A framework for secure coding: real-time detection of custom secure coding guideline violations

I.B.G.T.C. Bowala

2024

A framework for secure coding: real-time detection of custom secure coding guideline violations

A dissertation submitted for the Degree of Master of Computer Science

I. B. G. T. C. Bowala
University of Colombo School of Computing
2024

DECLARATION

Name of the student: Ihala Bowala Gedara Thilanka Chathurangi Bowala
Registration number: 2019/MCS/012
Name of the Degree Programme: Master of Computer Science
Project/Thesis title: A Framework for Secure Coding: Real-time detection of custom secure coding guideline violations

1. The project/thesis is my original work and has not been submitted previously for a degree at this or any other University/Institute. To the best of my knowledge, it does not contain any material published or written by another person, except as acknowledged in the text.

2. I understand what plagiarism is, the various types of plagiarism, how to avoid it, what my resources are, who can help me if I am unsure about a research or plagiarism issue, as well as what the consequences are at University of Colombo School of Computing (UCSC) for plagiarism.

3. I understand that ignorance is not an excuse for plagiarism and that I am responsible for clarifying, asking questions and utilizing all available resources in order to educate myself and prevent myself from plagiarizing.

4. I am also aware of the dangers of using online plagiarism checkers and sites that offer essays for sale. I understand that if I use these resources, I am solely responsible for the consequences of my actions.

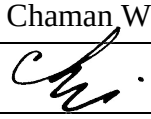
5. I assure that any work I submit with my name on it will reflect my own ideas and effort. I will properly cite all material that is not my own.

6. I understand that there is no acceptable excuse for committing plagiarism and that doing so is a violation of the Student Code of Conduct.

Signature of the Student	**Date (DD/MM/YYYY)**
	26-09-2024

Certified by Supervisor(s)

This is to certify that this project/thesis is based on the work of the above-mentioned student under my/our supervision. The thesis has been prepared according to the format stipulated and is of an acceptable standard.

	Supervisor 1	**Supervisor 2**
Name	Prof. G.D.S.P. Wimalaratne	Dr. Chaman Wijesiriwardana
Signature		
Date	26-09-2024	26-09-2024

This thesis is dedicated to my parents, family, teachers, lecturers, and friends.

For their endless love, support, and encouragement

ACKNOWLEDGEMENTS

This prototype-based research is the result of my continuous commitment with great support from various personnel who assisted me in numerous ways.

My very special gratitude goes to Prof. Prasad Wimalaratne, my main supervisor, for giving me the idea of the project and providing the background and prior work and providing guidance and necessary support to complete this project in a successful manner. He had been a great motivator and an advisor for me to overcome major obstacles faced during this project.

I also appreciate the great support provided by Dr. Chaman Wijesiriwardana, my co-supervisor, and lecturer at the University of Moratuwa for the assistance given to me, related to the depth of theoretical and subject wise matters of this project.

I also appreciate the work of Dasanayake, S. L., Mudalige, A. & Perera, M. L. T., who started the initial stage of this secure coding guideline prototype and appreciate the support given to me by providing their previous work-related documents and the codebase. This research is mainly based on their thesis (Dasanayake, et al., 2019).

This work builds upon the foundational research established by earlier scholars in the field. I also appreciate their research work.

My special thanks are extended to the evaluating panel for providing important feedback and for showing me the areas to improve.

Finally, I would like to extend my deepest gratitude to my beloved family members for supporting me in achieving my goals during my academic years and overcoming unexpected barriers in my life.

ABSTRACT

Secure Software Development refers to the process of developing software applications with minimized security vulnerabilities. In the release or maintenance phase of the Software Development Life Cycle (SDLC), fixing specific bugs is more expensive than correcting during the development phase. Therefore, it is essential to minimize software vulnerabilities within the coding phase by adhering to a set of coding best practices that are referred to as secure coding guidelines (SCG).

Following secure coding guidelines manually is challenging due to the lack of knowledge among developers. Further, distributing and following a set of custom secure coding guideline provided by the organization or the security expert of the development team is more challenging and time consuming. Therefore, software developers tend to commit code with secure coding guideline violations.

Currently there exist very few research studies which support detecting secure coding guideline violations on the fly in an Integrated Development Environment (IDE) along with custom rule generation. There is a research gap that needs to be addressed in this domain. This research study focuses on addressing the gaps in the specified domain.

The research study proposes a prototype-based framework that focuses on providing a new rule creation mechanism aiming to filling a gap in the rule creation domain. Further, focuses on developing a mechanism to automate the process of detecting secure coding guideline violations found in a source code of a software application, defined by the proposed rule creation mechanism.

The prototype is an IntelliJ IDEA based plugin and sample rules created for the evaluation are for java source code. The Artificial intelligence markup language (AIML) based proposed rule creation mechanism was able to define secure coding rules filling the existing gap, and the provided prototype-based framework was able to detect violations of these rules, benefiting the software development research area.

Key phrases: Secure coding guideline violations, Secure coding rule creation, Artificial intelligence markup language, Static code analysis, Software Development Life Cycle

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVATIONS

AIML	Artificial intelligence markup language
ALICE	Artificial Linguistic Internet Computer Entity
ASA	Automatic static analysis
ASIDE	Application Security IDE
AST	Abstract Syntax Tree
CI/CD	Continuous Integration and Continuous Delivery
CVE	Common Vulnerability and Exposures database
CWE	Common Weakness Enumeration
CxQL	extensive Query Language
CxSAST	Checkmarx Static Application Security Testing
ENISA	European Union Agency for Cybersecurity
FOD	Fortify on Demand
FSA	Fortify Security Assistant
FSCA	Fortify Static Code Analyzer
ICS	Industrial Control Systems
ICSE	International Conference on Software Engineering
IDE	Integrated Development Environment
NVD	National Vulnerability Database
OWASP	Open Web Application Security Project
RDF	Resource Description Framework
RML	Rule Markup Language
RuleML	Rule Markup Language for the Web
SAAS	Software as a service
SAFECode	Software Assurance Forum for Excellence in Code
SCG	Secure Coding Guidelines
SCPL	Source Code Pattern Language
SDLC	Software Development Life Cycle
SEI-CERT	Java secure coding guidelines standards
US	United States
USD	United States Dollars
VG	Veracode Greenlight
VSA	Veracode Static Analysis

VUDENC	Vulnerability Detection with Deep Learning on a Natural Codebase
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

CHAPTER 1
INTRODUCTION

1.1 Motivation

During the period from May 2001 to January 2014, lines of codes in the eclipse platform project have grown from 283,229 to 2,674,685, thus in thirteen years, size has grown almost 10 folds (Tantithamthavorn, et al., 2014). In December 2019, it was 12,925,016 lines, thus the size has increased approximately by 45 times since the start (Synopsys, 2019). This explosive growth of eclipse has increased more rapidly than the ability of a human to maintain them and therefore it has also increased the complexity of the software by a significant amount (Tantithamthavorn, et al., 2014) (Synopsys, 2019).

According to research studies, businesses spent an average of 380 million United States Dollars (USD) in 2017 (Gasiba, et al., 2021) to recover and deal with the consequences of Industrial Control Systems (ICS) incidents, and this value is still increasing. The total cost of poor software quality in the United States (US) for the year 2020 has estimated as 2.08 trillion USD and the large majority (75%, or an estimated $1.56 trillion) of it is due to software failures caused by the failure to patch known vulnerabilities (Armerding, 2021). As per new repots, in the year 2022, it was $2.41 trillion (Synopsys, 2022) (McGuire, 2022).

Software security is a critical need, therefore, the paradigm shift of *Building Security In* has emerged in recent decades (Abeyrathna, et al., 2020) (Wijesiriwardana, et al., 2020). This paradigm shift requires software security to be addressed in all phases of the software development lifecycle (Abeyrathna, et al., 2020) (Wijesiriwardana, et al., 2020) (Khan, et al., 2022) (Humayun, et al., 2022). This concept is called *Secure Software Engineering*.

There are different practices, methodologies, and tools to prevent the introduction of software security vulnerabilities. Most security vulnerabilities result from defects that are unintentionally introduced during the design phase and the implementation phase (Abeyrathna, et al., 2020) (Wijesiriwardana, et al., 2020). Garry McGraw has identified code reviews and architectural risk analysis as the top two best practices to minimize the software security vulnerabilities (McGraw, 2005, p. 101) (Abeyrathna, et al., 2020). Most of the companies in the software industry do not focus much on investing in making the code secure

but mainly consider penetration testing along with patching, after the development (McGraw, 2005, p. 182) (Dasanayake, et al., 2019) (Humayun, et al., 2022). It is important to detect software vulnerabilities after development but making software developers follow *secure code practices* (Synopsys Editorial Team, 2020) (Software Assurance Forum for Excellence in Code (SAFECode), 2018) helps developers to prevent security vulnerabilities while coding.

There are secure coding practices well documented, such as Carnegie Mellon's Software Engineering Institute C, C++, and Java secure coding guidelines standards (SEI-CERT), the Open Web Application Security Project (OWASP), checklists to follow while writing the source code (Concea-Prisăcaru, et al., 2023) (Gasiba, et al., 2021) (Carnegie Mellon University - Software Engineering Institute, 2018) (Software Assurance Forum for Excellence in Code (SAFECode), 2018) (The OWASP Foundation, 2017). Making software developers follow *secure code practices* while writing code will reduce the number of security vulnerabilities in the application by a significant amount, and it outweighs the cost (Wijesiriwardana, et al., 2020) (Abeyrathna, et al., 2020) (McGraw, 2005, pp. 273 - 295).

Though following a secure coding practice is a must, due to the complexity and length, it is hard to even for an experienced developer to remember all of these. Further, studies show that 53.7% of software developers do not know secure coding guidelines and more than 50% of software developers cannot spot security vulnerabilities in code (Gasiba, et al., 2021).

The most popular way to stop software security vulnerabilities in the development phase is using *static code analysis* (Alenezi & Almuairfi, 2019). There are different static analysis tools such as SonarQube, find-sec-bugs and Microsoft DevSkim (Software Assurance Forum for Excellence in Code (SAFECode), 2018) (Dasanayake, et al., 2019). All these tools attempt to highlight possible code issues within *static* (non-running) source code.

Static analysis tools that can plug directly into the Integrated Development Environment (IDE) allows developers to find security vulnerabilities effectively without leaving their native IDE environment, and this is an important feature for a static analysis tool (Software Assurance Forum for Excellence in Code (SAFECode), 2018). Further, it is important to detect vulnerability issues and notify developers in real time, to provide authoring time guidance to developers as they write code, so they can fix the issue at the time of introduction (Software Assurance Forum for Excellence in Code (SAFECode), 2018).

In most organizations, there are security experts as a separate team, or at least there are

developers within the project team, that might have expertise in security (Cremer, et al., 2020). Security experts guide other developers by providing them with guidelines and checklists. These instructions sometimes might not be clear to all developers, and even if they are understood, that does not guarantee the developer will be able to apply them in practice. To help improve this communication, it is important to provide a formal method to distribute rules.

Therefore, according to the literature, one of the major requirements of the tool is to share knowledge easily. Thus, customization and distribution of the secure code guideline rules are important (Cremer, et al., 2020). The rules customization must be scalable. The tool should not provide rules (Cremer, et al., 2020). The tool should allow security experts in the organization to distribute their secure guidelines related to their concepts. Further, it should allow developers to share project or team specific guidelines among them. Thus, there should be a formal mechanism to convert the user-specific rules into algorithms (Cremer, et al., 2020) (Dasanayake, et al., 2019). Furthermore, rule creation must be easy, fast, and versatile.

According to the literature, there are different *drawbacks and weaknesses* that arise *when using available static analysis tools*. They are,

I. Though there are some tools to support static code analysis, most tools detect security bugs only, and they do not provide a mechanism adhering secure coding guidelines (Cremer, et al., 2020) (Dasanayake, et al., 2019).
II. Some tools require special training; therefore, it requires additional steps (Gasiba, et al., 2021). Thus, hard to extend these tools for detecting violations of custom rules.
III. Static code analysis often generates false positives and false negatives. Thus, tools may not be able to detect certain vulnerabilities (Gasiba, et al., 2021).
IV. Even though there are some commercial applications with IDE support; most of them do not provide real-time solutions (Cremer, et al., 2020).
V. Tools with rule customization are limited (Cremer, et al., 2020).

Some research studies have been conducted to resolve the above issues, but still, existing research studies have *research gaps that must be filled*.

I. Studies show that, in the current context, there exists no automated mechanism to support software developers adhere to *secure coding guidelines* during the coding phase, or automation levels of existing studies are very little (Cremer, et al., 2020) (Dasanayake, et al., 2019).

II. Existing tools focus on detecting *security vulnerabilities and source code quality issues* but do not detect *secure coding guideline violations* (Dasanayake, et al., 2019).

III. Issues of rule generation - Even though there are few research studies to adhering secure coding guidelines with IDE support; most of them *do not provide custom rule creation support*. Certain guidelines or rules depend on the user. *Custom rule generation* (Cremer, et al., 2020) and a *formal mechanism to convert user-dependent rules into algorithms* (Dasanayake, et al., 2019) are still issues.

IV. Tools with rule customization still can't provide support for *complex rule creation* or *package level rule creation* due to limitations of rule creation mechanisms. The reason is that currently available *custom rule creation methods do not support tracking of the control flow of the source code*. Addressing limitations of custom rule creation still exist.

It has also been found that most vulnerabilities found were caused by ignoring proper secure programming practices by the developers because software developers lack the skills to judge whether they comply with the secure coding guidelines (Gasiba, et al., 2021) (Khan, et al., 2022) (Stefanovska, et al., 2022). Further, available custom rule creation methods have some major limitations and therefore, cannot create complex or package level custom secure coding guidelines using available systems (Cremer, et al., 2020) (YAML Org, 2021). In this research project, the aim is to address this problem. If the tool can support creating any secure coding guideline provided by the organization and can notify or warn a developer of possible vulnerabilities or insecure coding practices in real time, he would commit while coding. This will result in saving time and money that the company will need to fix these vulnerabilities later (Khan, et al., 2022). Then the developers will also become aware of good coding practices.

1.2 Statement of the problem

Following secure coding guidelines is the major solution to prevent software security vulnerabilities, and providing tools to distribute and check secure coding guidelines is the way to adhere. Though there are some research studies for on-the-fly detection of secure coding guidelines, in the custom secure coding rule creation domain, there is a gap that needs to be addressed. This research work concerns the problem of *available on-the-fly detection supporting systems do not support creating complex custom rules which need the tracking of*

the control flow of the code and *available on-the-fly detection supporting systems do not support creating custom rules for package level secure coding guideline violations* (Gasiba, et al., 2021) (Cremer, et al., 2020) (Dasanayake, et al., 2019) and no proper mechanism to *convert user-dependent complex or package level rules into algorithms* (Cremer, et al., 2020) (Dasanayake, et al., 2019) in existing research studies.

1.3 Aim and Objectives

The goal of this research is to build a prototype with custom package level and custom complex secure coding rule generation support and detect user defined complex secure coding guideline violations in real-time to assist and encourage software developers to adhere to these guidelines. The main objectives of this research are,

I. Conducting a literature review on secure coding (Gasiba, et al., 2021) (synopsys, 2019) and identification of existing approaches for detecting secure coding guideline violations and, their limitations (Dasanayake, et al., 2019) (Gasiba, et al., 2021).
II. Study available secure rule define mechanisms (Cremer, et al., 2020) (Fernandez & Mujica, 2011) of existing research studies, and based on the literature review, and by comparing and evaluating existing approaches, finding an approach to define custom complex secure coding guidelines.
III. Designing a methodology to integrate custom complex rule generation part into the proposed framework.
IV. Implementation of the plugin – provide the support for creating complex and package level secure coding guidelines and notify developers about potential violations of user-defined secure coding rules while they are coding in a real-time manner.
V. Study secure coding guidelines (synopsys, 2019) and identify the most suitable set of rules (Dasanayake, et al., 2019) (Carnegie Mellon University - Software Engineering Institute, 2018) (The OWASP Foundation, 2017) that are to be used to verify the prototype.
VI. Evaluate the capability of the plugin-based framework to create complex and package level rules and capability of detecting specified secure coding rule violations.

