Figure 3.1: Machine Learning Lifecycle

ML can be categorized into four types:

- **Supervised Learning:** Trains models on labeled data to predict or classify new, unseen data.
- **Unsupervised Learning:** Finds patterns or groups in unlabeled data, like clustering or dimensionality reduction.
- **Semi-supervised Learning** uses a mix of labeled and unlabeled data, making it helpful when labeling data is costly or time-consuming.
- **Reinforcement Learning:** Learns through trial and error to maximize rewards, ideal for decision-making tasks.

3.2.1 Model Evaluation

Confusion Matrix

A confusion matrix is a simple table that shows how well a classification model is performing by comparing its predictions to the actual results. As shown in Figure 3.2, it provides insight into the strengths and weaknesses of the model and highlights where misclassifications occur.

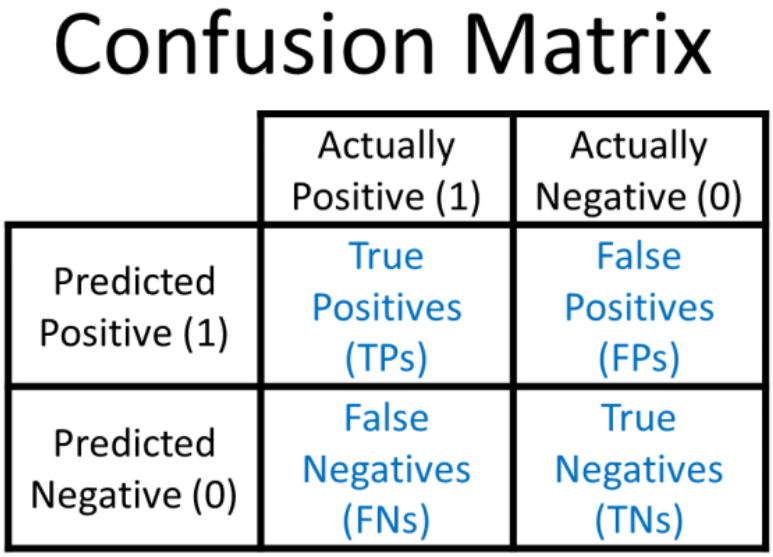


Figure 3.2: Confusion Matrix

True Positives (TPs): The number of positive examples that the model correctly classified as positive.
True Negatives (TNs): The number of negative examples that the model correctly classified as negative.
False Positives (FPs): the number of negative examples that the model incorrectly classified as positive (i.e. the negative examples that were falsely classified as "positive").
False Negatives (FNs): the number of positive examples that the model incorrectly classified as negative (i.e. the positive examples that were falsely classified as "negative").

3.2.2 Metrics based on Confusion Matrix Data

A confusion matrix: helps you see how well a model is working by showing correct and incorrect predictions. It also helps calculate key measures like **accuracy, precision, and recall,** which give a better idea of performance, especially when the data is imbalanced.[11]

Accuracy: measures the overall proportion of correct predictions made by the model, providing a general sense of its performance. However, accuracy can be misleading in cases of imbalanced datasets, where one class significantly outweighs others. In such scenarios, a model might achieve high accuracy simply by consistently predicting the majority class, while failing to correctly identify instances of minority classes, thus overlooking critical details.

$$\textbf{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision: also known as **positive predictive value PPV** measures the quality of the model's positive predictions. It indicates the proportion of instances predicted as positive that are actually positive. High precision is particularly important in scenarios where minimizing false positives is critical, such as in spam detection or fraud prevention.

$$\textbf{Precision:} = \frac{TP}{TP + FP}$$

Recall: also known as **Sensitivity, hit rate or true positive rate** evaluates how effectively the model identifies all actual positive cases. It represents the proportion of true positives captured out of all actual positives. High recall is crucial in applications where missing positive instances could have serious consequences, such as in medical diagnoses.

$$\textbf{Recall} = \frac{TP}{TP + FN}$$

F1-mesure: also knonw as **F1-score** combines precision and recall into a single metric to balance their trade-off. It provides a better sense of a model's overall performance, particularly for imbalanced datasets. The F1 score is helpful when both false positives and false negatives are important, though it assumes precision and recall are equally significant, which might not always align with the use case.

$$\textbf{F1-Score} = \frac{2 * Precision * Recall}{Precision + Recall}$$

Specificity: also known as **True Negative Rate TNR** is a key metric used to evaluate classification models, especially in binary classification tasks. It measures the model's ability to correctly identify negative instances, indicating how well the model avoids false positives. A higher specificity means the model is better at correctly rejecting negative cases.

$$\textbf{Specificity} = \frac{TN}{TN + FP}$$

AUC-ROC curve: used for evaluating the model's ability to differentiate between positive and negative classes across different thresholds. As illustrated in Figure 3.3, it plots the **True Positive Rate (TPR) against the False Positive Rate (FPR) at different thresholds** showing how well a model can distinguish between two classes such as positive and negative outcomes.
TPR (True Positive Rate): The ratio of correctly predicted positive instances.
FPR (False Positive Rate): The ratio of incorrectly predicted negative instances.

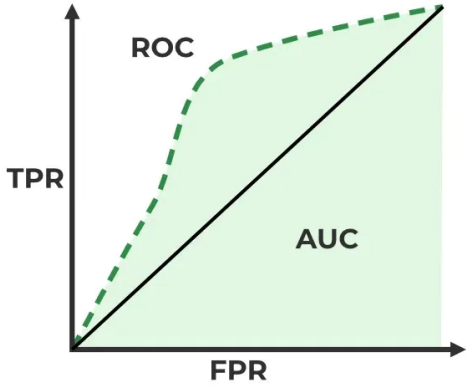


Figure 3.3: AUC-ROC

AUC-ROC near to 1: Excellent model performance.
AUC-ROC = 0.5: Performance similar to random guessing.
AUC-ROC < 0.5: Worse than random guessing.

AUC-PR curve: Area Under the Precision-Recall Curve evaluates model performance, especially for imbalanced datasets, using a plot of Precision versus Recall across thresholds, where one class significantly outnumbers the other. In such cases the ROC curve might show overly optimistic results as it doesn't account for class imbalance as effectively as the Precision-Recall curve. **AUC-PR near to 1:** Strong balance between precision and recall.
AUC-PR = 0.5: Similar to random guessing.
AUC-PR < 0.5: Performance worse than random.

3.2.3 Common ML Algorithms

There are many algorithms used in Machine learning each suited to different types of problems[12]. Some of the most commonly used ML algorithms are:

Decision Trees:

Decision Trees are supervised ML algorithm that is model decisions through a tree-like structure, where internal nodes represent feature tests, branches represent decision rules, and leaf nodes contain the final predictions. Decision Trees are widely used for classification and regression tasks and the process of creating Decision Trees involves:

1. **Selecting the Best Attribute:** starting at the root node with all data and using a metric like **Gini impurity, entropy, or information gain** to split data based on a chosen feature and a threshold.

2. **Splitting the Dataset:** The dataset is split into subsets based on the selected attribute.

3. **Repeating the Process:** The process is repeated recursively for each subset, creating a new internal node or leaf node until a stopping criterion is met (e.g., all instances in a node belong to the same class or a predefined depth is reached).

Key points:

Entropy: a concept from information theory, measures the uncertainty or disorder within a dataset. In decision trees, it is used to determine how to split data at each node, aiming to create more homogeneous subsets. **Entropy** values range from 0 (perfectly ordered) to maximum entropy (maximum disorder). At each node, the decision tree algorithm selects the feature and split that maximizes the reduction in entropy, a process called **Information Gain**.

Gini impurity: measures the degree of disorder or likelihood of misclassification within a dataset and is commonly used in classification tasks. A lower Gini impurity indicates a more homogeneous node. In decision trees, the algorithm selects features and split points that minimize Gini impurity at each node to create purer subsets.

$$\text{Entropy}(S) = -\sum_{i=1}^{c} p_i \log_2(p_i)$$

$$\text{Gini}(S) = 1 - \sum_{i=1}^{c} p_i^2$$

As illustrated in Figure 3.4, a decision tree splits data recursively based on feature tests, creating a tree-like structure that classifies inputs step by step until reaching a leaf node.

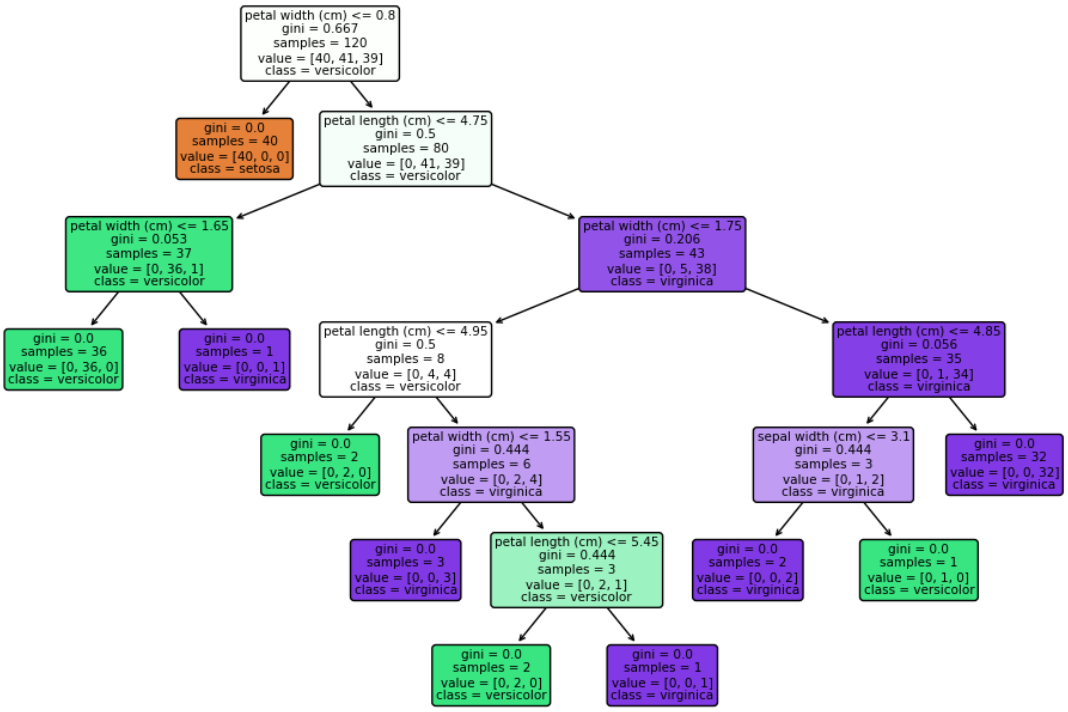


Figure 3.4: Example of Decision Tree on the Iris Dataset

Random Forest:

is an ensemble machine learning model that combines multiple decision trees to make predictions, it is widely used for classification and regression tasks. As illustrated in Figure 3.5, it takes different random parts of the dataset to train each tree and then it combines the results by averaging them and this approach helps improve the accuracy of predictions.

How Random Forests Works:

- It constructs multiple decision trees, each trained on a random subset of the original data, ensuring that each tree is unique.
- During the construction of each tree, the algorithm randomly selects a subset of features at each split, rather than considering all features, which increases diversity among the trees.
- Once trained, each decision tree makes a prediction:
  - For **classification tasks**, the final prediction is determined by a majority vote among the trees.
  - For **regression tasks, the final prediction is the average of the outputs from all trees.**

- The randomness in both data sampling and feature selection helps prevent overfitting, resulting in more accurate and reliable predictions.

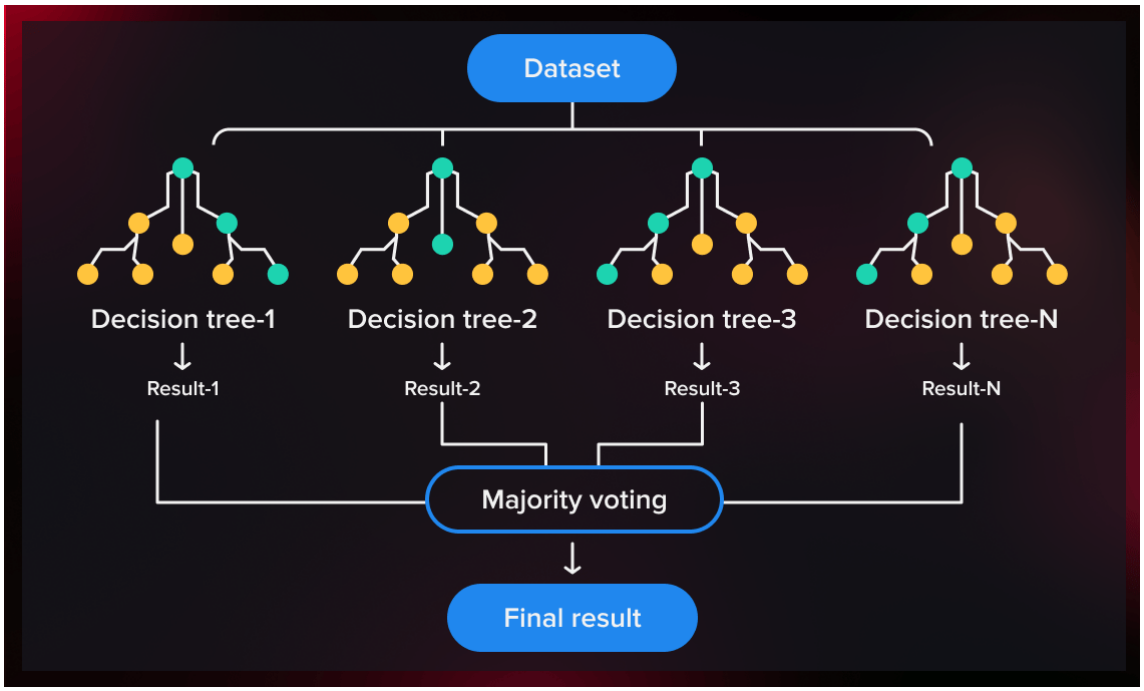


Figure 3.5: Random Forest Visualization

Support Vector Machines (SVM):

Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. SVM aims to find the optimal hyperplane in an N-dimensional space to separate data points into different classes, maximizing the margin between the closest points of different classes called **support vectors** and **the hyperplane**.

SVM is effective in high-dimensional spaces and can handle complex, non-linear patterns using **kernel functions**, which map data into higher dimensions to make them linearly separable. The model optimizes this hyperplane using quadratic programming, and predictions are made based on the side of the hyperplane a data point falls on.

SVM is memory-efficient, using only a subset of the training data (support vectors) in decision making. However, selecting an appropriate kernel function is crucial to prevent overfitting, especially when the number of features exceeds the number of training samples.

Naïve Bayes (NB):

NB classifiers are supervised machine learning algorithms used for classification tasks, based on Bayes Theorem to classify data based on the probabilities of different classes given the features of the data. It assumes that all input features are conditionally independent, an assumption rarely true in practice but effective nonetheless. The model calculates the posterior probability of each class using the prior probability of the class and the conditional probability of each feature given the class.

Despite its simplicity, Naïve Bayes performs surprisingly well in many real-world applications, particularly when using the **Maximum a Posteriori (MAP)** rule for prediction. It supports both categorical and continuous features, and reduces high-dimensional problems into manageable one-dimensional probability estimates.

Bayes' Theorem finds the probability of an event occurring given the probability of another event that has already occurred. Bayes' theorem is stated mathematically as:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)}$$

where:

- y and X are events with $P(X) \neq 0$.
- $X = (x_1, x_2, x_3, \ldots, x_n)$ (feature vector).
- **Posterior Probability** ($P(y|X)$): Probability of hypothesis y given observed evidence X.
- **Likelihood** ($P(X|y)$): Probability of observing X if y is true.
- **Prior Probability** ($P(y)$): Initial probability of y before observing X.
- **Marginal Probability** ($P(X)$): Total probability of observing X under all hypotheses.

Types of Naïve Bayes Models:

- **Gaussian Naïve Bayes:** Assumes that continuous features follow a normal (Gaussian) distribution. It is suitable for data where features are real-valued and normally distributed.
- **Multinomial Naïve Bayes:** Used for discrete feature counts, such as word frequencies in documents. Commonly applied in text classification tasks like spam detection or topic labeling.

- **Bernoulli Naïve Bayes:** Works with binary features, indicating the presence or absence of terms. It's ideal for document classification where term occurrence matters more than frequency.

K-Nearest Neighbors (KNN):

KNN is a supervised machine learning algorithm used for classification and regression tasks that predicts the class of a new data point based on the majority class of its **K nearest neighbors**. The algorithm calculates the **"distance"** between the new point and all other points in the dataset. The most common way to measure this distance is by using the **Euclidean distance and Manhattan Distance**. Once the distances are calculated, the algorithm takes a majority vote from the K-nearest points and classifies the new point accordingly, as shown in Figure 3.6

While **KNN** is simple and effective, it faces two main challenges, **High Computational Cost** where distance calculations become slow with large datasets and **Curse of Dimensionality** where performance degrades with too many features, as irrelevant ones distort distance metrics. Solutions include using feature selection to reduce noise and applying efficient search structures like **KDTree or BallTree** to speed up neighbor searches.

$$\text{Manhattan Distance} = \sum_{i=1}^{n} |X_1^i - X_2^i|$$

$$\text{Euclidean Distance} = \sqrt{\sum_{i=1}^{n} (X_1^i - X_2^i)^2}$$

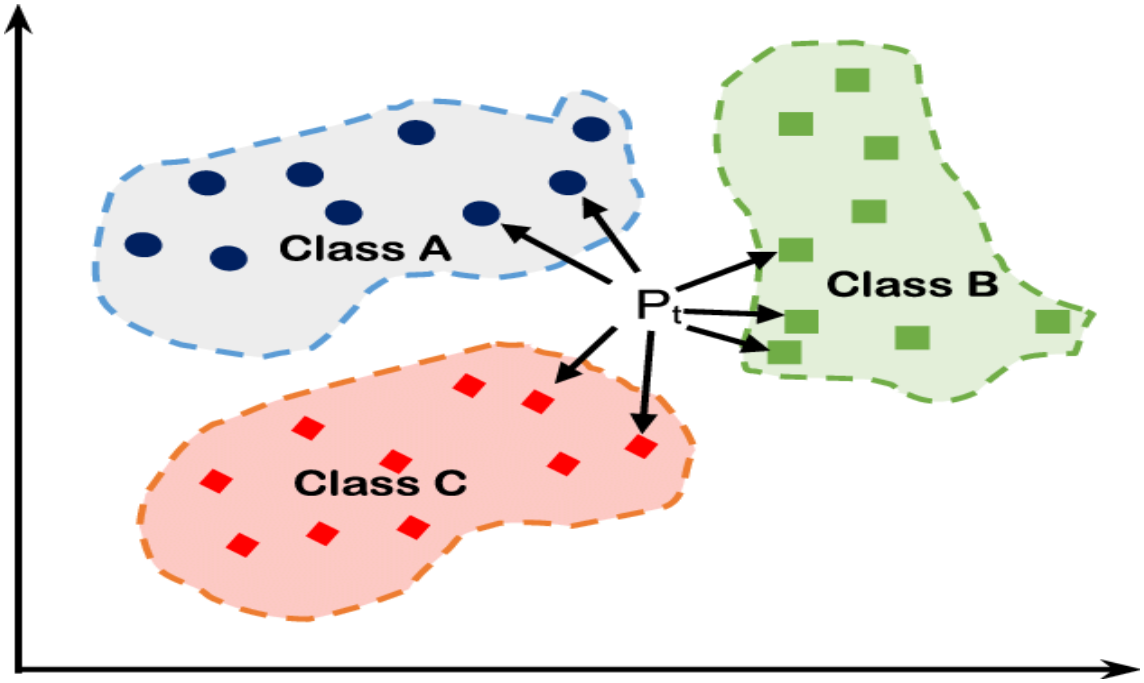


Figure 3.6: KNN Algorithm illustration

Logistic regression (LR):

Logistic regression **also known as the logit model)** is a supervised statistical machine learning algorithm used for classification tasks where the goal is to predict the probability that an instance belongs to a given class or not. It applies a logit transformation to convert probabilities into log odds, enabling prediction within a range of 0 to 1. **Logistic Regression** is applicable to binary, multinomial, or ordinal target variables and assumes independent observations and a linear relationship between the independent variables and the log odds.

Logistic regression applies a linear function to the input features and uses the sigmoid function, which maps any real-valued input to a probability between 0 and 1, forming an S-shaped curve. This probability is then used for binary classification by applying a decision threshold. While powerful, logistic regression typically requires a large sample size for reliable and accurate results.

3.3 Deep Learning (DL)

Deep learning is a sub-field of machine learning that involves neural networks with multiple layers (deep neural networks). It is particularly effective in modeling high-dimensional data and complex patterns, such as those found in natural language, images, and source code.

In this Section we will explore some Known **DL** Architectures:

3.3.1 Neural Networks

A neural network is a computational model inspired by the real neurons of the human brain, consisting of interconnected units called neurons. Each neuron processes inputs using **weighted sums and activation functions**, passing outputs through layers until a final result is produced, see Figure 3.7.

Neural networks are effective in tasks with imprecise input and no clear rules, such as image and speech recognition, forecasting, and medical diagnostics. When a neural network contains multiple hidden layers, it is referred to as a **Deep Neural Network (DNN) or Deep Learning model**.

Key Components of Neural Network:

- **Neurons:** Fundamental units that process input signals. Each neuron applies a threshold and an activation function to determine its output.
- **Connections:** Links between neurons that transmit information. These are influenced by weights and biases, which control the strength and direction of the signal.
- **Weights and Biases:** Core parameters that the network adjusts during training. Weights determine the importance of inputs, while biases shift the activation threshold.
- **Propagation Function:** Governs how inputs move and are transformed through the network, layer by layer.
- **Learning Rule:** The algorithm (e.g., **backpropagation**) that updates weights and biases to reduce prediction error and improve model performance over time.

Neural Networks: learn through:

- **Supervised Learning:** The network learns from labeled input-output pairs by minimizing the error between predicted and actual outputs through iterative adjustments.
- **Unsupervised Learning:** The network explores unlabeled data to discover hidden patterns or structures, using techniques like clustering and association.

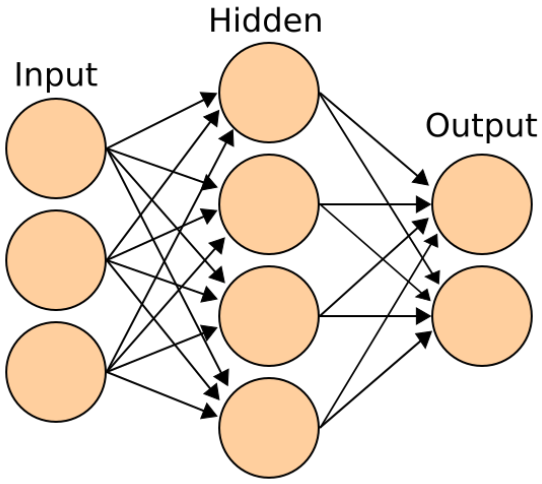


Figure 3.7: A Neural Network with one Hidden layer

- **Reinforcement Learning:** The network learns through trial and error by interacting with an environment, receiving rewards or penalties to optimize long-term outcomes.

3.3.2 Types of Neural Networks

Neural networks (NNs) are designed for different tasks, from image recognition to sequence prediction. Here iss an overview of some known types and architectures of neural networks:

Feedforward Neural Network (FFN):

FFN is a type of neural network where data flows in one way, input layer, through one or more hidden layers which learn complex data patterns using weighted sums and **activation functions (e.g., Sigmoid, Tanh, ReLU)**, to the output layer, without cycles or feedback. It is widely used for pattern recognition tasks like image, speech, and credit scoring.

Convolutional Neural Network (CNN):

Convolutional Neural Networks (CNNs) are a type of neural networks designed to learn spatial hierarchies of features through local connections, shared weights, and multiple layers of convolution, activation, and pooling operations. This is particularly useful for visual datasets such as images or videos, where data patterns play a crucial role.As illustrated in Figure 3.8, a typical CNN architecture consists of several key layers, each playing a distinct role in feature extraction and representation:

- **Input Later:** Receives raw pixel data (e.g., image matrices).
- **Convolutional Layers:** Apply learnable filters **(kernels)** that slide over the input to detect local patterns such as edges, textures, and shapes. The result is a feature map representing detected features.
- **Activation Functions:** Non-linear transformations (e.g., ReLU) are applied after convolution to introduce non-linearity, allowing the network to learn more complex patterns.
- **Pooling Layers:** Pooling Layers: Reduce the spatial dimensions of the feature maps through operations such as max pooling or average pooling, helping to decrease computational complexity and control overfitting.
- **Fully Connected Layers:** Towards the end of the network, fully connected layers (similar to those in traditional neural networks) are used for final classification or regression tasks.
- **Output Layer:** Produces the final prediction, often with a softmax activation for multi-class classification problems.

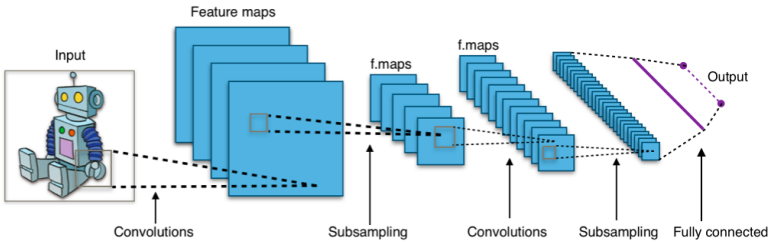


Figure 3.8: A Simple CNN architecture Example. Wikimedia Commons, 2015[5]

Training a Convolutional Neural Network (CNN) follows the same fundamental steps as other neural networks. First, **forward propagation** is used to compute the network's output based on the input data. Then, a **loss function—such as cross-entropy**—is applied to quantify the error between the predicted and actual outputs. This error is then propagated backward through the network during **backpropagation**, allowing the model to compute gradients with respect to its weights. These gradients are used in **gradient descent optimization** (or its variants like Adam) to iteratively update the weights and minimize the loss. To enhance the model's generalization capabilities and reduce **overfitting, regularization techniques such as dropout, data augmentation, and batch normalization** are often employed during training.

Recurrent Neural Networks (RNN):

Recurrent Neural Networks (RNNs) are a class of neural networks designed for processing sequential data. Unlike feedforward neural networks, RNNs have loops in their architecture, allowing them to maintain a memory of previous inputs by passing information from one step of the sequence to the next. This internal state makes RNNs particularly well-suited for tasks where the order or context of data is crucial, such as time series forecasting, natural language processing (NLP), and speech recognition.

As illustrated in Figure 3.9, the core idea behind RNNs is to **reuse the same network structure** across each time step while maintaining a **hidden state** that is updated at each step based on the current input and the previous state. Mathematically, this is represented as:

$$h_t = f(W_h h_{t-1} + W_x x_t + b) \tag{3.1}$$

Where:

- h_t is the hidden state at time step t
- x_t is the input at time step t
- W_h and W_x are weight matrices
- b is the bias
- f is an activation function, typically `tanh` or `ReLU`

This recurrent structure allows RNNs to exhibit **temporal dynamic behavior**, making them ideal for processing sequences of arbitrary length.