1.4 Scope

The proposed framework in this dissertation is only focused on addressing the mentioned limitations of the secure coding rule creation mechanisms of available research studies.

Therefore, focuses on achieving creations of complex and package-level custom secure coding guidelines and violation detection of created rules. The research focuses on the coding phase of the SDLC. The framework will focus on the selected Integrated Development Environment and implementation depends on the developed interpreter for the selected rule creation language.

Further, evaluation of the rule creation method and the prototype limited to a selected set of secure coding guidelines as a verification and will be using projects which are developed using Java Programming language.

1.5 Structure of the Thesis

The remaining sections of this thesis are as follows. Chapter 2 is associated with the literature review and background study related to the project. Chapter 3 provides a detailed explanation of the architecture of the project and the implementation of the project. Chapter 4 illustrates the evaluation methodologies along with their results and Chapter 5 concludes the dissertation along with a discussion regarding future work.

CHAPTER 2
LITERATURE REVIEW

This chapter provides the main theoretical background for this research work.

2.1 Background

During the period from May 2001 to January 2014, lines of codes in the eclipse platform project have grown from 283,229 to 2,674,685, thus in thirteen years, the size of the Eclipse Platform project has grown almost 10 folds (Tantithamthavorn, et al., 2014). In December 2019, it was 12,925,016 lines, thus the size of the code has increased approximately by 45 times since the start (Synopsys, 2019). This explosive growth of eclipse has increased more rapidly than the ability of a human to maintain them. Therefore, recognizing where and how a feature is implemented in the source code based on a given requirement, to implement new features, to enhance existing features, or to fix bugs must be done very carefully and time-consuming as well for developers (Tantithamthavorn, et al., 2014). Thus, this exponential growth of code has also increased the complexity of the software by a significant amount (Tantithamthavorn, et al., 2014) (Synopsys, 2019).

According to ‘Statista’, the number of software security vulnerabilities and exposes has grown year over year, achieving its peak in 2022, which is 25,227 (Petrosyan, 2023). Both National Vulnerability Database (NVD) and Common Vulnerability and Exposures (CVE) database, recorded over 14,500 new vulnerabilities in 2017 compared with 6,000 from 2016 (ENISA - European Union Agency for Cybersecurity, 2018). In 2018, 2,000 new vulnerabilities were reported during the first two months of the year (Chiu, 2020), which is equal to the number observed during the entire year of 2002 (ENISA - European Union Agency for Cybersecurity, 2018).

Some studies found that 96 percent of all scanned applications contain some open-source components (Synopsys, 2023) (Veracode, n.d.). According to the Open-Source Security and Risk Analysis Report 2023 by Synopsys, 84% of codebases contained at least one known open-source vulnerability, and 48% of the codebases contained high-risk vulnerabilities (Synopsys, 2023).

2020 was a challenging year for the software development industry with the pandemic. Within

days, companies were forced to adjust to remote work, which introduced new types of security threats (Goldstein, 2021). According to the 'WhiteSource' database, the number of published open-source software vulnerabilities in 2020 rose by over 50% compared to the previous year (Goldstein, 2021).

According to research studies, businesses spent an average of 380 million USD in 2017 (Gasiba, et al., 2021) to recover and deal with the consequences of ICS incidents, and this value is still increasing. A new report estimates the cost of poor software quality at 2.41 trillion USD for the U.S. in 2022 (McGuire, 2022). For the year 2020 this has been estimated as 2.08 trillion USD and the large majority (75%, or an estimated $1.56 trillion) of it is due to software failures caused by the failure to patch known vulnerabilities (Armerding, 2021).

These facts convey an important message regarding software security and the importance of taking actions to prevent vulnerabilities. It could be concluded that the software industry should take more actions to prevent vulnerability introduction in software, rather than detecting after vulnerability introduction and correction. Furthermore, can conclude that open-source software is a good point to initiate this.

2.1.1 Secure Software Engineering - Building Security In

The software security field is relatively new since it originated in the early 2000s (Lipner, 2014) and this is the major reason why secure software practices have not been widely adopted by software developers. Today, security is a necessary part of most software development projects. Software security best practices mainly involve incorporating security early in the software life cycle, identifying and understanding common threats, designing for security, and subjecting all software artefacts to thorough objective risk analyses and testing (McGraw, 2005, p. 26) (Stefanovska, et al., 2022) (Humayun, et al., 2022) (Khan, et al., 2022) (Concea-Prisăcaru, et al., 2023). According to literature, relative cost of fixing defects is higher in the maintenance phase than the coding phase (Dawson, et al., 2010). In Eclipse, if secure coding practices had not been used from the beginning when writing the code, it could imply that a massive number of vulnerabilities may exist in the code and patching such several vulnerabilities after following a testing phase would consume quite a lot of time. Even the detection of vulnerabilities that would have risen due to insecure code may be extremely difficult.

Having identified the critical need for software security, the paradigm shift of *Building Security In* has emerged in recent decades (Abeyrathna, et al., 2020) (Wijesiriwardana, et al., 2020) (McGraw, 2005, p. 47). A security problem is more likely to arise because of a problem in a system's standard-issue part than in some given security feature. This is an important reason why software security must be part of a full lifecycle approach (McGraw, 2005, p. 47). This paradigm shift requires software security to be addressed in all phases of the software development lifecycle (Abeyrathna, et al., 2020) (Wijesiriwardana, et al., 2020). This concept is called *Secure Software Engineering*. Secure software mainly involves incorporating software security into the software development process ensuring application confidentiality, integrity, and availability (Alenezi & Almuairfi, 2019) (Khan, et al., 2022).

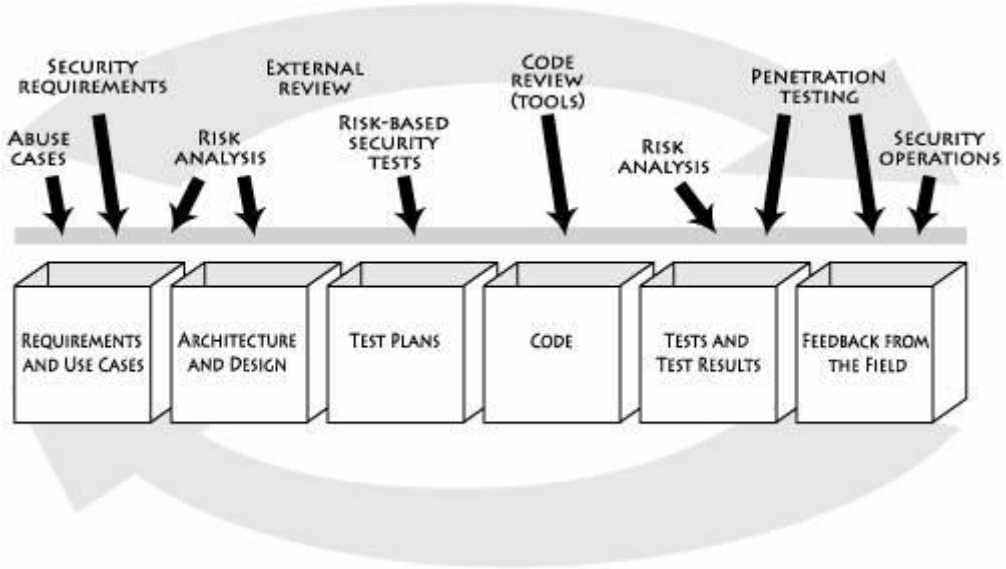


Figure 1: Software security best practices applied to various software artifacts (McGraw, 2005, p. 48)

Figure 1 specifies the software security touchpoints (a set of best practices) applied to various software artifacts. Although in this picture the artifacts are laid out according to a traditional waterfall model, most organizations follow an iterative approach today, which means that best practices will be cycled through more than once as the software evolves (McGraw, 2005, p. 48).

The descending order of effectiveness of the seven touchpoints has been identified as follows (Microsoft, 2024) (Dasanayake, et al., 2019) (McGraw, 2005, p. 101):

Code review → Risk analysis → Penetration testing → Risk analysis → Abuse cases → Security requirements → Security operations

It could be seen from the above order of effectiveness, the importance of code reviews and that they mainly involve examining the source code, identifying issues, and correcting them to improve source code quality. Also, it could be concluded that source code plays a major role in building secure software since code reviews are associated with the source code.

2.1.2 Secure coding guidelines and practices

The United States Department of Homeland Security estimates that most security incidents can be attributed to defects in software design and code (Gasiba, et al., 2021) (Khan, et al., 2022). Literature reveals that most security vulnerabilities result from defects that are unintentionally introduced into the software during the design phase and the implementation phase (Abeyrathna, et al., 2020) (Wijesiriwardana, et al., 2020). These facts convey an important message regarding the significance of following *secure coding practices* while writing the source code since it will reduce the time and cost of developing less vulnerable software applications (Concea-Prisăcaru, et al., 2023) (Khan, et al., 2022) (Synopsys Editorial Team, 2020) (Software Assurance Forum for Excellence in Code (SAFECode), 2018).

There are secure coding practices well documented, such as Carnegie Mellon's Software Engineering Institute C, C++, and Java secure coding guidelines standards (SEI-CERT) (Carnegie Mellon University - Software Engineering Institute, 2018) (Gasiba, et al., 2021), the Open Web Application Security Project (OWASP) (Gasiba, et al., 2021) (Software Assurance Forum for Excellence in Code (SAFECode), 2018) in their checklist (The OWASP Foundation, 2017) while writing the source code. Making software developers follow *secure code practices* while writing code will reduce the number of security vulnerabilities in the application by a significant amount (Wijesiriwardana, et al., 2020) (Abeyrathna, et al., 2020) (McGraw, 2005, pp. 273 - 295), and it outweigh the cost. Out of above secure code guidelines, some of the most popular secure coding practices are OWASP and SEI CERT (Concea-Prisăcaru, et al., 2023) (Gasiba, et al., 2021).

2.1.2.1 Awareness of people on software vulnerabilities

However, according to surveys done by various parties, it is hard to even for an experienced developer to remember all of these (Gasiba, et al., 2021). Further, studies show that 53.7% of software developers do not know secure coding guidelines (Gasiba, et al., 2021) (GitLab, 2019). Furthermore, more than 50% of software developers cannot spot security vulnerabilities in code (Gasiba, et al., 2021) (GitLab, 2019).

According to a survey conducted in 2021 (Gasiba, et al., 2021) most software developers (76%) disagree that SCG should be overlooked to deliver software faster. Further, software developers also disagree that secure coding guidelines should not be ignored to get a job done. Furthermore, they show that, software developers might lack the skills to judge whether they comply with the secure coding guidelines (Gasiba, et al., 2021).

Using tools to search the code to identify deviation from requirements helps verify that developers are following secure code guidelines and helps identify problems early in the software development cycle. This relieves developers of having to make special efforts to ensure that coding standards are being consistently followed (Software Assurance Forum for Excellence in Code (SAFECode), 2018).

Thus, literature conclude that, software industry should use automation tools to search the code to identify deviation from requirements to verify that developers are following secure code guidelines and to identify problems early in the software development cycle. Further, literature conclude that this relieves developers of having to make special efforts to ensure that coding standards are being consistently followed (Software Assurance Forum for Excellence in Code (SAFECode), 2018).

2.1.4 Knowledge distribution

In most organizations, there are security experts as a separate team, or at least there are developers within the project team, that might have expertise in security (Cremer, et al., 2020). Security experts guide other developers by providing them with guidelines and checklists. Coding guidelines are a result of knowledge sharing (Cremer, et al., 2020).

Take the case that some security-minded developers research a software security vulnerability in the codebase. They may have identified it themselves or have been made aware through an analysis or penetration test report. To fix the problem, they dig through documentation until they can patch the vulnerability. To help other software developers in the same team to recognize and avoid similar mistakes, it is usually easier to explain how to write the code to prevent the vulnerability, rather than explaining how the prevented attack works. Therefore, sharing secure coding guidelines with non-experts is easier than sharing an explanation of a type of attack that needs to prevent (Cremer, et al., 2020).

Take the case that the organization security expertise distributes instructions to prevent

vulnerabilities in a code base. These instructions sometimes might not be clear to all developers, and even if they are understood, that does not guarantee the developer will be able to apply them in practice. To help improve this communication, it is important to provide a formal method to distribute rules.

Therefore, according to the literature, one of the major requirements of the tool is to share knowledge easily. Thus, customization and distribution of the secure code guideline rules are important (Cremer, et al., 2020). The tool should allow security experts in the organization to distribute their secure guidelines related to their concepts. Further, it should allow developers to share project or team specific guidelines among them.

Thus, literature concludes that tools that are used to enforce secure coding guidelines should provide support for custom rule generations. The rules customization must be scalable. The tool should not provide rules (Cremer, et al., 2020) (Dasanayake, et al., 2019). Furthermore, rule creation must be easy, fast, and versatile.

2.2 Ensuring security in source code - Automation

To ensure coding standards are being consistently followed, there are some tools already in the industry. They differ in terms of usability, time taken to detect vulnerability, vulnerability detection phase, IDE support, on fly detection, supporting languages, support for user-own secure code rule generations and many more. Also, there are some previous research studies done by researchers as well.

2.2.1 Available Commercial Static Analysis Tools

To ensure code quality, efficient tools are required to help them avoid the introduction of such security bugs, and therefore write more secure code. Automatic static analysis (ASA) tools have been proven effective in uncovering security-related bugs early enough in the software development process (Siavvas, et al., 2018). Their main characteristic is that they are applied directly to the source or compiled code of the system, without requiring its execution (Siavvas, et al., 2018). A list of commercial static analysis tools is available under Appendix E: Available Commercial Static Analysis Tools.

However, since the results of ASA tools comprise long lists of raw warnings (i.e., alerts) or absolute values of software metrics, the literature concludes that they do not provide real insight to the stakeholders of the software products (Siavvas, et al., 2018). In fact, a great number of ASA tools have been proposed over the years providing a huge volume of such raw

data (Siavvas, et al., 2018), which may contain security-relevant information that may be useful for secure software development. Hence, the literature concludes that appropriate knowledge extraction tools are needed on top of the raw produced by ASA tools for facilitating the production of secure software (Siavvas, et al., 2018).

As concluded in previous section, the coding phase of the Secure SDLC is extremely important. To minimize the introduction of security vulnerabilities during the coding phase, an automated mechanism to support software developers adhere to secure coding guidelines is needed. According to the comparison in Appendix section, (Appendix E: Available Commercial Static Analysis Tools), in the current context, tools mainly focus on detecting security vulnerabilities and source code quality issues but do not detect secure coding guideline violations. Thus, can further conclude that, currently, most commercial applications do not provide a mechanism that adheres to secure coding guidelines. Furthermore, can conclude that even though there are some commercial applications with IDE support, most of them do not provide real-time solutions and they do not provide creating custom secure coding guidelines. Thus, currently, there is no commercial application that adheres secure coding guidelines with real-time IDE support and custom secure coding guideline generation support. Therefore, currently there does not exist any commercial automated mechanism to support software developers adhere to secure coding guidelines on fly, which supports creating custom secure coding guidelines and converting user dependent rules into algorithms.

2.3 Related research work

2.3.1 VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection

Hanif, et al. (Hanif & Maffeis, 2022-07), have conducted a research study, a deep learning approach to detect security vulnerabilities in source code. Their approach pre-trains a RoBERTa model with a custom tokenisation pipeline on real-world code from open-source C/C++ projects. The model learns a deep knowledge representation of the code syntax and semantics, which they leverage to train vulnerability detection classifiers. A significant and unsolved problem with vulnerability detection datasets is that manual inspection reveals occasional label inaccuracies. Deep learning should be resilient to label noise in training, the presence of noise during testing undermines the quantitative performance results. Further, this supports only for detect security vulnerabilities, not secure coding guideline violations.

Furthermore, according to them, although their models are relatively small, they are still expensive to train. Thus, can conclude that, even if we try to extend this model to detect secure coding guideline violations, using this in real time detection of custom secure coding guideline violations is not practical due to the manual inspection, expensiveness of the training process and the need of the training for each and every rule.

2.3.2 Recognizing lines of code violating company-specific coding guidelines using machine learning

Ochodek, et al. (Ochodek, et al., 2020) have conducted a research study, to support code reviews by automatically recognizing company-specific code guidelines violations in large-scale, industrial source code. Researchers have constructed a machine-learning-based tool for code analysis. Using this, software developers and architects can use a few examples of source code lines of violating codes or design guidelines to train decision-tree classifiers to find similar violations in their codebases. Researchers has developed this research for C/C++ languages. In this, they have trained the tool several rounds, for detection of one rule violation, to achieve the desired accuracy level. The best results have obtained for the rules requiring understanding the context of a single line. They have found that, rules requiring to understand the context of multiple lines are far more difficult to train. Thus, can conclude that, practically, when the rule set becomes larger and lines of code of a rule becomes higher, this research doesn't support scaling up the usage, due to the above reason and the multiple time training.

2.3.3 VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python

Wartschinskia, et al. (Wartschinski, et al., 2022) have conducted a research study, to automate vulnerability detection in Python source code, called VUDENC (Vulnerability Detection with Deep Learning on a Natural Codebase). They have built a deep learning-based vulnerability detection tool that automatically learns features of vulnerable code from a large and real-world Python codebase. To create the basis for VUDENC, a large dataset of commits has mined from GitHub and labelled according to the commit context. The raw source code has pre-processed and the datasets for each vulnerability have built by taking every single code token with its context. VUDENC applies a word2vec model to identify semantically similar code tokens and to provide a vector representation. A network of long-short-term memory cells (LSTM) is then used to classify vulnerable code token sequences at a fine-grained level,

highlight the specific areas in the source code that are likely to contain vulnerabilities, and provide confidence levels for its predictions. VUDENC has achieved a recall of 78%-87%, a precision of 82-96% and an F1 score of 80-90%. But there are few limitations of this research. First, they have conducted this for vulnerability detection only, and not for secure coding guideline violations. Thus, this research anyway doesn't support detecting custom rules. Even extending is hard, due to the long costly process such as mining GitHub repositories, labelling, preprocess, training, etc.

2.3.4 Just-in-time software vulnerability detection: Are we there yet?

Lomio, et al. (Lomio, et al., 2022) has conducted a research study to investigate how currently available machine learning-based vulnerability detection mechanisms can support developers in the detection of vulnerabilities at the commit level. They have performed an empirical study and have considered nine projects accounting for 8991 commits and experiment with eight machine learners built using process, product, and textual metrics. By their study, they have pointed out three main findings:

I. Basic machine learners rarely perform well.
II. The use of ensemble machine learning algorithms based on boosting can substantially improve the performance.
III. The combination of more metrics does not necessarily improve the classification capabilities.

Thus, they have concluded that further research should focus on just-in-time vulnerability detection, especially with respect to the introduction of smart approaches for feature selection and training strategies.

2.3.5 VulCNN: An Image-inspired Scalable Vulnerability Detection System

Wu, et al. (Wu, et al., July 5, 2022) have conducted a deep learning-based research study aiming both scalability and accuracy on scanning large-scale source code vulnerabilities. They have used an existing deep learning-based image classification approach to achieve this. They have proposed a novel idea that can efficiently convert the source code of a function into an image while preserving the program details. Then they have used it to implement a scalable graph-based vulnerability detection system called VulCNN (An Image-inspired Scalable Vulnerability Detection System). To evaluate, they have used large variety of functions and have compared results of it with different other vulnerability detectors. But this is again for vulnerability detection, and hard to extend this for detect violations of custom rules.

2.3.6 Secure Application Development

Alwan & Andersson (Alwan & Andersson, 2022) has conducted a research study and built a plugin for Java IDEs to perform an advanced analysis of security flaws. First, researchers have proposed a new algorithm to find the root cause of security violations. Reax is a command-line application, which controls information flow in Java programs, predicts information flow violations, and applies suitable countermeasures to prevent violations (Khakpour, et al., 2018). But it is hard to be used by developers, due to its command-line approach. Researchers have proposed a method to simplify the results of Reax and have developed a graphical plugin by integrating Reax and the algorithm in the development environment. Thus, this plugin performs advanced security analysis that detects and reacts directly to security flaws, by simplifying the results of Reax. As a result of this plugin, developers who use this plugin will be able to detect security violations and fix their code during the implementation phase. But this research has a few limitations. First this research study limits to detect security violations in software security flaws. Thus, not detecting secure coding guideline violations. Further, they don't support adhering to secure coding guidelines and thus this research study doesn't have a mechanism to convert user dependent secure coding guidelines into algorithms.

2.3.7 Framework for Secure Coding

Dasanayake et al. (Dasanayake, et al., 2019) has conducted a research study, to detect real-time secure coding guideline violations in the Java programming language. The plugin provides real time detection with IDE integration. The plugin also provides relevant countermeasures for the detected corresponding secure coding rule violations. This plugin automates 15 main violations of the "SEI CERT Secure Coding Rules" cheat sheet. But the plugin has a few drawbacks. The major drawback of this plugin is the user can't extend the framework to support their own rules. The user must use the plugin with the provided set of rules, instead of writing rules. Thus, finding a formal mechanism to input user-dependent rules and a formal mechanism to check the rule with secure coding guideline are highly needed.

Supported Secure coding guidelines				
NUM09-J	ERR07-J	MET09-J	OBJ10-J	SER01-J
NUM10-J	ERR08-J	OBJ05-J	DCL00-J	SEC07-J
ERR04-J	EXP02-J	OBJ01-J	THI00-J	FIO02-J

Table 1: Supported Secure coding guidelines of SEI CERT (Dasanayake, et al., 2019)

2.3.8 Sensei

Cremer, et al. (Cremer, et al., 2020) have conducted a research study called ‘Sensei’ to enforce secure coding guidelines in the IDE (Secure Code Warrior, 2000-2022). They have developed a plugin with a new rule generation approach. They have provided support for fixing the rule violation, but the rule writer must define the code fragment that needs to be put in place.

```
1  public void executeCommand(String command){
2      Runtime r = Runtime.getRuntime();
3      r.exec(command);
4  }
```

Listing 1: Insecure usage of Runtime.exec

```
1  public void executeCommand(){
2      Runtime r = Runtime.getRuntime();
3      r.exec("explorer.exe");
4  }
```

Listing 2: First secure usage of Runtime.exec.

```
1  public void executeCommand(){
2      Runtime r = Runtime.getRuntime();
3      String command = getSafeCommand();
4      r.exec(command);
5  }
```

Listing 3: Second secure usage of Runtime.exec.

```
1  search:
2   methodcall:
3    name: "exec"
4    type: "java.lang.Runtime"
5    args:
6     1:
7      type: "java.lang.String"
8      containsUntrustedInput: true
9      trustedSources:
10     - methodcall:
11       name: "getSafeCommand"
```

Listing 4: Rule trigger to prevent OS command injection.

Figure 2: Sensei rule to detect insecure usage of Runtime.exec

They used a model-based rule creation method at the first stage of the plugin development. It lets the user create new rules using predefined rule models, by filling in several fields. But according to them, for more complex models the number of input fields grew rapidly to accommodate a plethora of corner cases, and so did the number of models for multiple scenarios. Thus, as per them, the model-based rule creation process is not flexible now because currently there are over 40 different models. Thus, this concludes that the model-based rule creation process is not flexible and intuitive enough.

They have used a creation-based rule-generation approach at the second stage of Sensei and they have used YAML Ain't Markup Language as the language for the rule creation part.

Researchers have achieved writing rules to detect method level violations and class level violations. But this research study doesn't provide a way to create secure coding guidelines for package level code. This is a major drawback of this research study, and this is a considerable gap that needs to be addressed in future research studies. Further, even though they have achieved writing simple rules, there is a gap that needs to be addressed when the secure coding guidelines become complex.

The rule creation language, YAML is a data serialization language by its design, and uses indentation to define structure (Eriksson & Hallberg, 2011) (Ben-Kiki, et al., 2009) (YAML, 2001-2009) (Ben-Kiki, et al., 2005). Here, YAML (YAML, 2009-2022) (YAML, n.d.) (YAML, 2001-2009) has few major drawbacks.

YAML language does not support representing executable code or logic directly. YAML itself doesn't have built-in features for conditional statements like programming languages (YAML Org, 2021). Further, YAML itself doesn't have built-in "get" and "set" functionalities like traditional programming languages (YAML Org, 2021). These two are the main reasons for the limitations of this research study. Furthermore, the language does not have any built-in support for wildcards (Ben-Kiki, et al., 2009) (YAML, 2009-2022). These are major drawbacks which limit the rule creation functionality.

```
search:
  reference:
    name: "s"
```

```
private String s = "";

void doSomething() {
    System.out.println(this.s);
}
```

Figure 3: Sensei YAML rule: checks the name of the reference (Secure Code Warrior, 2019-2021)

```yaml
search:
  parameter:
    type: "String"
```

```
public void userFunction(
    String input,
    Object... args,
) {
    // ...
}
```

Figure 4: Sensei YAML rule: checks the type of the parameter (Secure Code Warrior, 2019-2021)

```yaml
search:
  instanceof:
    checkType: "MyClass"
```

```
void myMethod(Object o) {
    if (o instanceof MyClass) {

    }
    if (o instanceof MyOtherClass) {

    }
}
```

Figure 5: Sensei YAML rule: checks the type of the class that is being checked inside the 'instanceof' comparison (Secure Code Warrior, 2019-2021)

```yaml
search:
  localVariable:
    type: java.lang.String
```

```
void x() {
    String text = "hi";
    int y = 0;
}
```

Figure 6: Sensei YAML rule: checks the declaration type of the variable

```
search:
  throw:
    value:
      reference:
        name: "e"
```

```
Exception e = new RuntimeException();
throw e;
```

Figure 7: Sensei YAML rule: checks the value that is being thrown (Secure Code Warrior, 2019-2021)

```
search:
  throw:
    type: "RuntimeException"
```

```
if (!this.isValid()) {
    throw new RuntimeException();
}
throw new CustomException();
```

Figure 8: Sensei YAML rule: checks the type of the value that is being thrown (Secure Code Warrior, 2019-2021)

For the complex data set, the YAML output is larger than other languages, due to the amount of whitespace needed (Eriksson & Hallberg, 2011). Thus, relying on indentation can be error-prone and difficult to maintain, especially for large or complex rules. Thus, users must define basics in a complex manner.

2.3.9 Conclusion

Therefore, to provide a rule customization which supports package level rule creation and complex rule creations, finding a proper language is necessary and that is the major research gap that needs to be addressed.

2.4 Current approaches – Taxonomies and limitations

2.4.1 Existing tools and research studies: overall limitations

2.4.1.1 Commercial static analysis tools: limitations

Related work	Limitation

SpotBugs	– Tool can detect relevant *bugs* in source code but not *secure coding rule violations*. – The *rule creation provided through third party "detectors"*. But these must be implemented through an *API*. Thus, it is not that convenient. – Tool provides IDE integration, but; it is mostly used to *scan after development*, and it takes time up to 20 minutes.
SonarQube	– The tool can detect supported secure coding rule violations. But these are generating as *reports*, not as on-the-fly feedback. – Tool provides *custom rule generations* for vulnerabilities but using provided *templates* only. Adding desired rule is not supported.
SonarLint	– The tool cannot detect secure coding rule violations. – On-the-fly feedback is provided only when the Java class is *saved* not while the user types.
Fortify Static Code Analyzer (FSCA)	– Provides over 1000 violation detections and custom rule generations, but *the tool takes few minutes to scan source code and identify violations*. Therefore, *Can't* provide *on-the-fly feedback*.
Tricorder	– Data-driven program analysis platform integrated into the workflow of developers at Google. *Rule creation* is provided through *programming languages*. – But results of Tricorder analyzers are shown in a *review tool*. No mechanism to integrate with IDE. Doesn't provide on-the-fly IDE support.
Veracode	– Provides both a *Software as a service* (*SaaS)* platform and an *IDE* plugin. – Veracode focuses heavily on not only detecting vulnerabilities but also guiding remediation. A *security expert* can help the developer *determine* whether the reported *issue is a false positive* or what the best remediation is for the detected vulnerability. Scheduling such a consultation will usually take

	about *three days*. – *Doesn't encourage* sharing knowledge and *custom rule generation*. – The company claims most *scans* finish in under *an hour*. This means the feedback cycle is rather long compared to the other tools.
Checkmarx	– Plugins do not perform any local scans but instead allow *uploading* the source code and *scanning* it. *No on the fly feedback.* – Since the *rule-writing tool* is independent from the scanning tool and the IDE, it *requires long iterations to optimize rules* compared to the tools that provide instant feedback.
Snyk	– A tool designed to monitor and *fix insecure dependencies* and will not look for secure coding guideline violations.
The OWASP ASIDE/ESIDE	– *Scans* for secure coding guideline violations *need to be started manually*.

Table 2: Commercial static analysis tools - Ensure security in software code

2.4.1.2 Secure coding – research studies: overall limitations

Related work	Limitation
Framework for Secure Coding: An algorithmic approach for real-time detection of secure coding guideline violations	– Provides *IDE integration* and *on-the-fly feedback* but supports *a limited number of rules*. – *No* mechanism to create *custom secure coding guidelines*. Finding a *formal mechanism to input user-dependent rules* and a *way to find violations of custom rules* in the source code needed.
Sensei: Enforcing Secure Coding Guidelines in the IDE	– They have used YAML as the rule defining language. But YAML has a few major drawbacks which need to be addressed. *Addressing YAML limitations* is needed. – A mechanism to *create rules* and identify violations of *complex rules and package level secure coding guidelines* is a gap that needs a solution, due to limitations of YAML language.