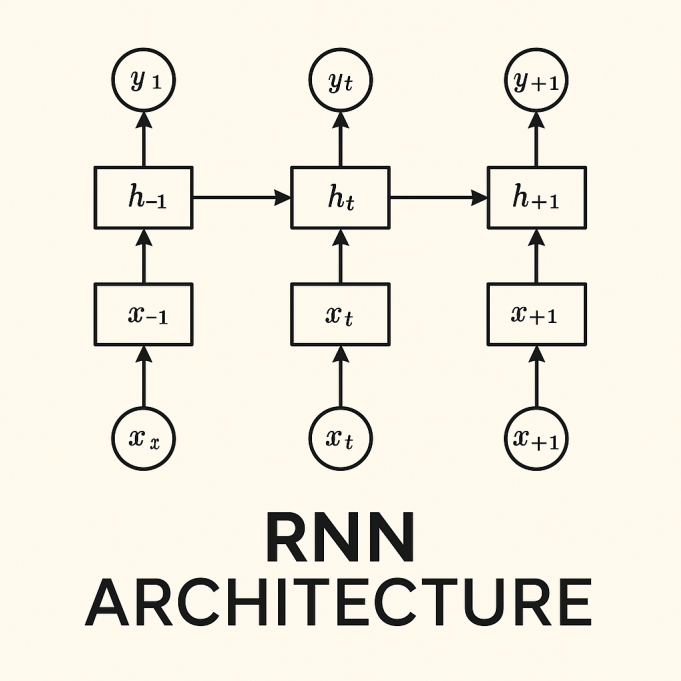


Figure 3.9: RNN illustration

Training RNNs typically involves **Backpropagation Through Time (BPTT)**, a variant of backpropagation adapted for sequential data. This process unrolls the network across time steps and calculates gradients for each step, which are then aggregated to update the weights. **Optimization algorithms such as SGD, Adam, or RMSprop** are commonly used, along with regularization techniques like **dropout, gradient clipping, or layer normalization** to improve generalization and training stability.

Long Short Term Memory (LSTM):

Long Short-Term Memory (LSTM) is an enhanced version designed to overcome the limitations of traditional **RNNs**. **LSTMs** can capture long-term dependencies in sequential data making them ideal for tasks like language translation, speech recognition and time series forecasting. While standard **RNNs** can theoretically retain information over long sequences, in practice they struggle to preserve and propagate relevant information over extended time steps due to gradient degradation. **LSTMs** address this issue by introducing a more complex memory unit with gating mechanisms that regulate the flow of information, allowing the network to retain useful information for longer periods and discard irrelevant inputs.

An **LSTM unit** consists of a cell state and three types of gates that control how information flows through the unit:

- **Forget Gate** (f_t): Decides which parts of the cell state to discard.
- **Input Gate** (i_t): Determines which values from the input should be used to update the cell state.
- **Output Gate** (o_t): Controls how much of the cell state is exposed as output.

Each gate is a neural network layer that uses a sigmoid activation function to output values between 0 and 1, representing how much information should pass through. The cell state acts like a **conveyor belt** running through the chain of LSTM units, carrying relevant information with minimal modification, while the gates make selective updates. This design allows LSTMs to learn long-range dependencies effectively.

Mathematical Formulation:

At time step t, given input x_t, previous hidden state h_{t-1}, and previous cell state c_{t-1}, the LSTM performs the following operations:

$$
\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
\tilde{c}_t &= \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \\
c_t &= f_t * c_{t-1} + i_t * \tilde{c}_t \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
h_t &= o_t * \tanh(c_t)
\end{aligned}
$$

Where:

- σ is the sigmoid activation function.
- $*$ denotes element-wise multiplication.
- W and b are learnable weights and biases.

Training LSTMs: LSTM networks are typically trained using **Backpropagation Through Time (BPTT)**, with optimization algorithms such as **Adam or RMSProp**. Regularization methods such as **dropout and gradient clipping** are often applied to improve generalization and training stability.

Graph Neural Networks (GNN)

Graph Neural Network (GNN) is a type of neural network designed to work directly with graph-structured data. GNNs can model complex, non-Euclidean relationships in data, such as social networks, molecular structures, and knowledge graphs. Graphs are data structures made of: Nodes (vertices) which represent entities and **Edges** represent relationships. Unlike images (grids) or text (sequences), graphs are irregular structures where each node can have a variable number of neighbors. Unlike images (grids) or text (sequences), graphs are irregular structures where each node can have a variable number of neighbors.
Graphs can be represented in several forms, depending on the application:

- **Adjacency Matrix:** A square matrix where each entry indicates whether an edge exists between two nodes. This format is well-suited for dense graphs.
- **Features Matrices:** Store attributes of nodes and edges (e.g., age, weight, or connection strength), providing additional contextual information about the graph.
- **Adjacency List:** Each node maintains a list of its neighbors, making this representation memory-efficient for sparse graphs.
- **Edge List:** Represents the graph as a collection of node pairs, making it convenient for algorithms that operate directly on edges.

As illustrated in Figure 3.10, **GNN** operates through message passing, where each node begins with an initial feature vector (such as a variable type or token embedding). At every layer, a node gathers information from its neighbors using operations like averaging, summation, or attention-based weighting and then updates its own representation by combining the aggregated message with its current state. Repeating this process over multiple layers allows nodes to develop context-aware embeddings that capture both their individual properties and the structure of their surrounding neighborhood. Finally, a readout function aggregates these node embeddings into a single graph-level representation, enabling tasks such as vulnerability detection.

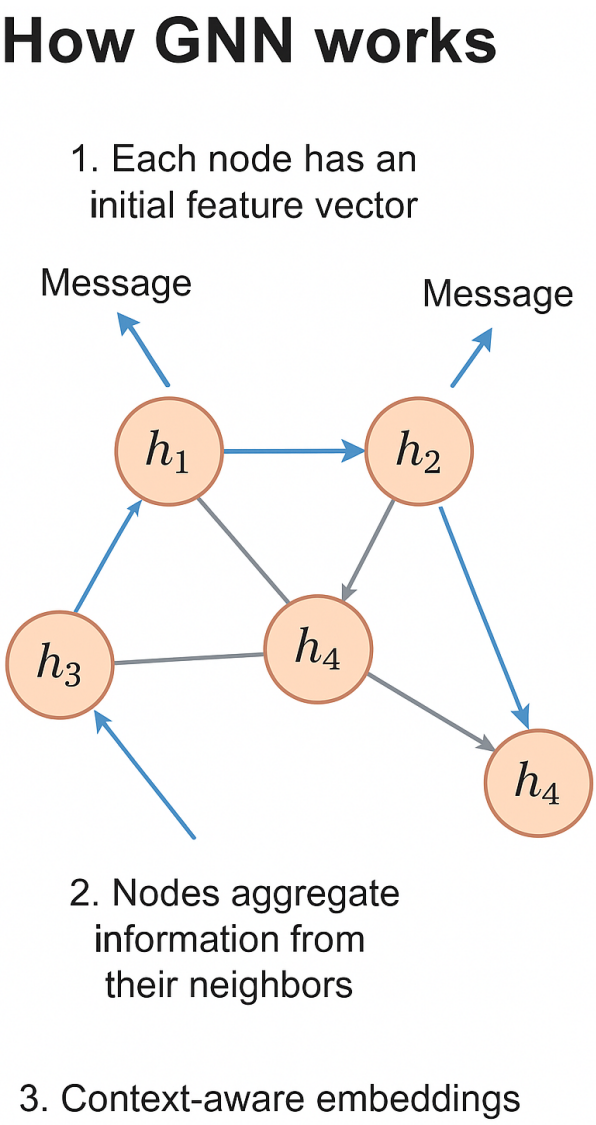


Figure 3.10: A simple illustrations of gnn

Transformers

Transformers are a deep learning architecture that completely changed how we handle sequential data like text, code, or even images. They were introduced in the paper "Attention is All You Need" (2017) and quickly replaced RNNs and LSTMs in many tasks because transformers overcome the problems in the previous architectures: vanishing gradients, Bad at capturing long-range dependencies problems.

The Transformer architecture begins by converting input tokens into embeddings and enriching them with positional encodings to preserve sequence order. These representations pass through stacked encoder layers, each containing multi-head self-attention mechanisms that let tokens attend to one another, followed by position-wise feed-forward networks, with residual connections and layer normalization ensuring stability. In sequence-to-sequence tasks, decoder layers are added, featuring masked self-attention (to prevent looking ahead) and encoder-decoder attention (to align inputs with outputs), along with their own feed-forward networks and normalization. Finally, the model projects the processed representations through a linear layer and softmax to produce predictions, such as the next token in text generation or labels for classification.
Core Concepts of Transformers[13]:

- **Self-Attention Mechanism:** Each token is mapped to Query (Q), Key (K), and Value (V) vectors; scaled dot-product attention computes how much focus each token should give to others.
- **Positional Encoding:** Adds sequence order information to embeddings since Transformers process tokens in parallel.
- **Adjacency List:** Each node maintains a list of its neighbors, making this representation memory-efficient for sparse graphs.
- **Multi-Head Attention:** Runs multiple self-attention mechanisms in parallel, allowing the model to capture diverse relationships.
- **Feed-Forward Networks:** Two linear layers with ReLU applied independently at each position to refine representations.
- **Encoder-Decoder Architecture:**
  - **Encoder:** Stacked layers of self-attention + feed-forward networks produce contextualized representations.
  - **Decoder:** Similar layers plus encoder-decoder attention to align input and output during generation (e.g., translation).

And there is also a wide variety of deep learning architectures exist, designed to address different challenges in fields such as: **Generative Adversarial Networks (GANs), Autoencoders** and different variations of architecture based on the existing architectures like **LSTM with RNN**.

3.4 Applications in Code Analysis

The application of **ML and DL** to software engineering tasks is growing rapidly. In the context of static code analysis, ML/DL can be used to:

- **Classify code snippets** as vulnerable or safe.
- **Detect code smells or refactoring opportunities**
- **Perform taint analysis:** using learned representations.
- **Predict potential bugs or security flaws**

3.5 Conclusion

Machine learning and deep learning offer **transformative** potential in the domain of code analysis and software security. Their ability to learn from data and uncover complex patterns complements traditional static analysis methods. In the next chapter, we will review state-of-the-art approaches that apply these techniques to Java vulnerability detection, highlighting current tools, benchmarks, and research trends.

Part II

Contribution

Chapter 4

JFClassifier

4.1 Introduction

This chapter presents the core contributions of this work, focusing on the design and implementation of a deep learning architecture for automated vulnerability detection in Java source code. While previous chapters outlined the background, here we translate those insights into a concrete solution. The proposed architecture leverages multiple program representations—Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Dependence Graphs (DDGs)—to capture both syntactic and semantic information from source code. These representations are then integrated using advanced neural mechanisms, such as multi-head attention and transformer-based encoders, enabling the model to focus on different aspects of the code simultaneously and capture long-range dependencies effectively.

The chapter is structured as follows: first, we describe how we gathered and constructed the datasets used for training and testing. Second, we describe the overall system design and its architecture components; third, we detail the preprocessing pipeline for transforming Java functions into their respective representations; and finally, we discuss the integration of the trained model into practical tools, such as a web application and a Visual Studio Code extension, to support developers in real-world settings.

4.2 Dataset Construction

After completing the state-of-the-art review for our master's thesis, we chose to combine datasets from multiple open-source projects in order to build a more diverse and representative benchmark. Specifically, we integrated the Juliet Test Suite from SARD[24], the OWASP Benchmark Project[1], and code snippets from "Finetuning LLM for Vulnerability Detection" project[25]. We managed to gather 69,850 java function which of whom 49,846 are safe and 20,004 are vulnerable. Each of the following sections presents these datasets in detail and explains how they were processed and utilized in our work:

4.2.1 Juliet Test Suite

Juliet test suite[19] is a collection of test cases in the Java language from the SARD Project. It contains examples organized under 112 different CWEs and currently in Version 1.3.

SARD

Software Assurance Reference Dataset (SARD)[24] created by NIST[1] (National Institute of Standards and Technology), it's a benchmark dataset widely used for evaluating static analysis tools, ML/DL models, and vulnerability detectors in source code. It covers multiple programming languages such as: Java, php, C and C++ with over 150 CWE classes and 450,000 test cases for the time being.
Basically, SARD is a collection of test cases where each **test-case** is a small, synthetic program or code snippet designed to represent either a true vulnerability instance (e.g., SQL injection, buffer overflow, hardcoded password, etc.) or non-vulnerable (safe) version of the same code.

Usage in Our Dataset

As we mentioned earlier, we used the Juliet test suite in our Dataset construction by following its user guide. Based on the juliet's user guide, the juliet test suite contains 4 types of test case-design: Non-class-Based Flaw, Class-based Flaw, Abstract Method test case, Bad-only Test case.
For our model design, we chosed only Non-class and Bad only test cases then following the user guide, we created a python scripts to extract the safe/vulnerable

[1]https://www.nist.gov/

functions with this structure/instructions:

1. extract the safe/vuln method with its constructor.
2. remove the comments.
3. convert code to follow the knr representation.
4. rename the function name with this format: **func_incrementalCounter**
5. save the code in a java file inside a folder of java files with a public class wrapping up the targeted function with this format: **Sample_incrementalCounter**
6. save the preprocessed java function into a **jsonl** file with its target class and CWE_id for future/easier usage.

this is an **Example** of a sample after extraction and preprocessing.

```
public class Sample_20 {
    public void func_20() throws Throwable {
    if (IO.STATIC_FINAL_FIVE == 5) {
        FileOutputStream streamFileOutput = null;
        try {
            String path = "C:\\test_bad.txt";
            File file = new File(path);
            long lastModified = file.lastModified();
            streamFileOutput = new FileOutputStream(file);
            streamFileOutput.write("This is a new line".getBytes("UTF-8"));


            file.setLastModified(lastModified - 10000L);
        }
        catch (IOException exceptIO) {
            IO.logger.log(Level.WARNING, "File I/O error", exceptIO);
        }
        finally {
            try {
                if (streamFileOutput != null) {
                    streamFileOutput.close();
                }
            }
```

```
                catch (IOException exceptIO) {
            IO.logger.log(Level.WARNING, "Error closing FileOutputStream", exceptIO);
                }
            }
        }
    }
}
```

this is an **Example** of an entry in **dataset.jsonl** file.


```
{
    "function": "public class Sample_646 {\n\tpublic void func_646() throws Throwable {\n  int data;\n  if (true) {\n  \n  data = 2;\n }\n   else {\n   \n\n
    data = 0;\n  }\n\n   if (true) {\n    \n   int array[] = { 0, 1, 2, 3, 4 }; IO.writeLine(array[data]);\n       }\n    }\n\t}",
    "target": 0,
    "cwe": "CWE129"
}
```


4.2.2 OWASP Benchmark Java

The OWASP Benchmark Project[1] is a Java-based test suite created to assess the accuracy, coverage, and performance of automated software vulnerability detection tools. Without such a benchmark, it is challenging to evaluate these tools objectively, understand their strengths and weaknesses, or compare their effectiveness.