Table 3: Related research studies - Ensure security in software code

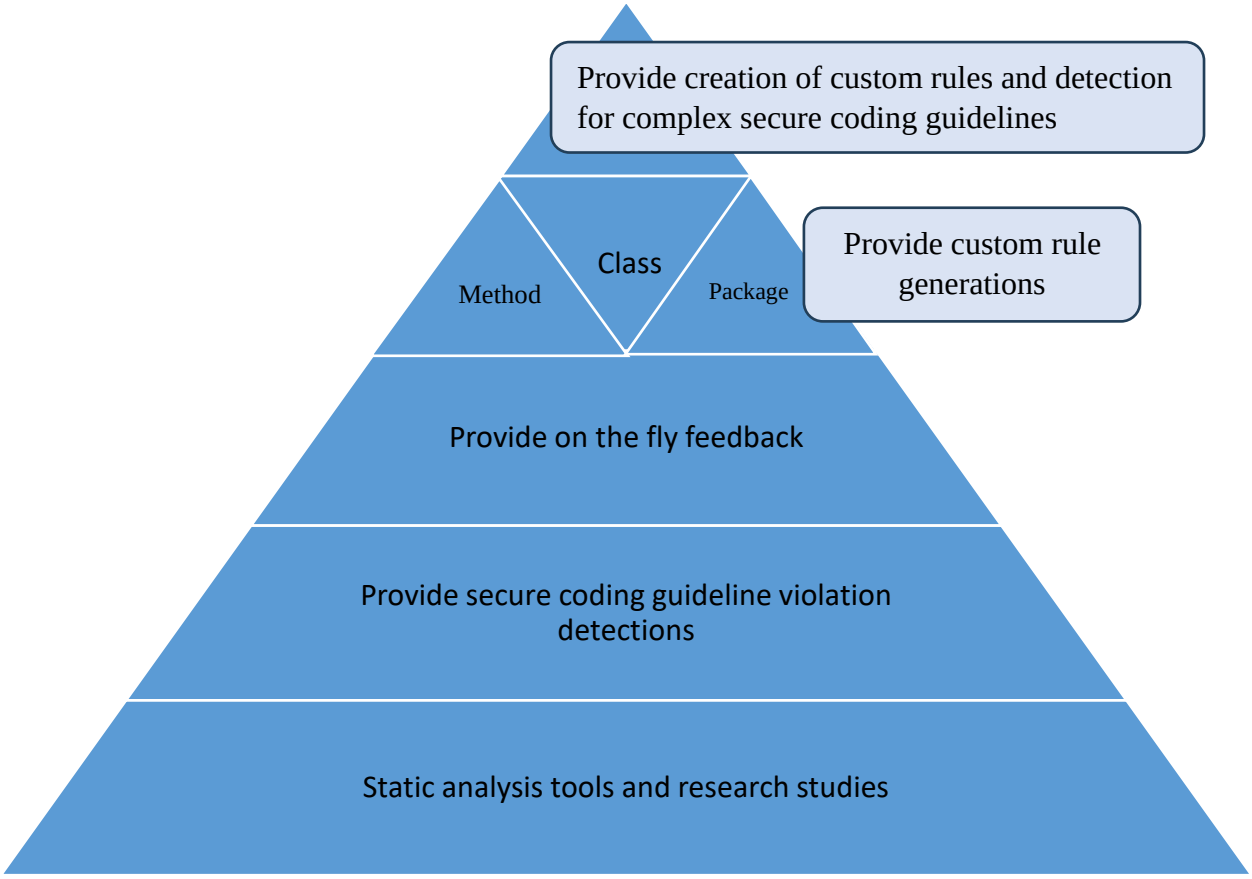


Figure 9: Taxonomy of available static analysis tools and research studies

2.4.1.3 Research gap that needs to address with compared to previous research studies:

Thus, from the above comparison, it can be concluded that tools with *real-time IDE support* are very less. Further, tools that *adhere to secure coding guidelines in real-time* are very less. Furthermore, tools that support *creating users' own rules* are also very less. Thus, with respect to the combination of these three, there is a gap to address.

Out of the above research studies, Framework for secure coding (Dasanayake, et al., 2019) and Sensei (Cremer, et al., 2020) can consider the research studies which support detection of secure coding guideline violations on the fly in the IDE. But there are limitations that needs to be addressed:

	Support method level secure coding guideline violations	Support class level secure coding guideline violations	Support package level secure coding guideline violations	Supports custom rule generations	Can create complex secure coding rules which needs to track the control flow

Framework for secure coding	Yes	Yes	Yes	**No**	**No**
Sensei	Yes	Yes	**No**	Yes	**No**

Table 4: Current research gap

Therefore, can concludes that, None of the above support:

1. Custom rule creation for **package level** secure coding guidelines and
2. Custom rule generations for **complex secure coding rules** which need to track the control flow of the source code of a project.

2.4.2 Rule creation mechanisms: limitations

Rule creation mechanism	Related work	Limitations / Conclusions of the study
Model based rule creation	Sensei – 1st stage	– Tool has provided an on-the-fly mechanism to detect vulnerability violations, with suggested solutions, but too many models to add. – This study *concludes that model-based rule creation process is not flexible* and intuitive enough.
Template based rule creation (using provided sets of models)	SonarQube	– Limited to *provided templates*. User must use templates to create the custom rules.
A trigger – way of YAML Ain't Markup Language	Sensei – 2nd stage	– Another approach: split up the rule in two parts: A trigger to identify the violation, plus an optional quick fix to correct the vulnerability consistently. – The study concludes that, *trigger provides more flexibility than 1st approach*. – Need to address *limitations of YAML*.
Writing rules through complex, well documented	Sensei – lit review (Cremer, et al., 2020)	– Most comparable tools allow writing custom rules or analyses in some way or another. Writing rules for them is done through

APIs (SpotBugs, Tricorder, Checkmarx)		complex but well documented APIs (SpotBugs, Tricorder, Checkmarx). – *None* of the tools allow *detecting violations of created rules on-the-fly in the code*.
XML	Sensei – lit review (Cremer, et al., 2020)	– *None* of the tools allow *detecting violations of created rules on-the-fly in the code*. – Sensei paper has already concluded that, *YAML is better than XML.*
Custom XML formats	SecureAssist, Fortify Static Code Analyzer (FSCA)	– *Learning* their syntax is needed. – *None* of the tools allow *creating rules and detect on-the-fly* secure code guideline violations in the code. – Sensei paper has already concluded that, *YAML is better than custom XML.*
Programming languages	Tricorder	Integrated into the *workflow*. *No mechanism to integrate with IDE*. This is a separate code review tool.

Table 5: Rule creation mechanisms

It can be concluded that, out of the above methods, the most flexible method is using markup languages. But currently used markup languages have a gap that needs to be addressed.

2.4.3 Rule comparison/vulnerability verification methods

A taxonomy of rule comparison and vulnerability verification methods are available under Appendix F: Taxonomy of Rule comparison/vulnerability verification methods.

2.5 Method, class, and package level violations

As per the literature, source code fragments that trigger violations can classify into method level, class level and package level based on the level at which they lie (Dasanayake, et al., 2019).

Method Level	Class Level	Package Level
Focuses on the source code fragments that belongs to the java.lang package (default	Focuses on the source code fragments of the java.lang package(default package) that	Focuses on the source code fragments that belongs to classes outside the existing

package) and exist inside a method of a class.	is inside a class but lies outside a method.	class.
1. Method parameters in method signature 2. Local variables 3. Loop controls (for, for each, while, do while) with no method calls 4. Exceptions belonging to the java.lang package (eg- NullPointerException) 5. Threads(That fall into java.lang package) 6. Try, catch, Finally blocks	1. Names of class variables 2. Data types of class variables 3. Access modifiers of class variables 4. Method names in a method signature 5. Return types of methods in a method signature 6. Access modifiers of methods in a method signature	1. Methods belong to packages outside java.lang package 2. Extended classes outside java.lang package 3. Library imports 4. Implemented interfaces which are outside java.lang package 5. Instances of classes outside java.lang package

Table 6: Classification of method level, class level and package level violations (Dasanayake, et al., 2019)

2.5.1 Method, class, and package level secure coding guidelines: examples

Examples for secure coding guidelines of these three categories are available under Appendix G: Method, class, and package level secure coding guidelines: examples

2.6 Nature of package level and complex secure coding guidelines

There are special characteristics a rule defining language must have, to be able to define package level secure coding guidelines and complex secure coding guidelines.

2.6.1 Nature of package level secure coding guideline violation

Below is an example of package level secure coding guideline violation.

THI00-J. Do not invoke Thread.run()

This is a secure coding guideline from the SEI CERT list (Carnegie Mellon University - Software Engineering Institute, 2023). This rule simply says, a programmer should avoid invoking a Thread object's run() method. In the below example code, ‘Foo’ class is a custom class, and that implements the ‘Runnable’ interface. Therefore, ‘Foo’ can be run as a thread.

But according to the guideline, a developer should not directly invoke the run method of a thread.

To determine a violation of this kind of rule, first we should be able to define this rule using the secure coding guideline rule defining language. To accomplish that rule defining language should be able to provide few features.

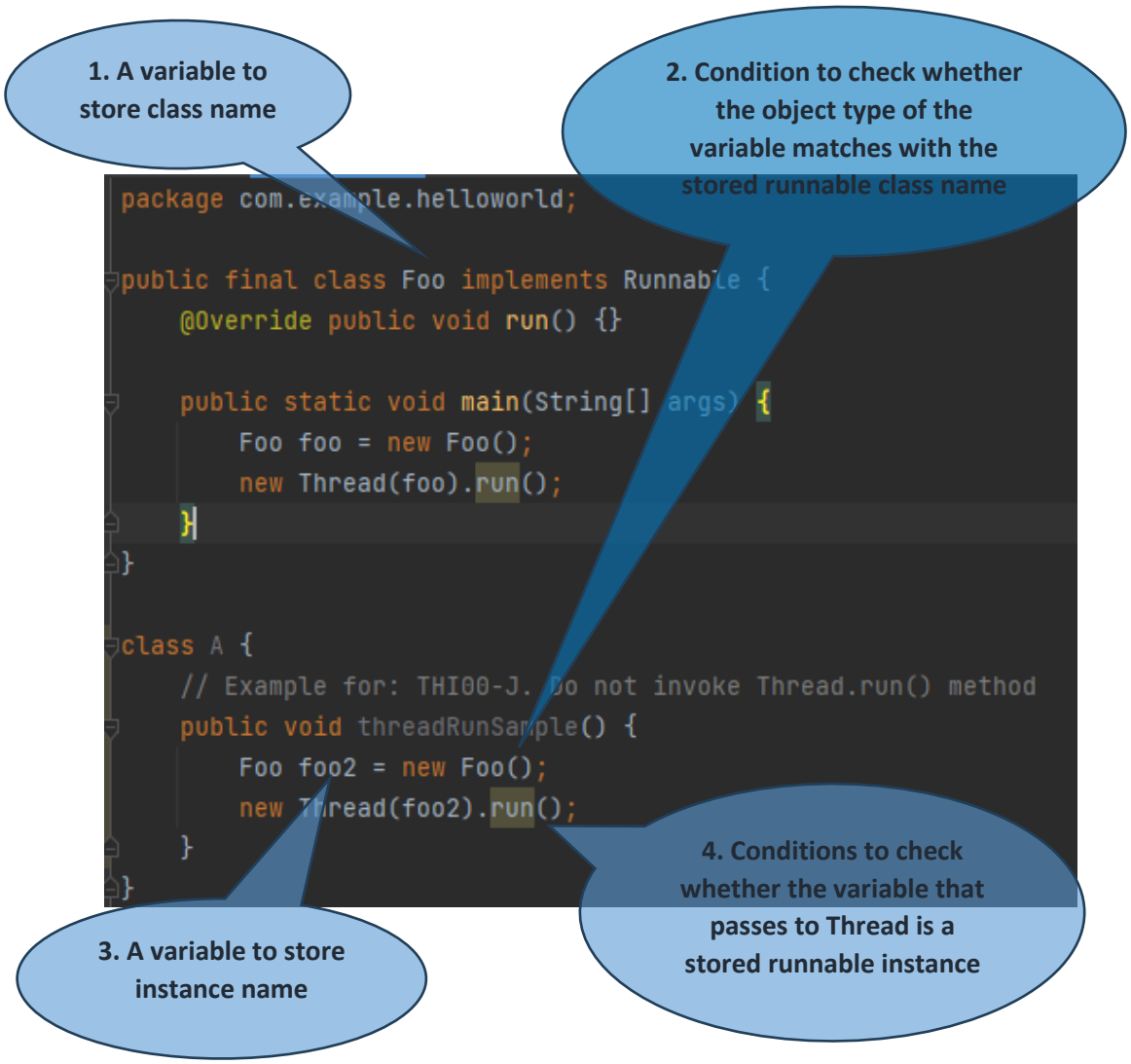


Figure 10: Nature of package level secure coding guideline violation

1. To detect this kind of violation, a secure coding rule defining language first should be able to track the custom class name. Therefore, the secure coding rule defining method should be able to **store variables/ should support getters and setters** in order to save custom data.
2. Then, to detect an instance of the 'Foo' class, when initializing any variable, the rule defining language should be able to say; compare the stored class type with the object type of the new variable. If matches, can say this is an object of the 'Foo' class.

Therefore, the secure coding rule defining method should have a way to create **conditionals**.

3. After identifying the ‘Foo’ instance, there should be a way to store the instance name of the variable. Thus, again, the secure coding rule defining method should be able to **create and assign variables.**
4. Then after detecting a run() invoke, the rule creation mechanism should be able to compare the variable that passes to Thread is a stored runnable instance. To support this also, again we need **conditionals**.

2.6.2 Nature of a complex secure coding guideline

Below is an example of a complex secure coding guideline violation, which needs to track control flow of the code execution.

IDS01-J. Normalize strings before validating them;

This is a secure coding guideline from the SEI CERT list (Carnegie Mellon University - Software Engineering Institute, 2023). This rule simply says, Applications that accept untrusted input should normalize the input before validating it. Consider the below example. Here, assume ‘strInput’ is a string variable from a user input. To prevent cross site scripting, a developer should validate the input string for script tags. To accomplish this, a developer might check for ‘<>’ parenthesis. But in the below example, parenthesis is in Unicode form. Normalization is important because in Unicode, the same string can have many different representations. When implementations keep strings in a normalized form, they can be assured that equivalent strings have a unique binary representation. Therefore, the secure coding guideline says, a developer should normalize a string before validating it.

1. To accomplish this, first, the rule creation language should be able to keep track of the string variables (‘strInput’ in the below code example). Therefore, a rule creation language should have support for **store variables/ should support getters and setters**. Further, we might need **another set of variables to keep track of the execution order of the code**. Because in the below example, even though the ‘strInput’ is normalized, it is normalized after validating it. Therefore, tracking code execution order is very important. Checking for a normalization is not just enough.
2. A **variable** in the rule creation language to track whether the ‘strInput’ is normalized might needed. At the beginning, the value of it might be ‘false’.
3. Then the rule might need to know the pattern instance name, and therefore, another

variable needed in the rule creation language.

4. There might be a lot of string variables in a program inside even a method. As an example, can take the ‘str1’ variable in the below code. **Condition** to check whether the name of the variable inside normalization matches with tracked variable name of the ‘strInput’ might needed in the rule creation language.
5. This step is also very important in the rule creation. Even though ‘strInput’ is normalized, it might be assigned to another new variable. Therefore, the rule must check for the new variable name after the normalization, in the validation step. If a developer uses a new variable to store the normalized string, and if the developer accidentally validates the older string, that is again incorrect. Therefore, rule creation language might need **variables/ getters and setters** again to keep track of this.
6. Final step is:
    a. Check whether the input string is normalized (A **condition** needed, to compare the stored track value is ‘true’, and a **getter** needed in the rule creation language to access the stored variable value)
    b. Check whether the string variable name inside the ‘matcher’ matches the string variable name after the normalization (A **condition** needed, to compare the stored string variable name after the normalization matches the string variable name inside the ‘matcher’. A getter needed to access the stored variable value after the normalization)
    c. Another **condition** is needed, to check whether the execution of matcher has happened after the string normalization. Here again, the rule creation language might use **getters** of variables that are used to track the execution order of the code.
7. If everything matches for a violation, then the rule creation language might **output** the violation description, in order to display in the IDE.