The Benchmark is a fully runnable open-source web application containing thousands of intentionally vulnerable test cases, each mapped to a specific CWE. For the moment, the project contains nearly 3,000 test cases(version 1.2) and all vulnerabilities included in the Benchmark are deliberately designed to be exploitable, ensuring that the evaluation is fair across various testing approaches.

Usage in our Dataset

Follwing the same approach similar to **Juliet-test-suite** dataset mentioned in **the last paragraph of Section 4.2.1 on page 65".** , we created a python script with the help of the project documentation in order to extract more code snippets for our dataset. Also, the resulted entries of the script will be similar to the previous **example**.

4.2.3 Finetuning LLM for Vulnerability detection

During the state-of-the-art review, we came across a paper entitled [25] "Finetuning Large Language Models for Vulnerability Detection" and the authors made the necessary code for reproducing their experiments conducted in their paper publicly

availble[23]. Their Dataset structure is similar to our needs and it was easier to proccess it using python scripting but we asserted to get the same results as with juliet and owasp test-suites.

4.3 Model Architecture and Design

Our model JFClassifier is a replicate of the work done in the paper entitled[32] "JFinder: A novel architecture for java vulnerability identification based quad self-attention and pre-training mechanism" but with a single twist "replacing DFG with DDG". As illustated in 4.1, JFinder architecture leverages multiple code representations to enhance vulnerability detection in Java source code including Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs) and Code snippet sequences (CSS) then it employs advanced deep learning concepts such as multi-head attention and transformers in order to proccess this Features and fuse them into a single matrix. Finally, the fused matrix will pass through a CNN (Convolutional Neural Network) and a Then through a Fully Connected Neural Network (MLP) for the final classification.

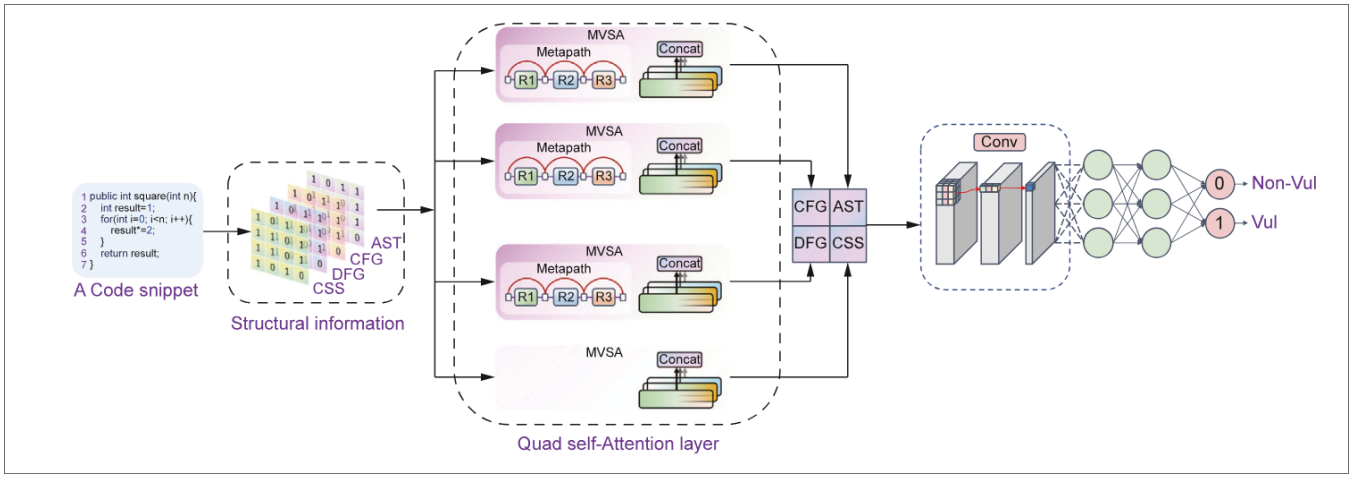


Figure 4.1: Illustration of the JFinder model.

As we said earlier, our model follows the same architecture of jfinder but instead of using DFG graphs, we used the DDG graphs. We chosed to replicate the JFinder architecture because of several reasons: it combines structural and semantic information from code which enhances the model's ability to capture data, it employs Multi-head self attention which no other research paper used to our knowledge and finally the jfinder model resulted in better metrics comparing to other approaches. Now we will explain each component of our architecture and explain how we managed to create or implemented this components in the next sections:

4.3.1 AST

An Abstract Syntax Tree (AST) is a hierarchical representation of the syntactic structure of source code. Each node denotes a code construct, while edges represent parent–child relationships between constructs. Sentences are linked to their tokens, forming a concatenation graph. In our model, the AST is used to capture the syntactic hierarchy of the code.

4.3.2 CFG

A Control Flow Graph (CFG) is a graph that represents all possible execution paths of a program. Its nodes correspond to control structures such as if, for, while, and switch, while directed edges represent the flow of execution between them. The CFG has a unique entry and exit point, illustrating how the program progresses through different branches depending on conditions.

4.3.3 DDG

A Data Dependence graph (DDG) is a graph-based representation that models how variables are defined, modified, and consumed in code. DDG helps capture semantic relationships in data usage, complementing AST (syntax) and CFG (execution flow).

4.3.4 CSS

Code Snippet Sequence (CSS) encoding the semantic information of source code into feature matrices using a pre-trained transformer model for code relatec tasks called UniXcoder[14].

4.3.5 Quad self-attention Layer

After extracting the structural representations (AST, CFG, and DDG) and the semantic representation (CSS), these views must be integrated to capture location-specific features and enhance the representation of code. To achieve this, we employ a **Multi-View Self-Attention (MVSA) encoder**, an extension of the multi-head attention mechanism.

The encoder takes as input the matrices Q, K, and V, representing the query, key, and value, respectively. In self-attention, these matrices are identical. Attention weights are computed by taking the dot product of the query with all keys, scaling by $\sqrt{d_k}$, and applying the softmax function. The output of a single attention head is defined as:

$$h_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

where Q_i, K_i, and V_i are the i-th submatrices of Q, K, and V, and d_k is the dimension of the key vectors.

Outputs from multiple heads are concatenated and projected with a learnable weight matrix W^o, producing the fused representation G:

$$G = \text{Concat}(h_1, h_2, \ldots, h_n)W^o$$

where n denotes the number of attention heads.

Finally, the model stacks four MVSAs into a **quadruple self-attention layer**, combining structural and semantic views as follows:

$$Q = \text{Layer}(\text{AST}, \text{DDG}, \text{CFG}, \text{CSS})$$

Here, Q represents the unified matrix containing both structural and semantic information from the code.

4.3.6 MetaPath Module

A **MetaPath** is a sequence of relations connecting different node types, forming a composite relation between the start and end nodes. Since enumerating all possible MetaPaths is computationally expensive and can cause data sparsity, We focuses on length-2 MetaPaths by adding reverse edges between connected nodes. For program graphs like AST, CFG, and DDG, which are mostly tree-like, this reflective connection improves graph completeness, strengthens connectivity, and helps reduce overfitting. The MetaPath will be applied when creating the adjancy matrices and we will showcase its module in the implementation section.

4.4 Architecture Implementation

4.4.1 Generating AST

For generating AST, we used javaParser[15] an open-source library which provides you with an Abstract Syntax Tree of your Java code. Using the javaParser modules, we generated AST for each sample input file in the form a DOT file which contains edges and nodes information, then we converted the DOT file to a DOT TXT file which represents only the edges between nodes. Finally, we create the AST adjacency matrix using python scripting for all the dataset and save all the matrices in a form of numpy file.
this is the java functions used to generate AST DOT and DOT TXT files

```
static void saveToDot(String filePath, String new_file_path) throws Exception {
        // FILE_NAME refers to the location of the sample in the Dot Directroy
        // retrurn a list of missed files to fix later on but continue in case no mistake
            try {
        CompilationUnit cu = StaticJavaParser.parse(new File(filePath));

        DotPrinter printer = new DotPrinter(true);
        try ( FileWriter fileWriter = new FileWriter(new_file_path + ".dot");
              PrintWriter printWriter = new PrintWriter(fileWriter)) {
            printWriter.print(printer.output(cu));
        }
            } catch (Exception e) {
                System.err.println("Error parsing file: " + filePath);
                System.err.println("  Error message: " + e.getMessage());
                // Add to failed files list
                failedFiles.add(filePath);
            }
}
static void saveEdge(ArrayList<Pair<Integer, Integer>> parseResult,
    String filePath, String fileName) throws IOException {
        System.out.println(filePath + "/" + fileName);
        BufferedWriter out = new BufferedWriter(new FileWriter(filePath + "/" + fileName)
        for (Pair<Integer, Integer> item : parseResult) {
            out.write(String.format("%d,%d\n", item.a, item.b));
        }
        out.close();
    }
```

This an example of DOT and DOT TXT files:

```
digraph {
n0 [label="root (CompilationUnit)"];
n1 [label="types"];
n0 -> n1;
n2 [label="type (ClassOrInterfaceDeclaration)"];
n1 -> n2;
n3 [label="isInterface='false'"];
n2 -> n3;
n4 [label="name (SimpleName)"];
n2 -> n4;
n5 [label="identifier='Sample'"];
n4 -> n5;
n6 [label="members"];
n2 -> n6;
n7 [label="member (MethodDeclaration)"];
n6 -> n7;
n8 [label="body (BlockStmt)"];
n7 -> n8;
n9 [label="statements"];
n8 -> n9;
n10 [label="statement (ExpressionStmt)"];
n9 -> n10;
n11 [label="expression (VariableDeclarationExpr)"];
n10 -> n11;
n12 [label="variables"];
n11 -> n12;
n13 [label="variable (VariableDeclarator)"];
n12 -> n13;
n14 [label="initializer (FieldAccessExpr)"];
n13 -> n14;
n15 [label="name (SimpleName)"];
n14 -> n15;
n16 [label="identifier='length'"];
n15 -> n16;
n17 [label="scope (NameExpr)"];
n14 -> n17;
n18 [label="name (SimpleName)"];
```

Figure 4.2: Illustration of an AST DOT file

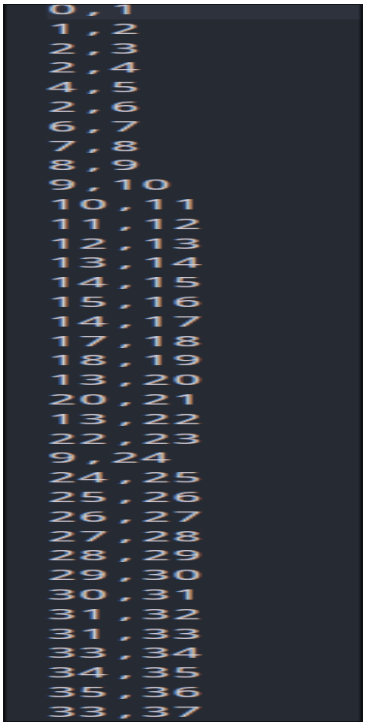


Figure 4.3: Illustration of an AST DOT TXT file

and this is the python code to generate adjacency matrices for all files and save as a numpy file

```
def build_ast(input_dir, output_dir, matrix_size=600):
    os.makedirs(output_dir, exist_ok=True)
    # Created batch for saving RAM
    # Get list of files
    filenames = []
    matrices = []
    for file in tqdm(os.listdir(input_dir), desc="Processing AST files"):
        filename = file.split(".dot")[0]
        filenames.append(filename)

        try:
            file_path = os.path.join(input_dir, file)
            # Read file in one go (more efficient)
            with open(file_path) as f:
                data = f.read().splitlines()
            array = np.zeros((matrix_size, matrix_size), dtype=np.uint8)
            for line in data:
                    try:
```

```
                        src, dst = map(int, line.split(","))
                        if 0 <= src < matrix_size and 0 <= dst < matrix_size:
                            array[src, dst] = 1
                        # No else:break which would skip valid edges
                        # after an invalid one
                    except ValueError:
                        continue
            array = apply_metapath(array)  # Apply Metapath to make it symmetric
            matrices.append(array)

        except Exception as e:
            print(f"Error reading {file}: {e}")
            break
    if not matrices:
        print("No valid matrices found. Exiting.")
        return
    # Stack matrices
    final_matrices = np.stack(matrices)
    print(f"Saving matrices with shape {final_matrices.shape}...")
    np.save(os.path.join(output_dir, "ast_metapath.npy"), final_matrices)
    np.save(os.path.join(output_dir, "ast_filenames.npy"), np.array(filenames))
```

Note:

To implement MetaPath, we needed to add reversed edges for each pair of nodes connected by a directed edge for AST, DDG and CFG adjacency matrices and for that we used this simple python function:

```
def apply_metapath(matrix):
    """
    Apply Metapath algorithm to make the matrix symmetric

    Args:
        matrix: A numpy array or sparse matrix

    Returns:
        A symmetric matrix after applying the Metapath algorithm
    """
    return np.maximum(matrix, matrix.T)
```

4.4.2 Generating CFG

For generating CFGs, we used Joern [16], an open-source tool designed for the robust analysis of source code, bytecode, and binary code, with support for multiple programming languages, including Java. We selected Joern because it is built on top of JavaParser, which has demonstrated high accuracy in state-of-the-art research. Using Joern, we automated the generation of DOT files for each dataset sample through a Bash script that launches the tool. Since Joern supports scripting with Scala, we were also able to run pre-scripts, which further simplified and streamlined the process of generating CFGs. Then we converted the generated CFG DOT files to DOT TXT and finally building the CFG adjancy matrices and saving them into a numpy file all using python code.
Since the bash script is little bit long , we will only display the relevant part that launches the joern tool:

```
    # Function to process a single batch
process_batch() {
    local batch_file=$1
    local batch_num=$(basename "$batch_file" | sed 's/batch_//')
    local batch_dir="$TEMP_DIR/batch_"$batch_num"_dir"


    # Create batch directory
    mkdir -p "$batch_dir"

    echo "Starting batch $batch_num ($(wc -l < "$batch_file") files)"
    # Run Joern for this batch with allocated memory
"$JOERN_CLI_PATH" --script "./generate_dot_cfg.sc" --param fileListPath="$batch_file"
    --param outputDir="$OUTPUT_DIR" 2> "$batch_dir/errors.log"

    local exit_status=$?
    if [ $exit_status -ne 0 ]; then
        echo "[$batch_num] Errors occurred during CFG generation"
        echo "[$batch_num] See $batch_dir/errors.log for details"

        # Add to failed files
        echo "Batch $batch_num" >> "$OUTPUT_DIR/failed_batches.txt"
    fi

    echo "[$batch_num] Batch processing complete"
```

```
    # Count results for this batch
    local batch_dots=$(find "$OUTPUT_DIR" -newer "$batch_file" -name "*.dot" | wc -l)
    echo "[$batch_num] Generated $batch_dots DOT files"
}
```

and this the python function used to build CFG adjacency matrices:

```
def build_cfg_adj_matrices(dotTxtfiles_folder, output_dir,  matrix_size=200):
    """
    Build CFG adj matrcies   from the generated txt files in single numpay array.
    """
    matrices = []
    filenames = []
    for file in tqdm(os.listdir(dotTxtfiles_folder), desc="Building adjacency matrices",
        filename = file.split(".dot")[0]
        filenames.append(filename)
        try:
                file_path = os.path.join(dotTxtfiles_folder, file)

                with open(file_path, "r") as f:
                    data = f.read().splitlines()
                array = np.zeros((matrix_size, matrix_size), dtype=np.uint8)
                for line in data:
                    try:
                        src, dst = map(int, line.split(","))
                        if 0 <= src < matrix_size and 0 <= dst < matrix_size:
                            array[src, dst] = 1
                    except ValueError:
                            continue
                array = apply_metapath(array)  # Apply metapath if needed
                matrices.append(array)
        except Exception as e:
                print(f"Error processing {file}: {e}")
    matrices = np.stack(matrices)
    print(f"Saving matrices with shape {matrices.shape}...")
    np.savez_compressed(os.path.join(output_dir, "cfg_metapath.npz"), matrices=matrices)
    np.save(os.path.join(output_dir, "cfg_filenames.npy"), np.array(filenames))
```

```
    #np.save(os.path.join(output_dir, "cfg.npy"), matrices)
    print(f"Saved {len(matrices)} matrices to {output_dir}")
```

This is an example of a CFG Generated DOT file:

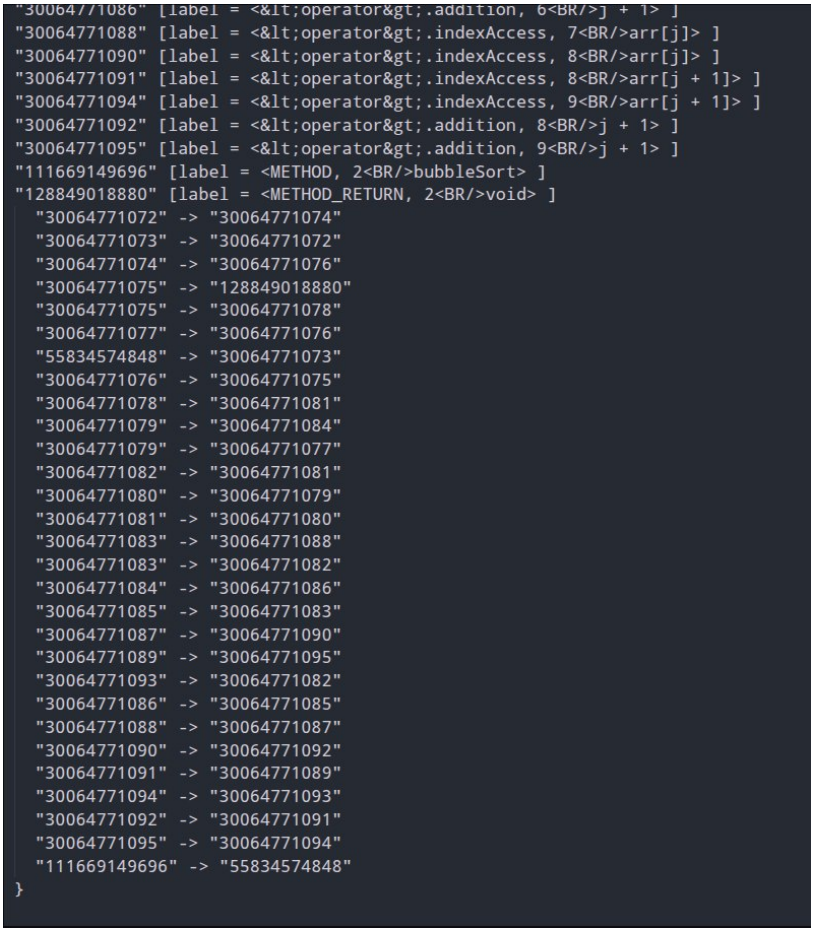
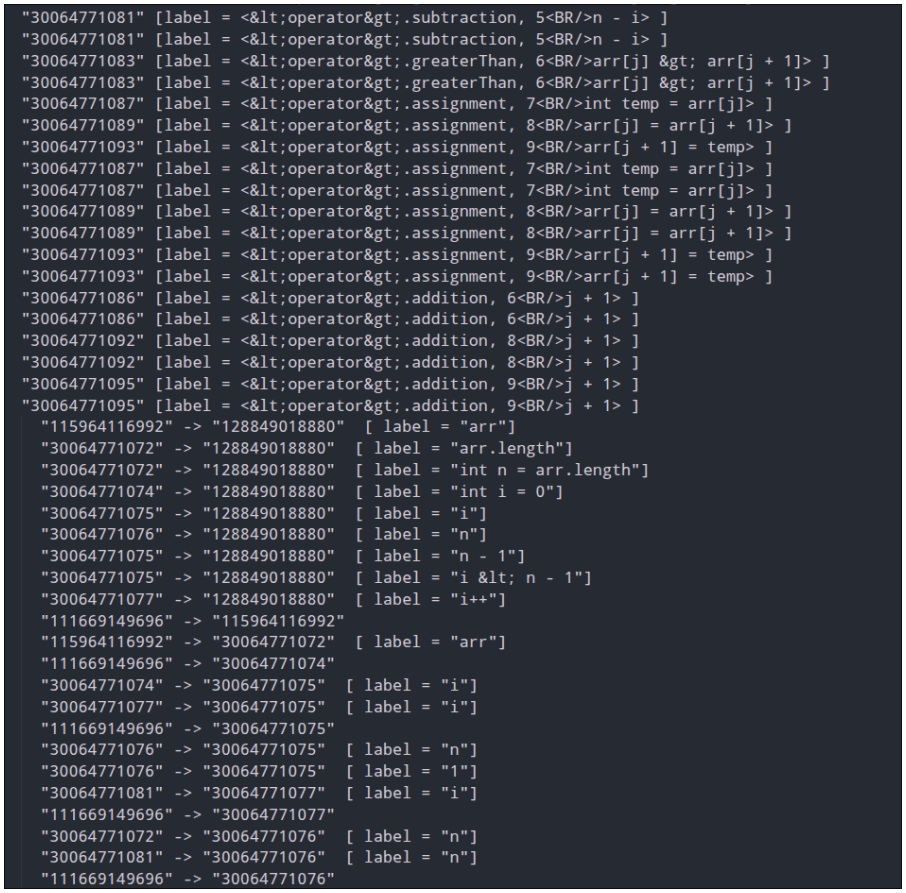


Figure 4.4: Illustration of a CFG DOT file

and for the CFG DOT TXT file see Figure 4.3, because it is similar.

4.4.3 Generating DDG

For generating DDGs, we also employed Joern [16], following the same process used for CFGs. The only modification was adapting the ”generate_dot_cfg.sc” script provided to Joern so that it produces DDG DOT files instead of CFG ones (see this code for details).
This is an example of a DDG Generated DOT file:



```
"30064771081" [label = <&lt;operator&gt;.subtraction, 5<BR/>n - i> ]
"30064771081" [label = <&lt;operator&gt;.subtraction, 5<BR/>n - i> ]
"30064771083" [label = <&lt;operator&gt;.greaterThan, 6<BR/>arr[j] &gt; arr[j + 1]> ]
"30064771083" [label = <&lt;operator&gt;.greaterThan, 6<BR/>arr[j] &gt; arr[j + 1]> ]
"30064771087" [label = <&lt;operator&gt;.assignment, 7<BR/>int temp = arr[j]> ]
"30064771089" [label = <&lt;operator&gt;.assignment, 8<BR/>arr[j] = arr[j + 1]> ]
"30064771093" [label = <&lt;operator&gt;.assignment, 9<BR/>arr[j + 1] = temp> ]
"30064771087" [label = <&lt;operator&gt;.assignment, 7<BR/>int temp = arr[j]> ]
"30064771087" [label = <&lt;operator&gt;.assignment, 7<BR/>int temp = arr[j]> ]
"30064771089" [label = <&lt;operator&gt;.assignment, 8<BR/>arr[j] = arr[j + 1]> ]
"30064771089" [label = <&lt;operator&gt;.assignment, 8<BR/>arr[j] = arr[j + 1]> ]
"30064771093" [label = <&lt;operator&gt;.assignment, 9<BR/>arr[j + 1] = temp> ]
"30064771093" [label = <&lt;operator&gt;.assignment, 9<BR/>arr[j + 1] = temp> ]
"30064771086" [label = <&lt;operator&gt;.addition, 6<BR/>j + 1> ]
"30064771086" [label = <&lt;operator&gt;.addition, 6<BR/>j + 1> ]
"30064771092" [label = <&lt;operator&gt;.addition, 8<BR/>j + 1> ]
"30064771092" [label = <&lt;operator&gt;.addition, 8<BR/>j + 1> ]
"30064771095" [label = <&lt;operator&gt;.addition, 9<BR/>j + 1> ]
"30064771095" [label = <&lt;operator&gt;.addition, 9<BR/>j + 1> ]
  "115964116992" -> "128849018880"  [ label = "arr"]
  "30064771072" -> "128849018880"  [ label = "arr.length"]
  "30064771072" -> "128849018880"  [ label = "int n = arr.length"]
  "30064771074" -> "128849018880"  [ label = "int i = 0"]
  "30064771075" -> "128849018880"  [ label = "i"]
  "30064771076" -> "128849018880"  [ label = "n"]
  "30064771075" -> "128849018880"  [ label = "n - 1"]
  "30064771075" -> "128849018880"  [ label = "i &lt; n - 1"]
  "30064771077" -> "128849018880"  [ label = "i++"]
  "111669149696" -> "115964116992"
  "115964116992" -> "30064771072"  [ label = "arr"]
  "111669149696" -> "30064771074"
  "30064771074" -> "30064771075"  [ label = "i"]
  "30064771077" -> "30064771075"  [ label = "i"]
  "111669149696" -> "30064771075"
  "30064771076" -> "30064771075"  [ label = "n"]
  "30064771076" -> "30064771075"  [ label = "1"]
  "30064771081" -> "30064771077"  [ label = "i"]
  "111669149696" -> "30064771077"
  "30064771072" -> "30064771076"  [ label = "n"]
  "30064771081" -> "30064771076"  [ label = "n"]
  "111669149696" -> "30064771076"
```

Figure 4.5: Illustration of a DDG DOT file

and for the DDG DOT TXT file see Figure 4.3, because it is similar.