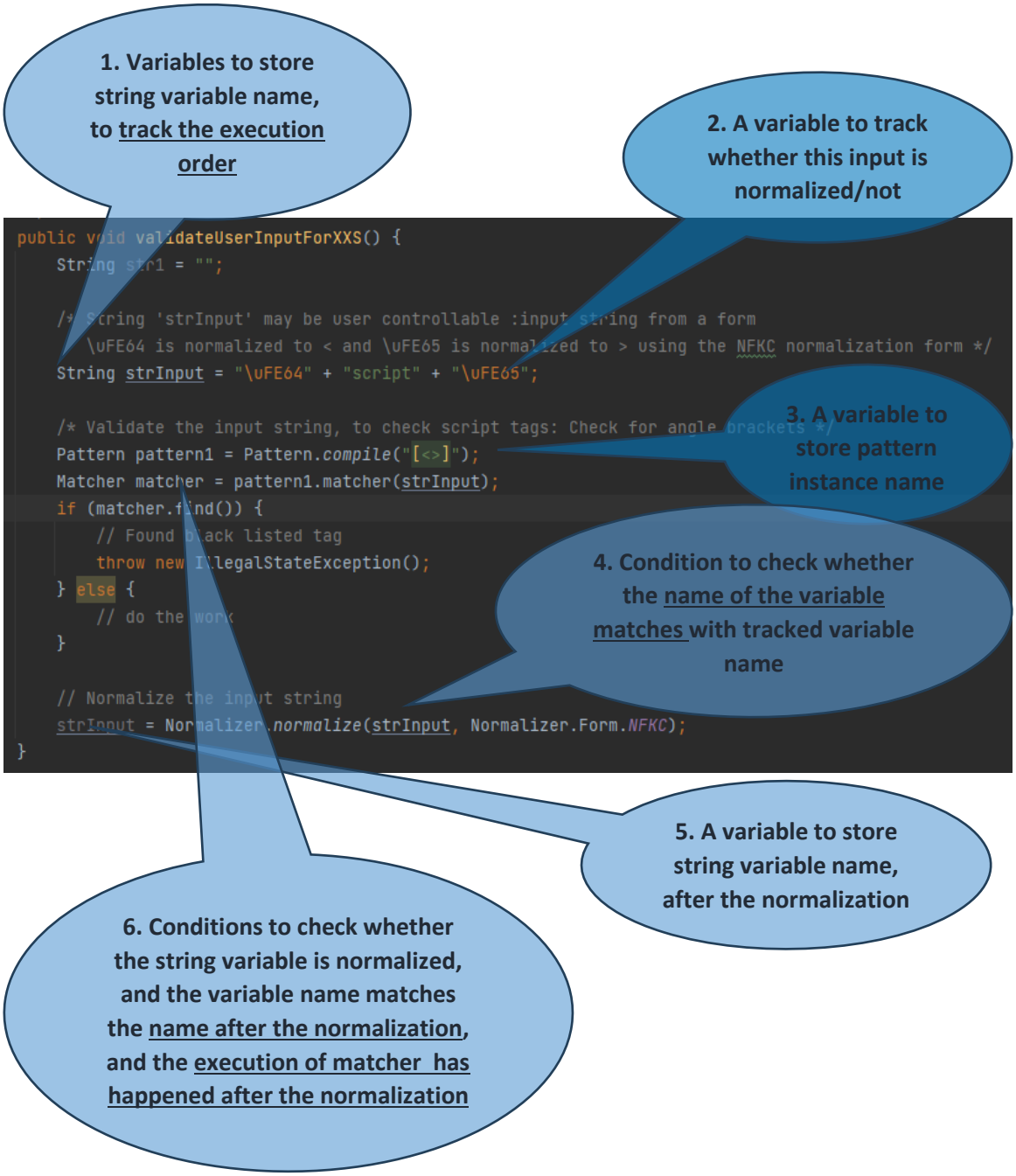


Figure 11: Nature of complex secure coding guideline violation

2.6.3 Conclusion

Therefore, can conclude that a rule creation language must be able to *keep track of the execution order of the input source code*, and that is an important feature for a rule creation language. Therefore, to achieve this, can conclude that *support for conditionals* and *support for variables* in the rule creation language are must features for a rule creation language.

2.7 XML and YAML Limitations when defining complex and package level secure coding guidelines

XML and YAML are the currently used languages for the rule creation part. ‘Sensei’ paper has already concluded that YAML is better than XML. But both YAML and XAL have a few drawbacks and due to the below mentioned drawbacks, both YAML and XML cannot specify *package level secure coding guidelines* and *complex secure coding guidelines, which needs to maintain the track of the control flow of the system*. Therefore, existing solutions are not capable of defining package level violations and complex rules which need to track the control flow of the source code.

2.7.1 Dynamic Code or Logic

2.7.1.1 YAML

YAML is primarily for data serialization and doesn't support representing executable code or logic directly. YAML itself doesn't have built-in features for true conditional logic like if-else statements (YAML Org, 2021) (YAML, 2009-2022). While some extensions might allow code snippets, it's not designed for complex programming tasks.

2.7.1.2 XML

Standard XML itself doesn't directly support conditional blocks or dynamic code in the way programming languages do (W3C, 2013).

2.7.2 Storing values in variables

2.7.2.1 YAML

YAML is primarily for data serialization and YAML itself doesn't have built-in "get" and "set" functionalities like traditional programming languages (YAML Org, 2021) (YAML, 2009-2022).

2.7.2.2 XML

XML itself doesn't inherently offer built-in functionalities like "get" and "set" for directly manipulating data within the document (W3C, 2013). To achieve this, external processing and libraries are needed.

2.7.3 Wildcards

2.7.3.1 YAML wildcards:

YAML itself doesn't have built-in features for wildcards and therefore YAML doesn’t support wildcards (YAML Org, 2021) (YAML, 2009-2022).

2.7.3.2 XML wildcards:

XML supports 2 main types of wildcards (W3C, 2013).

Asterisk (*): This wildcard matches any sequence of characters, including zero characters. It can be used in element names, attribute names, and attribute values.

Question mark (?): This wildcard matches any single character. It can be used in element names, attribute names, and attribute values.

2.8 Custom rule defining: Markup languages and other languages

‘Sensei’ research paper already concludes that using a trigger is more scalable than previous approaches (Cremer, et al., 2020). Further, they conclude that using YAML over XML and custom XML formats is better. But YAML has limitations that need to be addressed.

YAML (YAML, 2009-2022) (YAML, n.d.) (YAML, 2001-2009) is a data serialization language by its design (Ben-Kiki, et al., 2009) (Ben-Kiki, et al., 2005). YAML is known to be simple in terms of human readability, due to its limited data types. Thus, YAML produces the most compact output for the simple data set.

The biggest disadvantages of YAML in the rule creation context are, the language does not have any built-in support for conditionals, storing variables and wildcards (Ben-Kiki, et al., 2009) (YAML, 2009-2022). Thus, users must define basics in a complex manner. This makes the rule-generation process harder and more time-consuming. This adds extra complexity to the work of a secure rule creator. Even researchers that have used YAML for other domains in computer science have pointed out this wild card issue (McGhee, et al., July 2022, p. 7). Further, as explained earlier, there is a major gap that cannot be filled using YAML. Therefore, the secure coding rule defining domain needs a novel solution to this problem.

In YAML block styles, it uses indentation to define structure (Eriksson & Hallberg, 2011)

(Ben-Kiki, et al., 2009, p. 30) (YAML, 2001-2009) (Ben-Kiki, et al., 2005). Indentation is zero or more space characters at the start of a line. As the document hierarchy gets more complex, with deeper nesting being added, the amount of whitespace needed for YAML to correctly indent everything grows noticeably (Eriksson & Hallberg, 2011). Thus, for the complex data set, the YAML output is larger than other languages (Eriksson & Hallberg, 2011). Thus, relying on indentation can be error-prone and difficult to maintain, especially for large or complex rules. Thus, using indentation makes the rule buggy if the rule writer uses it in an incorrect way. Thus, this will result in incorrect secure coding violation detections. Further, due to this indentation problem, YAML files can be hard to edit, especially for large files. Accidentally getting the indentation wrong often isn't an error for YAML parsers; it will often just deserialize to something you didn't intend. Thus, in terms of the rule defining language context, there is a gap that needs to be addressed.

Since the introduction of YAML, remote code execution vulnerabilities have been reported for YAML parsers (Rasheed, et al., 2019). Thus, YAML and YAML parsers are known to be insecure. Even most libraries are known to be unsafe by default. Several remote code execution vulnerabilities in YAML parsing libraries and deserializing YAML data have been reported since 2013 (Rasheed, et al., 2019).

2.8.1 Rule Markup Language (RML)

RML is a general-purpose data mapping language, and it is used for rule-based transformations of XML (Jacob, 2006). RML Focused on representing and managing rules in general, not necessarily web specific. With RML the user can define XML wildcard elements. Further, it supports defining variables containing parts of the XML such as variables for element names or variables for lists of elements (Jacob, 2006).
RML can be used to specify semantics for state charts and class–diagrams in UML models.
Mostly, RML is only used to define transformation rules, while the input and output of a transformation are pure problem domain XML. The RML tools are available as platform-independent command–line tools. They can be used together with other tools that have XML as input and output (Jacob, 2008). RML re-uses the problem domain XML, extended with only a few constructs (Figure 12)

Elements that designate rules				
div	class="rule"			
div	class="antecedent" context="yes"			
div	class="consequence"			
element	attribute	A	C	meaning
Elements that match elements or lists of elements				
rml-tree	name="X"	*		Bind 1 element at this position to RML variable X.
rml-text	name="X"	*		Bind XML text-content to variable X.
rml-list	name="X"	*		Bind a sequence of elements to X.
rml-use	name="X"		*	Output the contents of the RML variable X at this position.
Matching element names or attribute values				
rml-X	...	*		Bind element name to RML variable X.
rml-X	...		*	Use variable X as element name.
...	...="rml-X"	*		Bind attribute value to X.
...	...="rml-X"		*	Use X as attribute value.
...	rml-others="X"	*		Bind *all* attributes that are not already bound to X.
...	rml-others="X"		*	Use X to output attributes.
...	rml-type="or"	*		If this element does not match, try the next element in the antecedent if that also has rml-type="or".
Elements that add constraints				
rml-if	child="X"	*		Match if X is already bound to 1 element, and occurs *somewhere* in the current sequence of elements.
rml-if	nochild="X"	*		Match if X does not occur in the current sequence.
rml-if	last="true"	*		Match if the preceding sibling of this element is the last in the current sequence.
A * in the A column means the construct can appear in a rule antecedent. A * in the C column is for the consequence.				

Figure 12: All the RML constructs (Jacob, 2008, p. 40)

The execution of a rule consists of binding variables in the matching process, and then using these variables to produce the output. The part of the input that matches the rule antecedent is replaced by the consequent of the rule (Jacob, 2008). This is because this language is for logical foundations. Theoretically in our context, this is not what we need. Therefore, cannot use RML.

2.8.2 Rule Markup Language for the Web (RuleML)

RuleML expresses the rules in XML syntax for the Web (RuleML, 2023) (Mehla & Jain, 2019). It can be used to transform structured data into RDF (Resource Description Framework) triples. RDF is a standard for representing information on the web, enabling you

to describe resources and their relationships in a machine-readable format. RML re-uses the problem domain XML, extended with only a few constructs (Figure 12) to define rules, whereas RuleML superimposes a special XML vocabulary for rules (Jacob, 2008). This makes the RuleML approach complex and thus difficult to use in certain cases. The idea of using wildcard elements for XML has not been incorporated as such in the RuleML approach (Jacob, 2008). This language is also for logical foundations. Theoretically in our context, this is not what we need. Therefore, cannot use RuleML.

2.8.3 SCPL - A markup language for source code patterns localization

Silva, et al. have introduced a Source Code Pattern Language (SCPL), a pattern-finding language which uses markups in code examples to facilitate development of custom static analysis rules (Silva & Mendonça, 2021). SCPL provides a rich feature set and facilitates the programming of custom static analysis rules by using markups directly in source code. But this is still at the initial stage and still hasn't been used in practical applications by other researchers in the computer science domain.

2.8.4 Artificial intelligence markup language (AIML)

AIML is a markup language, that is widely used for chatterbots (Khin & Soe, 2020) (Satu, et al., 2015). It has a more powerful pattern matching language, a built-in NLP engine, and a larger community of developers. Further, this is a markup language that is widely used in the pattern recognition area (Marietto, et al., 2013) (Khin & Soe, 2020 February). AIML combines the technical and theoretical pattern recognition infrastructure in its development (Wallace, 2003) (Marietto, et al., 2013).

Ease of implementation, since AIML is an XML-based markup language and it is tag-based (Marietto, et al., 2013). Thus, when it comes to rule defining, we can use human readable tags and patterns. This tag-based syntax allows to define complex rules and patterns. Thus, AIML is more expressive than YAML. By its design, AIML is designed to support wildcards as well (Khin & Soe, 2020 February) (Marietto, et al., 2013) (Wallace, 2003). AIML is a rich language, is proven for using for complex chatbots, and thus, can be used to define complex rules as well.

AIML is more modular than YAML. Further, when referring AIML documentation, can see very rich features like conditions, loops, predicates and support for getters and setters (variables) when compared to YAML (AIML Foundation, 2018) (Wallace, 2009) (Wallace, 2003) (YAML Org, 2021). Therefore, AIML is a good candidate for filling the gap in the rule

defining in secure coding guideline violation detection domain. Having support for modules means can organize coding guidelines into reusable modules and this can be helpful when defining large coding guidelines where you want to avoid repetition. Further, using a rich language to define rules makes the process easy and smooth.

```
<category>
<pattern>WHO IS HE</pattern>
<template><srai>WHOISHE <get name="he"/></srai></template>
</category>

<category>
<pattern>WHOISHE *</pattern>
<template>He is <get name="he"/>.</template>
</category>

<category>
<pattern>WHOISHE UNKNOWN</pattern>
<template>I don't know who he is.</template>
</category>
```

```
<think>
  <set name="it">
    <set name="topic">
      PIZZA
    </set>
  </set>
</think>
```

Figure 13: AIML creating and accessing variables with <set> and <get> tags (Wallace, 2003)

```
1  <category>
2     <pattern> HOW ARE YOU? </pattern>
3     <template>
4        <condition name="state" value="happy">
5           It is nice being happy.
6        </condition>
7        <condition name="state" value="sad">
8           Being sad is not nice.
9        </condition>
10    </template>
11 </category>
```

Figure 14: <condition> Tag Usage Example in AIML (Marietto, et al., 2013)

```
1  <category>
2     <pattern> I LIKE * </pattern>
3     <template>
4        I like <star/> too.
5     </template>
6  </category>
7
8  <category>
9     <pattern> A * IS A * </pattern>
10    <template>
11       When a <star index="1"/> is not a <star index="2"/>?
12    </template>
13 </category>
```

Figure 15: Wildcards and <star> Tag example in AIML (Marietto, et al., 2013)

2.8.5 Conclusion

Out of the above three methods, AIML is the method previous researchers have used in pattern matching domain (Khin & Soe, 2020 February) (Ahmed & Singh, 2015). Further, it

supports conditionals, variables and wildcards by its design (Khin & Soe, 2020 February) (Marietto, et al., 2013) (Wallace, 2009) (Wallace, 2003). Thus, this fills the existing gap of the rule creation mechanism. Further by its design, it is rich when compared to other languages. This makes the rule creation process easy and neat. Thus, it can easily conclude that, out of the above methods, AIML is the best method. Thus, AIML is selected as the rule creation method.