4.4.4 Generating CSS

We utilized the HuggingFace Transformers library [34], which provides a wide range of pre-trained models. Within this framework, we employed the UniXcoder model [14], a unified cross-modal pre-trained model for programming languages that supports both code understanding and generation tasks. Prior to generating embeddings, code snippets were tokenized into token sequences using the UniXcoder tokenizer. However, we observed some incorrect tokenization.
To mitigate this issue, we extended UniXcoder's vocabulary by traversing our datasets, recording tokens absent from its vocabulary, and appending them as a custom word list. This adjustment improved tokenization consistency. For each code snippet, UniXcoder then produced a semantic embedding matrix that encoded its semantic information. It is important to note that UniXcoder limits input length to 512 tokens; thus, any snippet exceeding this limit was truncated using overlapping window method. Finally, the embeddings will be all saved as a numpy file to be used later on.
Here it is the function used to generate code embeddings:

```
def generate_code_embeddings(code_snippets, model, device):
    embeddings = []
    window_size = 450
    overlap = 150
    windows = []
    for func in code_snippets:
        init_tokens = model.tokenizer.tokenize(func)
        if (len(init_tokens) <= 508): # for special chars added by the unixcoder.
        tokens_ids = model.tokenize([func], max_length=512, mode="<encoder-only>",
            padding=True)
            source_ids = torch.tensor(tokens_ids).to(device)
            with torch.no_grad():  # Disable gradient calculation for inference
               tokens_embeddings, func_embeddings = model(source_ids)
            func_embeddings = func_embeddings.cpu().detach().numpy()
            embeddings.append(func_embeddings)
        else: # Here Processing long code
            windows = []
            window_embeddings = []
            for i in range(0, len(init_tokens), window_size - overlap):
                end_idx = min(i + window_size, len(init_tokens))
                window_tokens = init_tokens[i:end_idx]
                if len(window_tokens) < 100:
```

```
                    continue
                # Convert tokens back to string for processing
                window_text = model.tokenizer.convert_tokens_to_string(window_tokens)
                windows.append(window_text)
            for window in windows:
                tokens_ids = model.tokenize([window], max_length=512,
                mode="<encoder-only>", padding=True)
                source_ids = torch.tensor(tokens_ids).to(device)
                with torch.no_grad():
                    _, window_embedding = model(source_ids)
                window_embeddings.append(window_embedding.cpu().detach().numpy())

                # Combine window embeddings (with average pooling)
            if window_embeddings:
                combined_embedding = np.mean(np.vstack(window_embeddings), axis=0,
                keepdims=True)
                embeddings.append(combined_embedding)
            else:
                # Fallback for edge cases - just use truncated version
                tokens_ids = model.tokenize([func], max_length=512, mode="<encoder-only>"
                 padding=True)
                source_ids = torch.tensor(tokens_ids).to(device)
                with torch.no_grad():
                    _, func_embeddings = model(source_ids)
                func_embeddings = func_embeddings.cpu().detach().numpy()
                embeddings.append(func_embeddings)

            # Optional: Free memory
            if torch.cuda.is_available():
                torch.cuda.empty_cache()

    if embeddings:
        return np.vstack(embeddings)
    else:
        return np.array([])
```

4.4.5 Quad self-attention Layer

The Quad Self-Attention layer is a custom module composed of four MVSAs (Multi-View Self-Attention), each corresponding to one representation of the code: AST, CFG, DDG, and CSS. We implemented the MVSA using keras[7] and Tensorflow[2] a powerful deep learning framework that provides a high-level interface for designing, training, and evaluating neural networks.
Here it is the code for the MVSA Layer:

```
from keras import initializers
from keras.layers import Layer
import tensorflow as tf

class MyMultiViewAttention(Layer):
    def __init__(self, output_dim, num_head, kernel_initializer='glorot_uniform',
        **kwargs):
        self.output_dim = output_dim
        self.num_head = num_head
        self.kernel_initializer = kernel_initializer
        super(MyMultiViewAttention, self).__init__(**kwargs)

    def build(self, input_shape):
        initializer_object = initializers.get(self.kernel_initializer)
        self.W = self.add_weight(name='W',
            shape=(self.num_head, 3, input_shape[2], self.output_dim),
            initializer=initializer_object,
            trainable=True)
        self.Wo = self.add_weight(name='Wo',
            shape=(self.num_head * self.output_dim, self.output_dim),
            initializer=initializer_object,
            trainable=True)
        self.built = True

    def call(self, x):
        q = tf.linalg.matmul(x, self.W[0, 0])
        k = tf.linalg.matmul(x, self.W[0, 1])
        v = tf.linalg.matmul(x, self.W[0, 2])

        e = tf.linalg.matmul(q, k, transpose_b=True)
        e = e / (self.output_dim ** 0.5)
```

```
        e = tf.nn.softmax(e)
        outputs = tf.linalg.matmul(e, v)

        for i in range(1, self.W.shape[0]):
            q = tf.linalg.matmul(x, self.W[i, 0])
            k = tf.linalg.matmul(x, self.W[i, 1])
            v = tf.linalg.matmul(x, self.W[i, 2])

            e = tf.linalg.matmul(q, k, transpose_b=True)
            e = e / (self.output_dim ** 0.5)
            e = tf.nn.softmax(e)
            o = tf.linalg.matmul(e, v)
            outputs = tf.concat([outputs, o], axis=-1)

        z = tf.linalg.matmul(outputs, self.Wo)
        return z
    def compute_output_shape(self, input_shape):
        return input_shape[0], input_shape[1], self.output_dim

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            "output_dim": self.output_dim,
            'num_head': self.num_head,
            'kernel_initializer': self.kernel_initializer
        })
        return config
```

4.4.6 Final classification Layer

The output of the Quad Self-Attention layer is a single fused matrix, which serves as input to the **CNN module**. A sequence of convolutional layers (Conv2D) with progressively fewer filters (32, 16, and 8) is applied, each followed by batch normalization, **LeakyReLU activation**, and either **dropout** or **max-pooling** to enhance stability, capture hierarchical features, and mitigate overfitting. The resulting feature maps are then flattened into a one-dimensional vector and passed through a fully connected dense layer with 1024 units. Finally, a dense layer with a **sigmoid activation** outputs a probability indicating whether a given code snippet is vulnerable. The model takes four inputs corresponding to different program representations (CSS, DDG, CFG, and AST), and is compiled using the **Adam optimizer** with a learning rate of 1×10^{-5}, **binary cross-entropy loss**, and evaluated with **accuracy**, **precision**, and **recall** metrics.

4.5 Use Cases Scenarios

To demonstrate the practical applicability of our proposed model, we present a set of use case scenarios. These scenarios illustrate how the system operates in real-world settings and how it can assist developers in identifying and mitigating vulnerabilities during the software development process.

4.5.1 Web Application for Java Functions classification

To provide an accessible interface for our model, we developed a web application using the **Django**[9] framework. This application allows developers to interact with the vulnerability detection system in a simple and efficient manner. The workflow is as follows: a user enters a Java method into the web application, which then forwards the code snippet to the backend for processing. The deployed model analyzes the method and classifies it as either vulnerable or safe. Once the analysis is complete, the user is redirected to a results page where the prediction is displayed . This use case demonstrates how the proposed model can be seamlessly integrated into a practical tool that assists developers in evaluating the security of their code during the development process, without requiring any direct interaction with the underlying machine learning pipeline.

This is the main interface in our Web application **java-function classifier**:

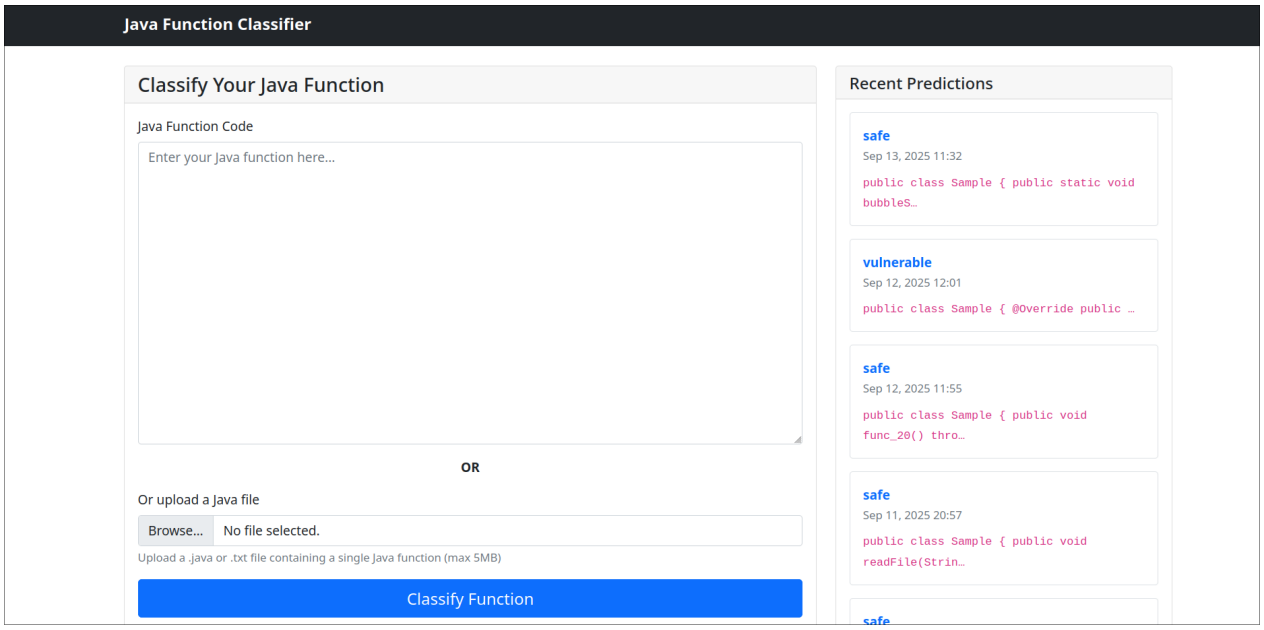


Figure 4.6: Illustration of the Home page of java-function classifier web app

and this is the prediction result interface:

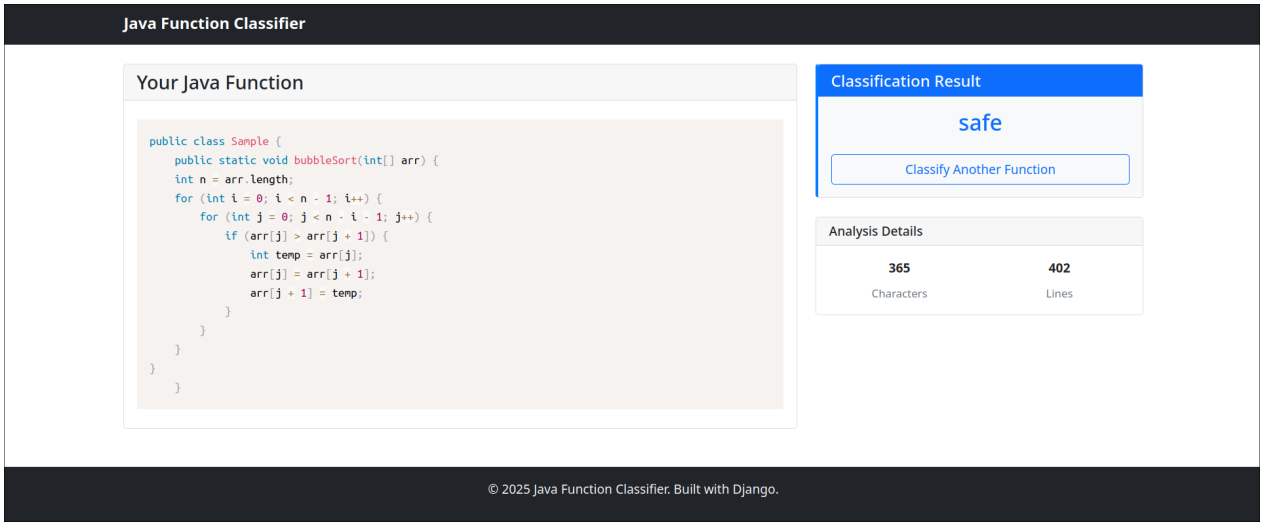


Figure 4.7: Illustration of the result page of java-function classifier web app

Also we made a prediction History Dashboard only accessible by admins which will showcase the functions code, time of making a prediction and the prediction result as shown in Figure 4.8

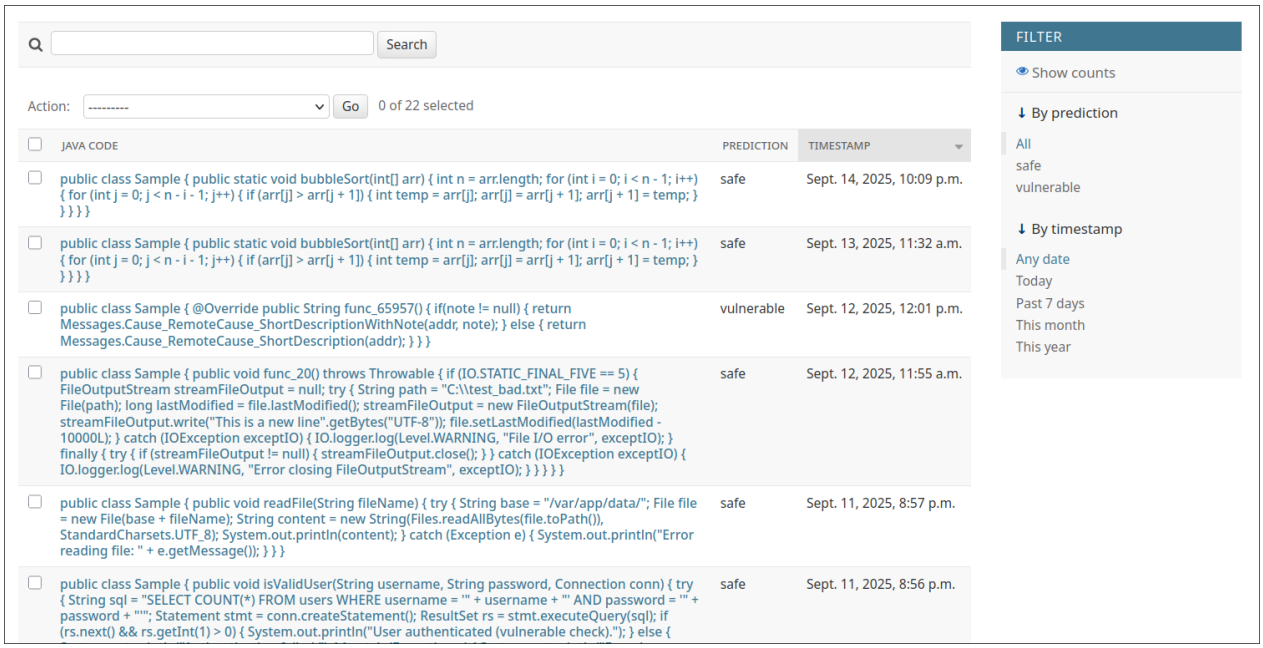


Figure 4.8: Illustration of the prediction history dashboard of thr java-functions classifier web app

4.5.2 VS Code Extension

Visual Studio Code (VS Code)[17] is a free, open-source, cross-platform source code editor developed by Microsoft. It has become one of the most widely adopted integrated development environments (IDEs) in both academia and industry, thanks to its lightweight architecture, extensibility, and support for a wide range of programming languages and frameworks. Unlike traditional heavyweight IDEs, VS Code combines the performance of a simple text editor with powerful development tools, making it suitable for projects ranging from rapid prototyping to large-scale software engineering.

We developed a custom Visual Studio Code (VS Code) to integrate the trained model for vulnerability detection directly into the developer's environment. As illustated in Figure 4.9, The extension communicates with a Django REST API hosting the model, allowing developers to classify Java functions as either safe or vulnerable without leaving the editor. It supports two modes of input: selecting a function from the source code or pasting code into a dedicated text box, for illustration see Figure 4.10. Once submitted, the prediction is returned instantly and displayed

as a notification, ensuring lightweight and non-intrusive feedback, see Figure 4.11. Built with TypeScript and the VS Code extension API, it is both modular and extensible, making it possible to expand the analysis to other languages or provide vulnerability-specific insights. By embedding vulnerability classification directly into the development workflow, this extension promotes early detection of security flaws and exemplifies the shift-left security approach in software engineering.

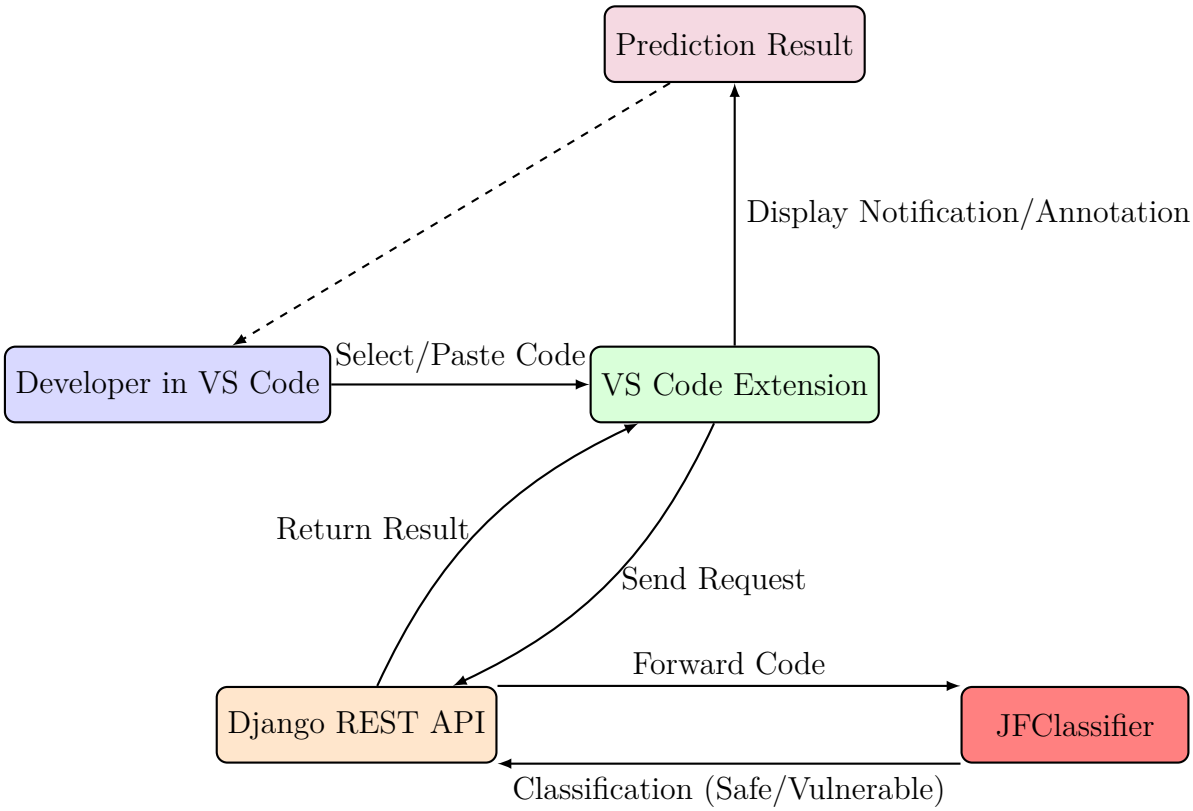


Figure 4.9: Workflow of the jfc extension integrated into VS Code.

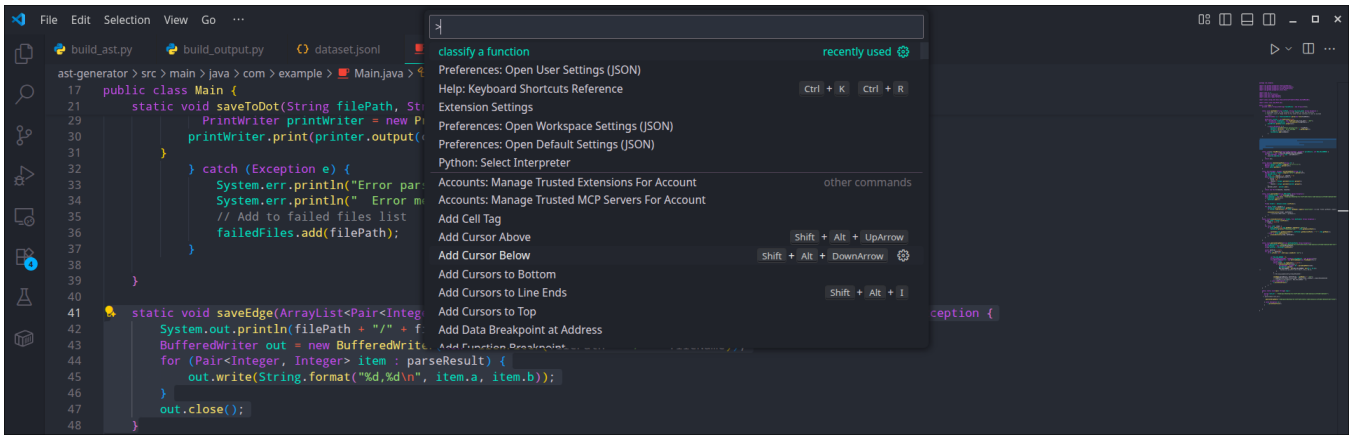


Figure 4.10: Illustration of jfc extension usage

Figure 4.11: Illustration of jfc extension prediction example

4.6 Conclusion

In this chapter, we detailed the implementation of the proposed deep learning architecture for automated vulnerability detection in Java source code. We described the process of generating structural representations (AST, CFG, DDG) and semantic embeddings (CSS), and presented how these views were fused using the Multi-View Self-Attention encoder. A custom Quad Self-Attention Layer was designed and integrated with convolutional and dense layers to capture hierarchical patterns and predict code vulnerabilities with improved accuracy.

Furthermore, we explained the deployment of the model through a Django-based web application and a Visual Studio Code extension, making the system accessible to developers in practical environments.

Overall, this chapter demonstrated how the proposed model was implemented and deployed in real-world use cases. This solution shows demonstrated the potential of combining program representations with advanced deep learning mechanisms to enhance the accuracy and usability of software vulnerability detection tools.

Chapter 5

Analysis and Results

5.1 Introduction

This chapter presents the experimental results obtained from evaluating the proposed deep learning architecture for Java vulnerability detection. We begin by outlining the datasets, metrics, and evaluation protocols used in our experiments. The performance of the model is then analyzed through quantitative results, including accuracy, precision, recall, F1-score, and confusion matrix visualization.

5.2 Constructed Dataset

We managed to gather a total of 69,850 samples. These samples are divided into two binary classes: Vulnerabe and Safe. As shown in Figure 5.1, The analysis reveals an uneven distribution between these two classes where:

- **Sample Count**: The majority class, "Safe", contains 49,846 instances, while the minority class, "Vulnerable", consists of 20,004 instances.
- **Proportions**: The "Safe" class constitutes 71.4% of the entire dataset, compared to 28.6% for the "Vulnerable" class.

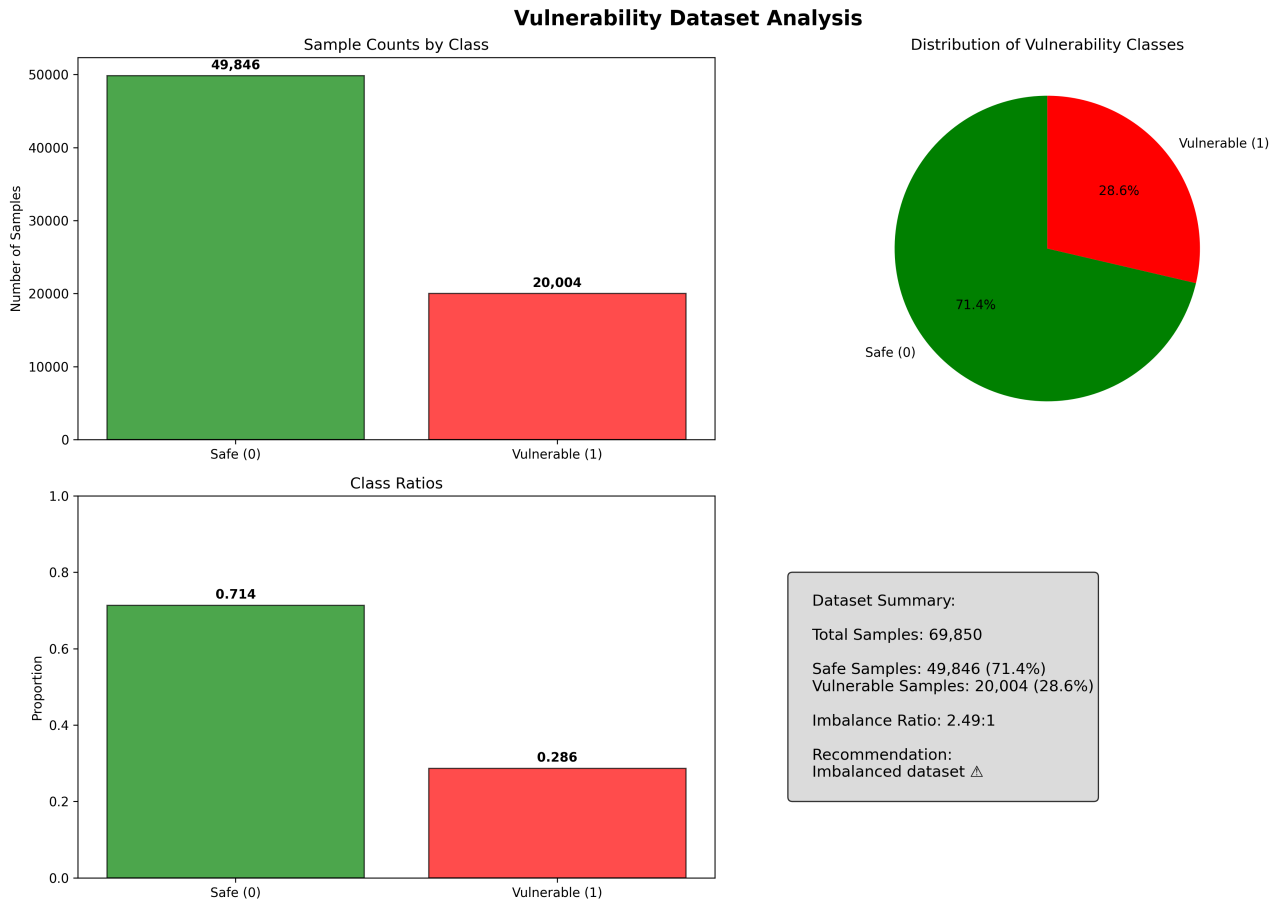


Figure 5.1: Statistical analysis of the dataset

5.2.1 Training Split

To train our model, the dataset was divided into three subsets: 80% for training, 10% for validation, and 10% for testing. The distribution of samples across these subsets is as follows:

- **Training set**: 55,880 samples
- **Validation set**: 6,985 samples
- **Test set**: 6,985 samples

5.3 Model Performance

In this section, we present the primary performance metrics used to evaluate the model. Given that our dataset is imbalanced, with a ratio of approximately 2.5[1], we selected Precision, Recall, and F1-Score in addition to Accuracy as evaluation metrics.

5.3.1 Accuracy

Accuracy is the ratio of the number of samples correctly predicted by the model to the total number of samples. For our model we got an Accuracy of 91.88% on the Validation Set and 91.55% on the Testing set.

5.3.2 Precision

Precision measures the proportion of code snippets predicted as vulnerable that are actually vulnerable. Our model achieved a precision of 90.72%, indicating that it produced relatively few false positives (safe code incorrectly flagged as vulnerable).

5.3.3 Recall

Recall measures the proportion of genuinely vulnerable code snippets that were correctly identified by the model. On the testing set, our model achieved a recall of 80.63%, meaning that while most vulnerabilities were detected, some were still missed (false negatives).

5.3.4 F1-score

F1 score is the harmonic mean of precision and recall, calculated as follows:

$$\textbf{F1-Score} = \frac{2 * Precision * Recall}{Precision + Recall}$$

The model achieved an F1-score of 85.93% on the testing set. This high value demonstrates that the model maintains a good balance between precision and recall, effectively detecting vulnerabilities while minimizing false positives.