2.9 AIML (Artificial Intelligence Markup Language)

AIML is a custom form of XML. The most important parts of an AIML document are (Ahmed & Singh, 2015) (Wallace, 2009) (Wallace, 2003):

a) <aiml>
b) <topic>
c) <category>
d) <pattern>
e) <template>
f) <condition>
g) <think>
h) <set>
i) <get>

There are many others tags in AIML which are used to describe a scene. The main objective of a tag is to simplify pattern matching (Ahmed & Singh, 2015). AIML consists of AIML objects. These AIML objects consist of topics and categories, which contain either parsed or unparsed data from their information is extracted (Ahmed & Singh, 2015). Data consists of characters and this character data is parsed by the AIML parser or interpreter. An AIML interpreter is one who scan AIML objects and provides a response according to those characters (Ahmed & Singh, 2015). An interpreter is the biggest part of an application. In this research also, the interpreter plays a major role.

2.9.1 AIML Objects

Below are main AIML objects (Ahmed & Singh, 2015) (Wallace, 2003).

1. AIML Root: Initiation of the AIML document.
   <aiml>….</aiml>

2. AIML TOPIC: First level optional item that specifies category of an elements. It has the attribute ‘name’ to indicate the topic name.

<aiml:topic name='topic name'>…</aiml:topic>

3. AIML Category: First level if topic not present. Else this is the second level element. This contains only one pattern and only one template. This doesn't contain attributes.
   <aiml:category>
   ……Content for category…
   </aiml:category>

4. AIML Pattern: This is an element which contains mixture of a pattern. A pattern should appear inside a category.
   <aiml:pattern>
   …..Pattern…..
   </aiml:pattern>

5. AIML Template: This is an element which goes inside a category element.
   <aiml:template>
   …..Template…
   </aiml:template>

6. Star: Indicates that AIML should replace value by a particular wildcard
   <aiml:star index='integer value to be stored'/>

7. That: Tell AIML to replace the previously produced output.
   <aiml:that index='single integer| comma separated integer value'/>

8. Set: Set a variable, and the variable name is specified by the 'name' attribute
   <set name="name_of_the_variable">…. Value to set ….</set>

9. Get: Get the value of a variable. The variable name is specified by the 'name' attribute.
   <get name = " name_of_the_variable "/>

10. Think: Tell AIML to process inside this, without generating output
    <think>
        …anything that needs to process. Ex: process and store variable…
    </think>

11. Condition: Condition to check a match, with 2 attributes 'name' and 'value'
    <condition name=" name_of_the_variable " value="value_to_match">
        … work to do, if the value to match matches with the value of the variable specified by the 'name' attribute …
    </condition>

12. Li: List item, with a value to match
    <li value=" value_to_match ">
        … work to do if the parent component value matches with the value to match …

13. AIML wildcards
    a. Asterisk (*): Matches any sequence of words, including zero words.

b. Underscore (_): Matches a single word.

2.9.2 AIML wildcards

2.9.2.1 Types of Wildcards:

Below are types of wildcards in AIML (batiaev, 2019).

1. Asterisk (*)
   Matches any one or more sequence of words. This is the most common wildcard in AIML.

2. Caret (^)
   Matches any sequence of words (including zero words). This is the most common wildcard in AIML.

3. Underscore with priority (_)
   Matches any single word. At every node, the "_" has first priority, an atomic word matches second priority, and a "*" matches next priority, and "^" matches lowest priority.

   Ex: <pattern>what is the _ capital of France?</pattern>
   would match "what is the capital of France?" or "what is the economic capital of France?".

4. Hashtag (#)
   Matches zero or more characters.

2.9.2.3 Wildcard Priority:

_ has the highest priority, followed by *, then #, and then ^. This means if a pattern contains multiple wildcards, the higher-priority ones will be matched first (batiaev, 2019) (Wallace, 2003).

2.9.2.4 Wildcards and Spaces:

Wildcards does not separate text characters and spaces. To match spaces, you need to explicitly include them in your pattern (batiaev, 2019).

Examples:

^hello^: Matches any input containing "hello", regardless of surrounding characters (e.g., "hello", "helloworld", "how are you doing hello").

_world: Matches any single character followed by "world" (e.g., "hello world", "good world").

#bye: Matches "bye" or any input ending with "bye" (e.g., "bye", "goodbye", "see you bye").

2.9.2.5 Advanced Usage:

Can embed regular expressions inside AIML patterns as well.

Wildcard Sets: Enclose characters within [] to match any one of them (e.g., [hw]ello matches "hello" or "wello").

Negation: Use ! before a wildcard to match anything except that character (e.g., !_ot matches any word except "bot").

2.10 Available AIML interpreters

As we have already decided to go with AIML, this section focuses on previous methods researchers used to generate interpreters for AIML language.

Since AIML is an XML dialect, it is not a programming language, it needs to be interpreted or parsed to be of any computational use. The interpreter must guarantee the compliance of properly formed AIML documents, perform all the necessary pre-processing duties for the correct usage of the bot and ensure the correctness of both pattern matchings of users' source code and bot response (Malvisi, 2014).

Officially recognized interpreters have been developed using following programming languages (Malvisi, 2014):

a) Lisp
b) Java
c) PHP
d) Ruby
e) Perl
f) Pascal
g) NET
h) C++
i) SETL

The AIML interpreter must implement a set of tasks to successfully implement the AIML specification correctly (Malvisi, 2014):

1. Pre-processing: load optional files if there are any (substitution file, predicates file, sentence splitting tokens). This is mostly for the chatbot domain, so that all incoming

inputs get translated according to it before pattern matching is initialized.

2. AIML file parsing: AIML files are loaded, categories might be translated into a data structure to be easily readable by the interpreter.
3. Pattern matching: the interpreter needs to match incoming input to the loaded AIML categories and provide the result according to template elements.

To develop an interpreter which works for the source code domain, it is important to study existing interpreters even in other domains as well.

2.10.1 Current interpreter usage of AIML in other domains

2.10.1.1 AIML interpreters written for ALICE

The concept of chatterbot came into existence with A.L.I.C.E. (Artificial Linguistic Internet Computer Entity). ALICE is a natural language processing chatterbot, the first chatterbot that used AIML (Google, 2014) (Anon., 2011) (Wallace, 2009) (Wallace, 2003). It is used to receive questions from users. It was based on pattern recognition. That pattern-matching algorithm was simple as a string-matching technique. To parse and declaration of different kinds of sentences easily, AIML has been used in this model (Ahmed & Singh, 2015) (Wallace, 2003). ALICE takes the text as input and produces output as text. It acts like a question-and-answer-based system. There are several AIML interpreters written with the invention of ALICE.

2.10.1.2 Program AB

Program AB is a free and open-source reference interpreter for AIML, and this is written in Java. This is widely used with chatterbots that uses AIML as the language (Google, 2013). It serves as a reference implementation of the AIML 2.0 specification. Program AB contains tree-walking interpreter and AIML source files for a chatbot ALICE 2.0 (Barnisin, 2022).

The current AIML version is AIML 2.1, and this supports AIML specification 2.0 (Barnisin, 2022) (Google, 2013). According to the literature, the project does not seem to be in the active development, as the last commit was in 2014 (Barnisin, 2022).

2.10.1.3 Program Y

Program Y is a chatbot framework written in Python 3 by Keith Sterling (Barnisin, 2022) (Sterling, 2021). It contains an AIML 2.1 compliant interpreter. The interpreter is flexible and

can be extended with custom coded tags and new pattern-matching constructs (Barnisin, 2022). It also supports non-standard extensions for list processing, dynamic sets and maps and pattern matching based on regexes. This library comes with three different chatbots, and a set of extensions – additional files with topic-specific categories. This project is actively maintained, with the last recorded activity in 2021 (Sterling, 2021).

2.10.1.4 PyAIML

The original Python 2 implementation of an AIML interpreter, PyAIML, by Cort Stratton. This is compliant with AIML 1.0.1. This offers a tree-walking interpreter, a simple interface to work with the chatbot and an option to store the chatbot's configuration into a file (Barnisin, 2022). The project will not be extended to support newer versions of AIML and is no longer in development, as the last activity was recorded in 2005 (Barnisin, 2022).

2.10.1.5 pyaiml21

Michal Barnisin has built a new AIML 2.1 interpreter, pyaiml21, in Python (Barnisin, 2022) (Barnišin, 2022, p. pyaiml21 documentation). The aim of this research is to provide a short summary of AIML. This presents existing systems implementing its specification. They have analyzed the requirements for a new interpreter, written in Python, compatible with the recent AIML standards. Based on that, they have designed, implemented, evaluated, and published the new interpreter, pyaiml21. In this research, considering the AIML code might contain different non-AIML tags, the researcher has created a two-phase AIML parser, which includes XML parsing separately and AIML validation separately. According to the researcher, this is because AIML is an untyped, Turing-complete programming language. But this is written in the Python language, and this is for chatbots.

2.10.1.6 Development of a Framework for AIML Chatbots in HTML5 and JavaScript

This research has focused on the implementation of an AIML interpreter written in JavaScript to allow for web-based client-side specific usages of AIML chatbots (Malvisi, 2014). The goal of the researcher is to assure the compliance of properly formed AIML documents, perform all the necessary pre-processing duties for the correct usage for a chatbot and ensuring the correctness of both pattern matchings of user input and chatbot response.

The implemented interpreter exploits the DOM tree manipulation functions of the jQuery library to achieve above mentioned goals, treating AIML files as if they were normal XML files. This AIML interpreter supports AIML 1.0 specification.

2.11 Available secure coding guidelines and rules

During the literature review, secure coding guidelines provided by 3 parties namely OWASP, Oracle, and SEI CERT were identified (Dasanayake, et al., 2019) (Carnegie Mellon University - Software Engineering Institute, 2018) (The OWASP Foundation, 2017). According to the literature, the below are some of the parameter values to check before identifying the most feasible set of secure coding guidelines to be used for the evaluation of the proposed system along with the proposed rule generation mechanism.

Parameters	**SEI CERT** (Carnegie Mellon University - Software Engineering Institute, 2018)	**Oracle** (ORACLE, 2023)	**OWASP** (OWASP, 2010)
Number of Resources / References available	Significant	Less	Less
Code examples provided	Significant	Significant	Less
Nature (language specific/not)	Language specific	Language specific	Not language specific
Security domain Coverage (Security areas covered)	Significant coverage	Low coverage	Significant coverage

Table 7: Comparison of secure coding guidelines (Dasanayake, et al., 2019)

The aim of this research study is to provide custom secure rule generations. Based on the analysis shown in Table 7 it could be concluded that the secure coding rules provided by SEI CERT are the most suitable set of secure coding standards to be used for the evaluation process of the rule creation mechanism and the overall system. Thus, a set of secure coding guidelines from SEI CERT will be selected and secure coding guideline rules will be created to detect violations of the selected set, to show the capability of the custom rule generation mechanism and for the demonstration purposes.

2.12 Summary

There is a huge increase in remediation costs when software defects are detected and corrected at post-implementation phase of the SDLC when compared with the development or coding phase of the SDLC. Following secure coding guidelines while developing of software applications, can be considered as the well-known and accepted method to overcome this issue since these best practices primarily focus on vulnerabilities that may arise in the source code level.

The review of existing approaches gives an idea that there exists no automated mechanism to identify the violation of secure coding guidelines in the source code with a formal mechanism to convert user dependent rules into algorithms. Further, existing tools gives only a raw set of violations, and not a user-friendly dashboard. Developing a software product that can detect these violations in the form of a plugin-based framework with a user-friendly user interface could be considered as the best solution for the problem.

CHAPTER 3
METHODOLOGY

3.1 Introduction

This chapter mainly describes the proposed design of the framework that was implemented to provide a solution for the above-mentioned problem. Based on the literature review that was carried out in the background study a design methodology was identified. The system architecture and the system model were developed based on that design methodology. Several parser selection criteria were also analyzed in this phase.

This research methodology falls under a constructive research methodology. The below diagram explains the constructive research methodology type briefly. The research focuses on solving the existing real-world problem. To achieve that, have done a theoretical literature review, to understand the theoretical background. The output for the real world is a tool with the resolved gap, and the output for the theoretical domain is, how AIML works beyond the chatbot domain and the capability of it.

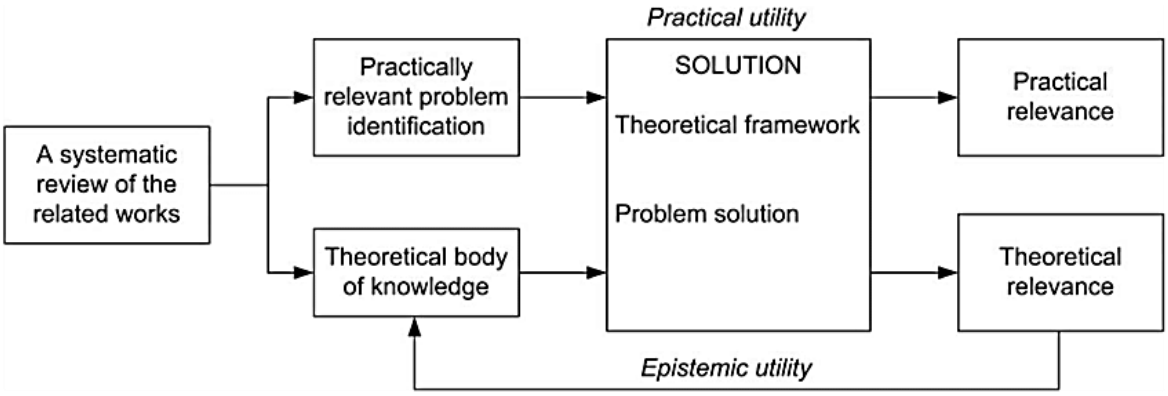


Figure 16: Diagram of a constructive research approach (Dagiene, et al., 2015)

3.2 Problem analysis

The goal of the project is building a framework to detect secure coding guideline violations in real-time along with a new secure coding rule defining method to fill the existing gap. Currently, most organizations and software project teams use cheat sheets to distribute their custom secure coding guidelines, and developers do not follow these guidelines due to time constraints or sometimes they do not understand these guidelines. To achieve the goal, an extensive background study was conducted by referring relevant artifacts such as white papers, dissertations, existing tools, etc.

The acquired knowledge from the background study was used to identify relevant requirements, design system architecture and system model, identify related components of the framework, etc. Since the solution involves creating AIML interpreter to support source code domain, previous approaches of AIML interpreter creation were studied, and created a new interpreter based on previous approaches.

Overall system architecture is proposed based identified theories to construct a solution, and then implemented as a prototype. To validate the end user source code in real-time, used a pattern matching technique. Further, to validate the proposed solution, identified set of available complex and package level secure coding guidelines, and created AIML files to represent these secure coding guidelines. Then created a sample project with violations of above selected secure coding guidelines and checked for the plugin-generated output of the end users' IDE.

3.3 Design assumptions and dependencies

1. The end users side rule creator who writes custom secure coding guidelines is responsible for writing secure coding AIML rules correctly. AIML rules should be syntactically correct, to correctly interpret rules and generate output.
2. The user of this plugin would be a software developer who should be able to fix security vulnerabilities shown by the secure coding framework, after referring to countermeasures given.
3. This framework requires the user to have a compatible version of IntelliJ IDEA IDE or any other supported JetBrains IDE up and running.

3.4 Parts of the secure coding guideline plugin for an IDE

According to the literature, secure coding guideline plugin mainly should provide features such as:

Code scanning: Scans the code for potential security vulnerabilities. This can be done using a variety of techniques, such as static code analysis, dynamic analysis, and fuzzing. Static code analysis focuses on analyzing code without running it. Dynamic code analysis focuses on analyzing code while running it. Fuzzy combines static and dynamic code analysis and uses fuzzers to analyze the code. This research focuses on detection of custom secure coding guidelines on the fly in the IDE. Therefore, static analysis is selected.

Customizable rule sets: Allows developers to add custom secure coding rules that the code scanner uses. This is useful for developing custom security guidelines for the organization/project or for focusing on specific security vulnerabilities.

Vulnerability reporting/Feedback: Provides feedback to the developer on the potential vulnerabilities that have been identified. The feedback typically includes a description of the vulnerability, and a recommendation for how to fix the vulnerability, if the rule creator has specified.

Integration with the IDE: The plugin is integrated with the IDE, so that developers can get feedback on their code as they write it. This helps developers to identify and fix potential security vulnerabilities early on.

Thus, a plugin that enforces secure coding guidelines and that supports IDE integration should mainly consists of:

I. **An IDE integration layer:** This allows the plugin to interact with the IDE and to provide features such as code scanning and code refactoring.
II. **Listener**: A listener who listens to detect a change of the editor, to scan and detect violations.
III. **Rule store:** A rule store to keep custom secure coding rules. This allows organizations to tailor the plugin to their specific needs and to enforce their own security policies.
IV. **Bot:** A bot who reads and understands AIML rules, and who matches end users' source code along with defined AIML rules to detect violations.
V. **Reporting layer:** A reporting layer that can generate outputs on the security vulnerabilities that were found in the code, to display in the IDE.

3.5 Main Design Choice: Interpreter and bot Architecture

As Identified and stated in Chapter 2, AIML was identified as the rule specifying language through the literature. This section focuses on the AIML interpreter generation part, to be able to understand AIML rules before the violation detection identification.

3.5.1 Available methods for generating an AIML interpreter

To be able to understand the AIML code, the plugin should be able to identify AIML code. While parsing elements of the AIML language to identify patterns, categories, and templates is involved, the goal is to interpret and execute the logic defined within the AIML. The plugin

should be able to process AIML patterns, match user input code files against them, and then execute the corresponding actions defined in the templates. These actions could involve assigning variables, checking for conditions, and generating responses. To be able to do this, we need an AIML interpreter. The primary function of this interpreter is parsing the structure of AIML but involves understanding the meaning and executing the intended actions.

The current AIML specification is AIML 2.1. According to the official site, in AIML 2.1 the only newly added support is new tags which support multimedia features (AIML Foundation, 2018). But in our coding guidelines domain, support for multimedia features is not needed. Therefore, in this context support for the AIML 2.0 specification is enough.

According to the literature, there are some AIML interpreters available for chatbots written in different languages and they support different AIML specifications. These AIML interpreters are built for chatbots, and there for simple natural language. But in the code matching context, there can be different characters like new line characters, tab characters etc. Therefore, in this interpreter implementation, per-processing of user input is needed before the matching part. Currently there is no AIML interpreter which supports performing of all the necessary pre-processing duties for code files, which supports latest AIML specification 2.1 or AIML 2.0 specification and written in Java (AIML Foundation, 2018). Therefore, in this research a new interpreter will be developed and will be used in the implementation of the system.

In pyaiml21 (Barnisin, 2022), considering the AIML code might contain different non-AIML tags, the researcher has created a two-phase AIML parser, which includes XML parsing separately and AIML validation separately. According to the researcher, this is because AIML is an untyped, Turing-complete programming language. This method is good in the domain of chatbots. But in our context, as the initial stage, we do not have and focus on arbitrary AIML tags. The interpreter will be created to parse pure AIML tags, which supports AIML 2.1 specification (AIML Foundation, 2018).

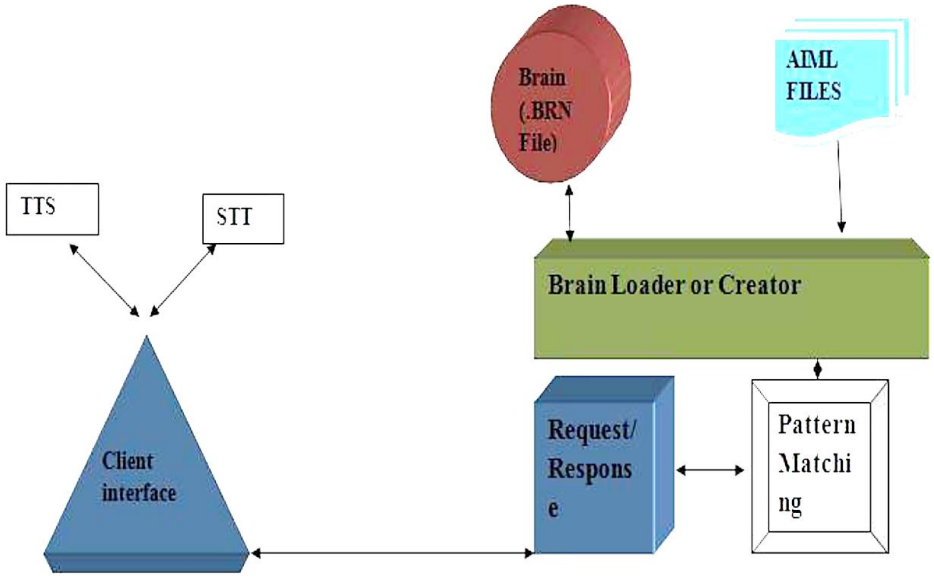


Figure 17: Flow Diagram of Request and Response of a chatbot which uses AIML (Ahmed & Singh, 2015)

The above diagram shows a data flow of requests and responses of a chatbot who uses AIML (Ahmed & Singh, 2015). Here, TTS is text to speech system and STT is the speech to text system. The interpreter loads the brain in the beginning and if it does not have a brain file, it creates a default one. Also, the interpreter loads AIML files in the beginning. Then, when a request comes, the system search for a match and generate an output according to AIML templates and returns it. According to the literature, the below is the complete procedure of a chatbot which uses AIML (Ahmed & Singh, 2015):

1. The first step is to create a brain file, it can be created by using AIML file.
2. If the brain file is not present, then create a brain.
3. If the brain file is present, then load the brain file in the model.
4. After loading the brain, BOT waits for the request from user or client.
5. A user can provide request either in format of voice or text.
6. Model receives request and forward to pattern manager.
7. Pattern matching algorithm is applied and sent to the brain.
8. The model uses the brain and gets appropriate response and forward to user.

3.5.2 Design of the AIML interpreter and the bot

3.5.2.1 Design

Therefore, can concludes that, the interpreter in his research should be responsible for:

1. Loading the brain in the beginning
2. Loading the end user defined AIML rule files in the beginning

Further, can concludes that, the bot in his research should be responsible for:

1. Pattern matching.
2. Generating outputs according to loaded AIML files.

3.5.2.2 Implementation decisions

When referring to the initial AIML interpreter for the A.L.I.C.E. chatbot, there are standard set of conceptual words that uses in an AIML interpreter. They are:

1. 'Graphmaster': consists of a collection of nodes called 'Nodemappers'.
2. 'Nodemappers': map the branches from each node. The branches are either single words or wildcards.
3. Botmaster: the master (developer) of robot

Out of above, first two are the main important concepts.

When searching for available GitHub repositories for an AIML interpreter written in Java there are a few which have developed for chatbots. Appendix B: Available GitHub repositories of AIML interpreters) lists the currently available AIML interpreters, developers have developed over the past years for chatbots. Out of the listed three, the first two have followed the initial concepts, and are more comprehensive. Further, out of the first two, the first one has followed more coding standards, and it is more readable. Therefore, re-used some files of the first repository.

3.6 Design Choice: The plugin base and output displaying

3.6.1 Plugin base

IntelliJ platform is powered by a plugin generation template and an official documentation on plugin development. Developed the base of the plugin using the template. Official plugin template repository of JetBrains and official documentation links are available under Appendix D: JetBrains official plugin development template and official documentation

3.6.2 Interaction with users' code and output display

For the editor listening part and for display the newly generated output, some code files of the previous research of secure coding plugin development are reused (Dasanayake, et al., 2019). Original source code repository link is available under Appendix C: Source codes and available plugins of previous research studies

3.7 Overall System Design

After identifying the whole process, a plugin is implemented and evaluated as stated in the next chapter. The high-level architectural overview below explains the layers of the system prototype (Figure 18: Architectural overview of the system prototype).

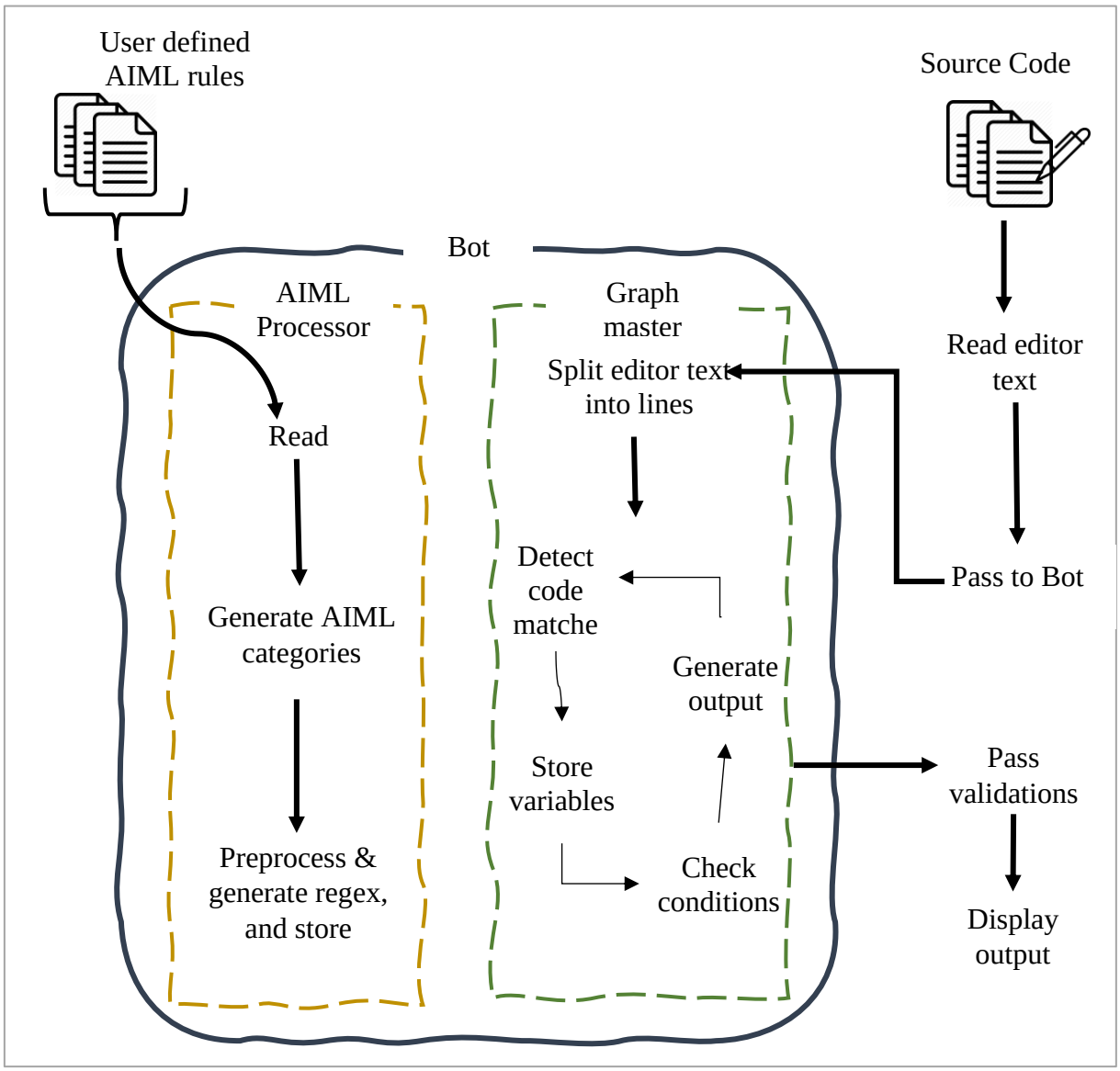


Figure 18: Architectural overview of the system prototype

3.7.1 Parts of the system prototype

3.7.1.1 An IDE integration layer

IDE integration layer allows the plugin to interact with the IDE. This consists of a ‘listener’, a reporting layer, and a rule store.

Listener: A one who listens to detect a change of the editor, then passes the opened code editor file to the system, to scan and detect violations.

Reporting layer: A reporting layer that can generate user friendly outputs based on the output of the bot on the security vulnerabilities that were found in the code, to display in the IDE. This layer is responsible for displaying the violation on the opened code file and also displaying the violation description under the plugin tab of the users' IDE.

Rule store: A rule store to keep custom secure coding rules of the user. This is a folder of anywhere inside the end users' machine.

3.7.1.2 Bot

As explained in the section 3.5.2.1 Design) this bot is responsible for loading the brain in the beginning, reading, and understanding user defined AIML rules, pre-processing code lines, and comparing end users' source code along with defined AIML rules to detect violations. This bot consists of two main parts:

AIML processor:

AIML processor is responsible for parsing AIML tags as per the AIML specification. This implements the AIML 2.0 specification as described in the official site (AIML Foundation, 2018). As described earlier, the only newly added feature for AIML 2.1 from 2.0 is the support for multimedia tags. But in our context, we do not need it. This processor is responsible for reading and understanding user defined AIML rules in the beginning.

Graph master:

This is the Brain of the bot. This holds data structures that store AIML sets and maps, and this holds an instance of the AIML processor. This is responsible for pre-processing input code lines. Further, this has AIML Pattern matching algorithms, and is responsible for detecting violations according to the patterns matched. Furthermore, this is responsible for generating outputs according to templates of matched categories in user defined AIML files.

3.8 Summary

In this chapter, the design of the framework for secure coding plugin is presented along with design choices and using the system architecture diagram. Design assumptions are listed

under the design assumptions section. The rationale for the need for an interpreter and the approach followed when designing and implementing the interpreter is also presented under this chapter.

CHAPTER 4
EVALUATION AND RESULTS

4.1 Introduction

The evaluation process was carried out to assess the solution of the rule creation mechanism and the detection process of prototype to verify whether the intended requirements have been met and are up to relevant standards. The focus is to ensure that the plugin-based framework has achieved the expected research objectives.

In the evaluation process, custom rule-based evaluation method and a user-based evaluation were selected to evaluate the ability to create custom secure coding guidelines, and to evaluate the ability of the framework to detect a violation of the specified rule.

An evaluation is carried out using the below methods.

1. Custom secure rule-based evaluation:
    a. Purpose: To evaluate the secure rules customization.
    b. Materials: A selected set of secure rules from both package-level and complex categories are used.
    c. Method: Created a selected set of rules, using the rule defining method of the framework. As the second step, reviewed results manually to identify false positives, false negatives, true positives, and true negatives of the Secure Coding Plugin.
2. User-based evaluation:
    a. Purpose: Evaluated the extent to which developers are tended to not to commit securely violated codes and evaluated a selected set of user-related aspects.
    b. Materials: plugin-based framework, downloadable, with a questionnaire

4.2 Justification: achieving the goal of the research study

This section focuses on how AIML solves the gap in this domain. Under this section, examples of AIML rules for package level secure coding guideline violation and a complex secure coding guideline violation will be described mapping AIML features and how AIML features solves the research gap.

4.2.1 Solving the gap of package level secure coding guideline violation

Let's take the previously described example of package level secure coding guideline violation again.