[1]This indicates that for every vulnerable sample in the dataset, there are about 2.5 safe samples

5.3.5 Confusion Matrix

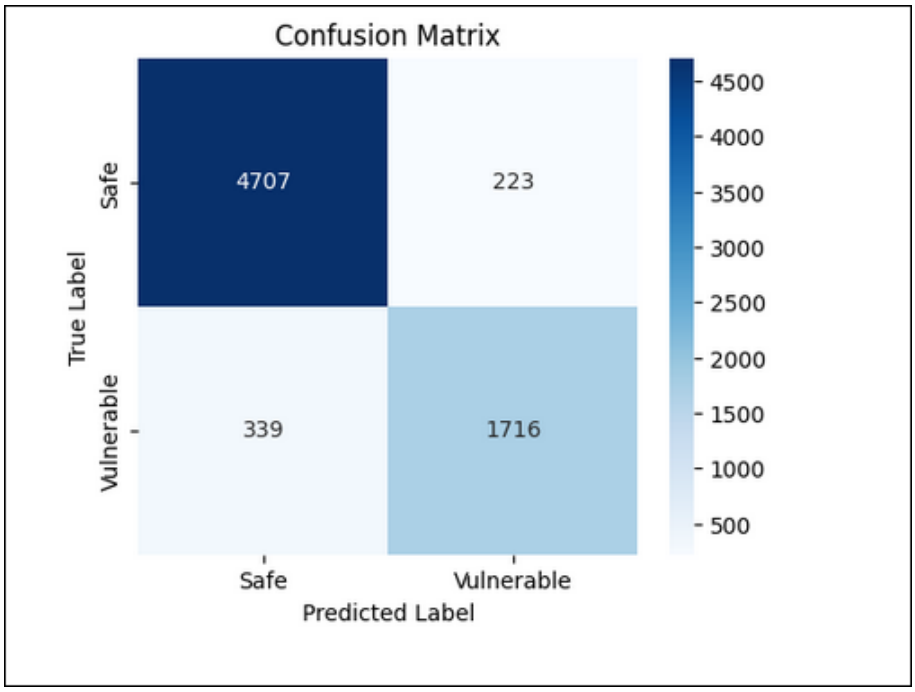


Figure 5.2: Confusion Matrix on the Testing set

The confusion matrix in Figure 5.2 summarizes the performance of our model on the testing dataset. Out of all safe code snippets, the model correctly classified 4,707 as safe, while misclassifying 223 as vulnerable (false positives). For vulnerable code snippets, the model correctly identified 1,716 as vulnerable, but misclassified 339 as safe (false negatives).

These results indicate that the model performs strongly in distinguishing between safe and vulnerable code. The relatively low number of false positives shows that the model rarely flags safe code incorrectly, while the moderate number of false negatives suggests that some vulnerabilities are still missed. This balance is consistent with the precision of 90.72% and recall of 80.63%, as previously reported, and is further reflected in the overall F1-score of 85.93%.

and Figure 5.3 illustrates the confusion matrix in percentage form.

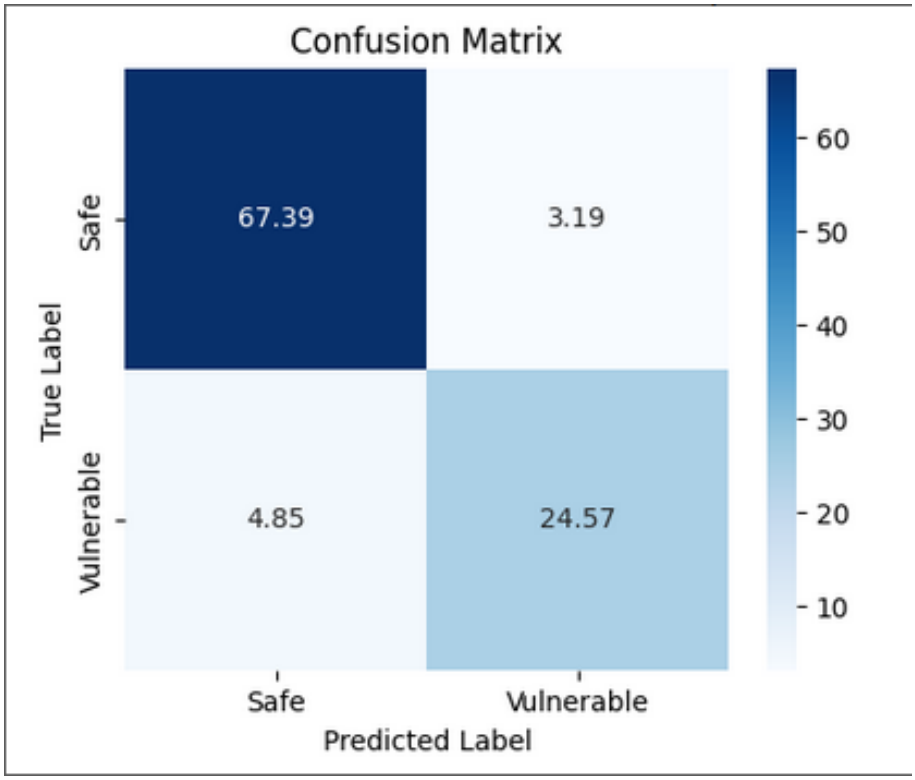


Figure 5.3: Confusion Matrix on the Testing set percentage-wise

5.4 Conclusion

This chapter evaluated the proposed deep learning architecture for vulnerability detection using a constructed dataset of 69,850 Java code samples. Despite the dataset's inherent imbalance (71.4% safe vs. 28.6% vulnerable), the model achieved strong results, with an accuracy of 91.55%, precision of 90.72%, recall of 80.63%, and an F1-score of 85.93%. These results confirm that the model is capable of effectively distinguishing between vulnerable and safe code snippets while maintaining a balance between minimizing false positives and detecting as many true vulnerabilities as possible.

The confusion matrix further highlighted the model's strengths, showing that false positives remain relatively rare, whereas a moderate number of false negatives still occur. This indicates that while the model is highly reliable in avoiding incorrect alerts, some vulnerabilities may still go undetected. Overall, the results validate the effectiveness of combining multiple program representations with advanced deep learning mechanisms and demonstrate the potential of the proposed approach for practical software security applications.

Part III

Conclusion and Future Works

Conclusion and Future Works

Conclusion

This thesis has addressed the critical problem of detecting software vulnerabilities through static code analysis enhanced by deep learning techniques. Traditional approaches to static analysis, while widely adopted, often suffer from high false-positive rates and limited ability to capture deep semantic and structural relationships within source code. To overcome these limitations, we investigated advanced program representations such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Dependence Graphs (DDGs), along with semantic embeddings (CSS), to provide a richer understanding of code behavior.

Building upon state-of-the-art research, we implemented and adapted the JFinder architecture, introducing a Quad Self-Attention Layer to effectively combine multiple views of code into a unified representation. This architecture, coupled with convolutional and dense layers, demonstrated strong performance in classifying Java functions as vulnerable or safe. The evaluation on our dataset confirmed that our approach achieves a robust balance between precision, recall, and F1-score, highlighting its potential for practical adoption.

Beyond theoretical contributions, this work emphasized usability and integration. We deployed the model through a Django-based web application and a Visual Studio Code extension, enabling developers to analyze code directly within their workflow. These tools illustrate how research prototypes can be transformed into accessible, real-world solutions that support secure software development.

In summary, this thesis demonstrates that combining program representations with deep learning architectures can significantly enhance automated vulnerability detection. At the same time, the study highlights ongoing challenges such as dataset quality, scalability, and generalization to diverse programming languages. Future research should focus on standardized benchmarks, improved interpretability of AI

models, and hybrid systems that integrate static and dynamic analysis.

The convergence of software engineering, program analysis, and artificial intelligence opens a promising pathway toward building more secure software systems. While much work remains, the results of this thesis contribute a step toward the long-term goal of reliable, automated, and developer-friendly vulnerability detection.

Future Work

While this thesis has demonstrated the potential of deep learning techniques for automated vulnerability detection in Java, several directions remain open for further exploration:

- **Extension to multiple programming languages:** The current model is trained on Java code only. Extending the approach to other widely used languages such as C, C++, or Python would broaden its applicability and validate its generalization capabilities.
- **Hybrid analysis techniques:** Combining static analysis with dynamic or symbolic execution could improve coverage and reduce false negatives, particularly for vulnerabilities that are difficult to detect from source code alone.
- **Dataset quality and diversity:** A critical challenge remains the scarcity of large, diverse, and realistic datasets. Future work should focus on constructing standardized benchmarks that better reflect real-world codebases and vulnerabilities.
- **CWE-specific models:** Instead of training a single model to detect all types of vulnerabilities, future work could explore building a set of specialized models, each dedicated to a specific CWE category. By training on datasets filtered for a single vulnerability type, these models could learn more fine-grained patterns and potentially achieve higher accuracy for their targeted CWE. Such an ensemble of CWE-specific classifiers could then be combined into a unified framework, allowing for both specialized detection and comprehensive coverage across multiple vulnerability types.
- **Scalability and efficiency:** Optimizing the model for faster inference and lower resource consumption would make it more suitable for large-scale industrial projects and continuous integration pipelines.

- **Integration with development pipelines:** Deeper integration with IDEs, CI/CD systems, and code review tools would ensure that vulnerability detection becomes a natural part of the software development lifecycle.

Pursuing these directions would not only enhance the accuracy and robustness of automated vulnerability detection systems but also contribute to their practical adoption in real-world software engineering environments.

Bibliography

[1] OWASP team. OWASP Benchmark Project. https://owasp.org/www-project-benchmark/, 2016.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Alexander Obregon. Understanding Java's Security Architecture. https://medium.com/@AlexanderObregon/understanding-javas-security-architecture-c5fa0925d318, 2023. Accessed on: 2025-05-15.

[4] Alexandru G. Bardas. STATIC CODE ANALYSIS. *Research Paper in Economics*, 2012.

[5] Aphex34. Typical CNN Architecture. https://commons.wikimedia.org/wiki/File:Typical_cnn.png, 2015.

[6] BLACKDUCK. Coverity scan static analysis. https://scan.coverity.com/.

[7] Chollet, François and others. Keras. https://keras.io, 2015.

[8] S. Conté. Software weaknesses detection using static-code analysis and machine learning techniques. Master's thesis, Instituto Politécnico de Viana do Castelo, September 2023.

[9] Django-team. Django framework. https://www.djangoproject.com/.

[10] findbugs team. Findbugs™ - find bugs in java programs. https://findbugs.sourceforge.net/.

[11] GeekforGeeks. Evaluation Metrics For Classification Model in Python. https://www.geeksforgeeks.org/evaluation-metrics-for-classification-model-in-python/, 2025. Accessed on: 2025-05-17.

[12] GeekforGeeks. Machine Learning Tutrial. https://www.geeksforgeeks.org/machine-learning/, 2025. Accessed on: 2025-05-17.

[13] GeekforGeeks. Transformers in machine learning. https://www.geeksforgeeks.org/machine-learning/getting-started-with-transformers/, 2025. Accessed on: 2025-05-17.

[14] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[15] javaparser. Java 1-21 parser and abstract syntax tree for java with advanced analysis functionalities. https://github.com/javaparser/javaparser.

[16] joern team. Open-source code analysis platform for c/c++/java/binary/-javascript/python/kotlin based on code property graphs. https://github.com/joernio/joern.

[17] Microsoft. Visual studio code the open source ai code editor. https://code.visualstudio.com/.

[18] MITRE assosiation. About CWE. https://cwe.mitre.org/about/index.html. Accessed on: 2025-05-15.

[19] NSA Center for Assured Software. Juliet Java 1.3 Test suite #111. https://samate.nist.gov/SARD/test-suites/111, 2017.

[20] ORACLE team. Overview of Java Security Models. https://docs.oracle.com/cd/E12839_01/core.1111/e10043/introjps.htm#JISEC1819, 2025. Accessed on: 2025-05-15.

[21] OWASP. Java Security Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Java_Security_Cheat_Sheet.html. Accessed on: 2025-05-15.

[22] pmd team. Pmd source code analyzer. https://pmd.github.io/.

[23] Ravil Mussabayev. Finetuning Large Language Models for Vulnerability Detection. https://github.com/rmusab/vul-llm-finetune. Accessed on: 2025-06-15.

[24] SAMATE team. NIST Software Assurance Reference Dataset. https://samate.nist.gov/SARD/. Accessed on: 2025-06-15.

[25] A. Shestov, R. Levichev, R. Mussabayev, E. Maslov, P. Zadorozhny, A. Cheshkov, R. Mussabayev, A. Toleu, G. Tolegen, and A. Krassovitskiy. Finetuning large language models for vulnerability detection. *IEEE Access*, 13:38889–38900, 2025.

[26] J. Shrestha. static program analysis. Master's thesis, Uppsala University, September 2013.

[27] sonarsources sa. Automated code quality and security reviews. https://www.sonarsource.com/products/sonarqube/.

[28] soot team. Soot - a framework for analyzing and transforming java and android applications. https://soot-oss.github.io/soot/.

[29] spotbugs team. Spotbugs find bugs in java programs. https://spotbugs.github.io/.

[30] Tony Ekpo. Understanding Java's Security Model. https://www.linkedin.com/pulse/understanding-javas-security-model-tony-ekpo-l5uff/, 2025. Accessed on: 2025-05-15.

[31] wala team. wala. https://github.com/wala/WALA.

[32] J. Wang1, Z. Huang1, H. Xiao, and Y. Xiao. jfinder: A novel architecture for java vulnerability identification based quad self-attention and pre-training mechanism. *High-Confidence Computing*, 3:2667–2952, 2023.

[33] Wikipedia. Java (programming language). https://en.wikipedia.org/wiki/Java_(programming_language), 2015. Accessed on: 2025-05-15.

[34] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush. Transformers: State-of-the-art natural language processing. In

Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pages 38–45, Online, Oct. 2020. Association for Computational Linguistics.