THI00-J. Do not invoke Thread.run()

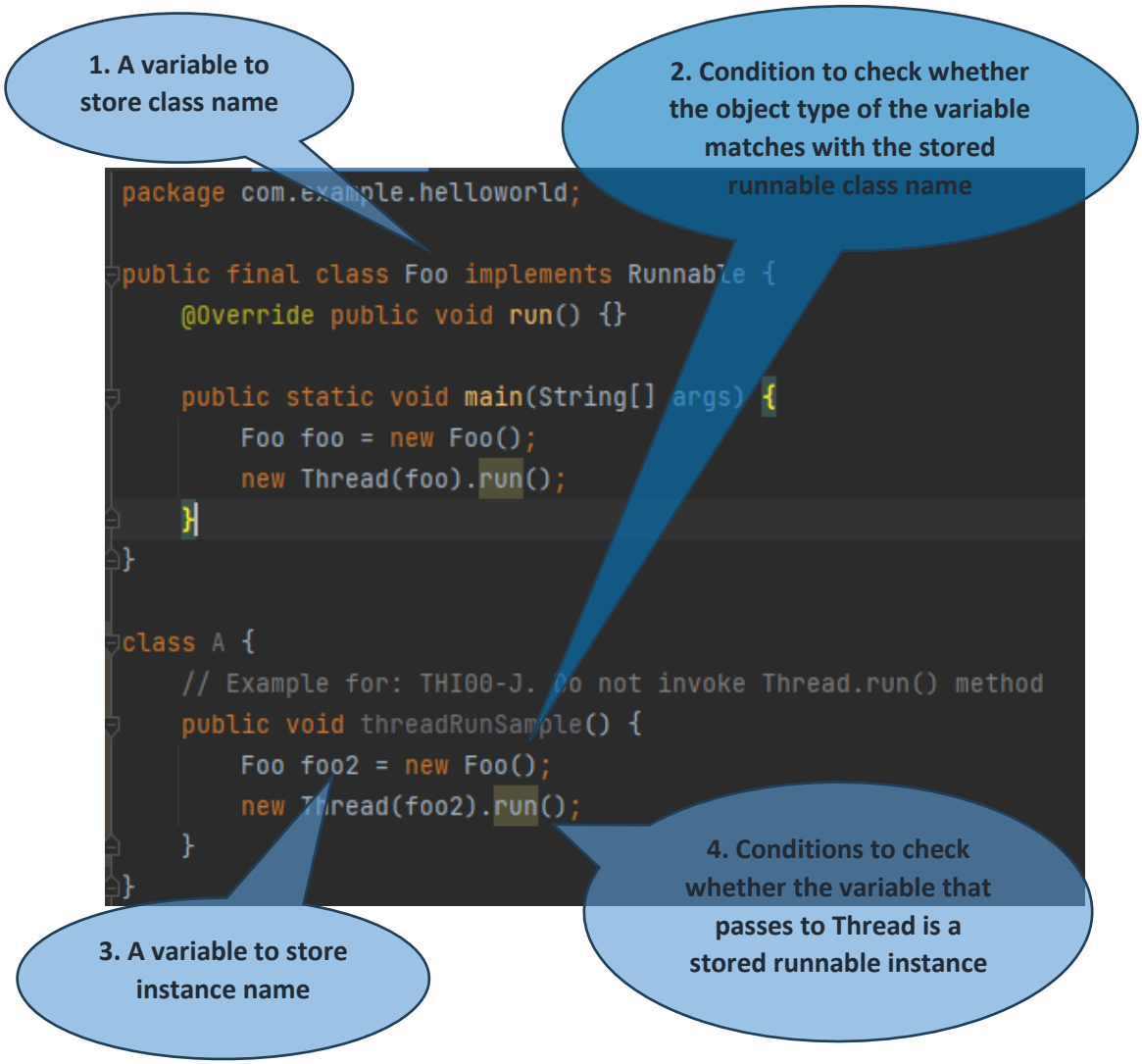


Figure 19: Ex1: A package level secure coding guideline violation

Here, as described earlier in the section (2.6.1 Nature of package level secure coding guideline violation), a secure coding guideline rule creation language should provide the facilities below:

1. The secure coding rule defining language should be able to track the custom class name, and therefore, it should be able to **store variables/ should support getters and**

setters to save custom data.

2. To detect a ‘Foo’ instance, when initializing any variable, the rule defining language should be able to say; compare the stored class type with the object type of the new variable. If matches, can say this is an object of the ‘Foo’ class. Therefore, the secure coding rule defining method should have a way to create **conditionals**.
3. After identifying the ‘Foo’ instance, should store the instance name of the variable. Again, the secure coding rule defining method should be able to **create and assign variables.**
4. Then after detecting a run() invoke, the rule creation mechanism should be able to compare the variable that passes to Thread is a stored runnable instance. To support this also, again we need **conditionals**.

The impact of lacking above mentioned things is, current secure coding guideline creation languages cannot define the **tracking of the code execution order**; and therefore, cannot define **detecting object creations of the custom ‘Runnable’ class beyond the original class**. Below is the AIML rule created to detect a violation of this coding guideline.

AIML coding guideline: THI00-J. Do not invoke Thread.run()

The AIML rule defined for detection of THI00-J is in below (Figure 20: Ex 1: How AIML provide facilities to create package level secure coding guidelines - part 1, and Figure 21: Ex 1: How AIML provide facilities to create package level secure coding guidelines - part 2). The steps of the rule are described in step numbers.

1. A pattern matching, and this step is to detect a custom runnable class.
2. Custom runnable class name is accessed based on the wildcard index.
3. Store custom runnable class name in a variable (name: runnableClassName). Further, variables to store a runnable instance name (name: runnableInstanceName), a variable to keep track whether the ‘run’ method is executed or not (name: calledRun), a variable to keep track whether the ‘run’ method is the run method of an instance of our custom Runnable instance (name: invokedThreadRun) are declared and initialized here. These setters and wildcard accessing in step 2 are in inside a think tag. Think tag tells the interpreter to process inside it without create/display an output.
4. Inside another category, a pattern matching to detect an object creation is in this step.

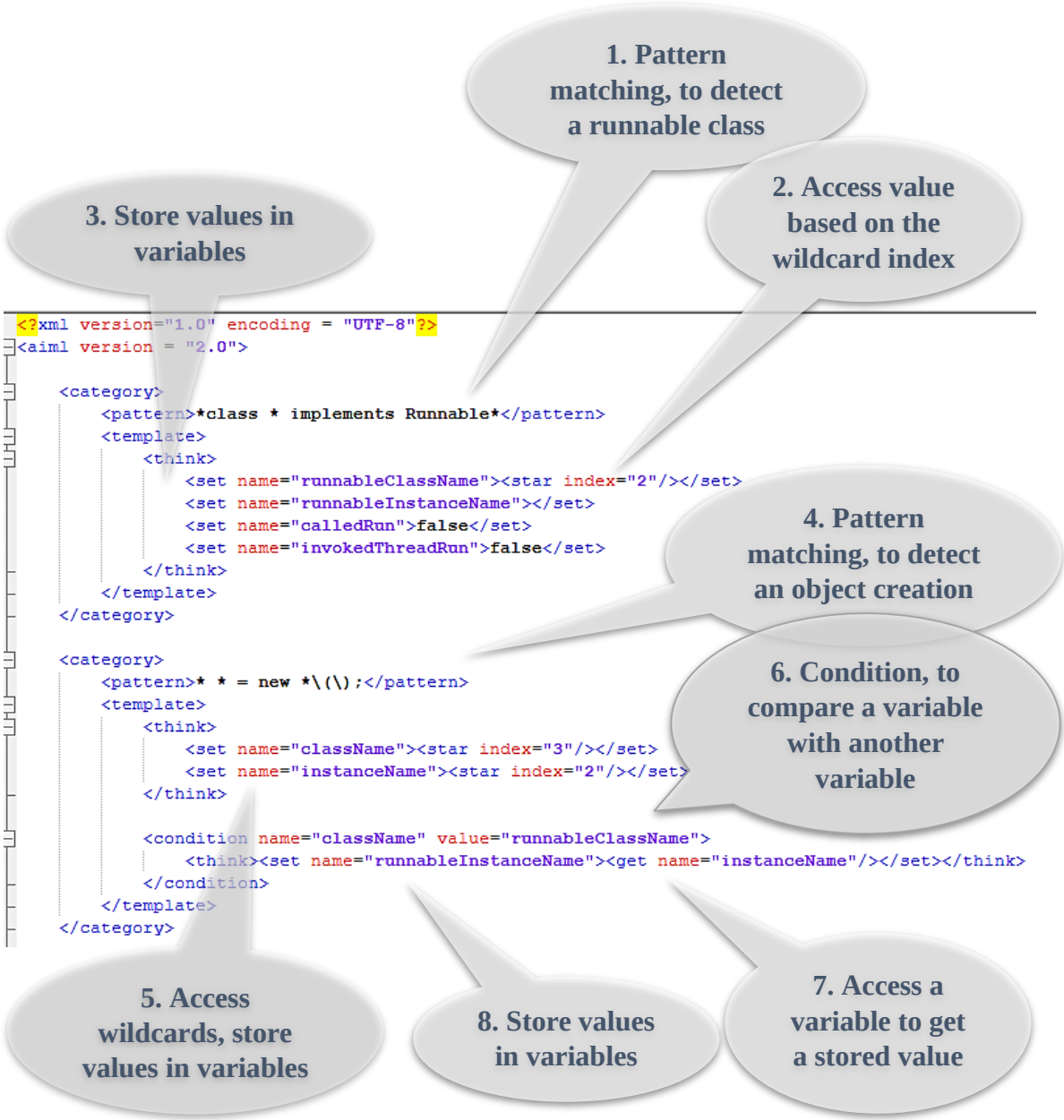


Figure 20: Ex 1: How AIML provide facilities to create package level secure coding guidelines - part 1

5. The object type of the new object creation is accessed based on the wildcard index (index: 3) and stored in a variable (name: className). Further, instance name is accessed based on the wildcard index (index: 2) and stored in a variable (name: instanceName). These are inside a ‘think’ tag.
6. A condition to compare the value of a variable with another variable value. This is to check whether the custom Runnable class name matches the object type of the newly created instance variable (variable names: className, runnableClassName). If condition is true, then do the work inside it.
7. A ‘getter’, to access a stored value of a variable. This is to get the instance name of the newly created variable (name: instanceName), to proceed further.

8. Change the variable value created to store runnable instance name to the value of the above accessed variable (variable name to store: runnableInstanceName), to keep track of the runnable instance name. Step 7 and this step are inside a ‘think’ tag.

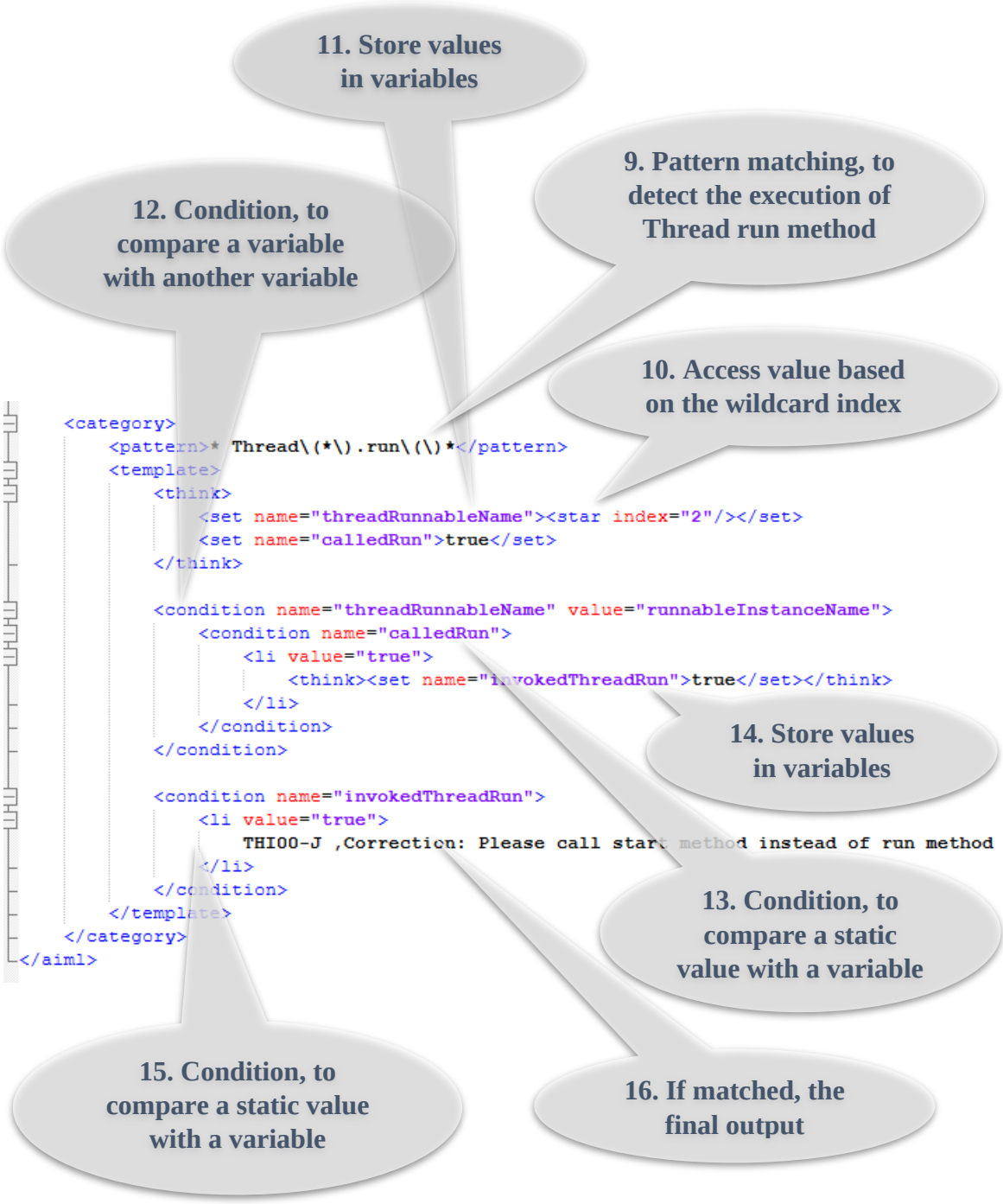


Figure 21: Ex 1: How AIML provide facilities to create package level secure coding guidelines - part 2

9. A pattern matching, to detect the execution of Thread run method.
10. Access the 2nd wildcard, to access the custom class name, used to create Thread.
11. Store the used custom class name which used to create Thread; before checking whether this matches our original custom Runnable instance name. Further changing

the variable value created to keep track whether the run method executed/not, to true (name: calledRun)

12. A condition to compare a variable value with another variable value. This is to check whether the used variable name inside ‘Thread’ matches our original custom Runnable instance name (variable names: threadRunnableName, runnableInstanceName). If this is a success, the inner condition will be checked.
13. A condition to compare a static value with a variable value (variable name: calledRun, static value: true). This is to check whether the run method has been executed. This condition will be executed only if the variable used to create ‘Thread’ matches with the custom runnable instance name as described in step 12. If this is a successful match, inner steps inside the ‘think’ tag in ‘li’ tag will be executed.
14. Set the variable name value to true, to keep track of the successful match of the invocation of the Tread.run() method along with the custom Runnable class (variable name: invokedThreadRun, value to assign: true).
15. A condition to check for the invocation of the Tread.run() method along with the custom Runnable class (variable name: invokedThreadRun, value to check: true). If this is a success, do the work inside it.
16. Displaying the violation code and the description. This is not inside a ‘think’ tag, because we need to display the output here.

Results: Detected violation of THI00-J. Do not invoke Thread.run()

Below shows the output of the secure coding guideline violation detection plugin (Figure 22: Ex 1: Detected violations of package level secure coding guideline). The plugin has detected violations of the above rule in both class scope and the package scope. In this screenshot, cursor is on the 2nd highlighted violation and it displays the output as a popup. This is a true positive.

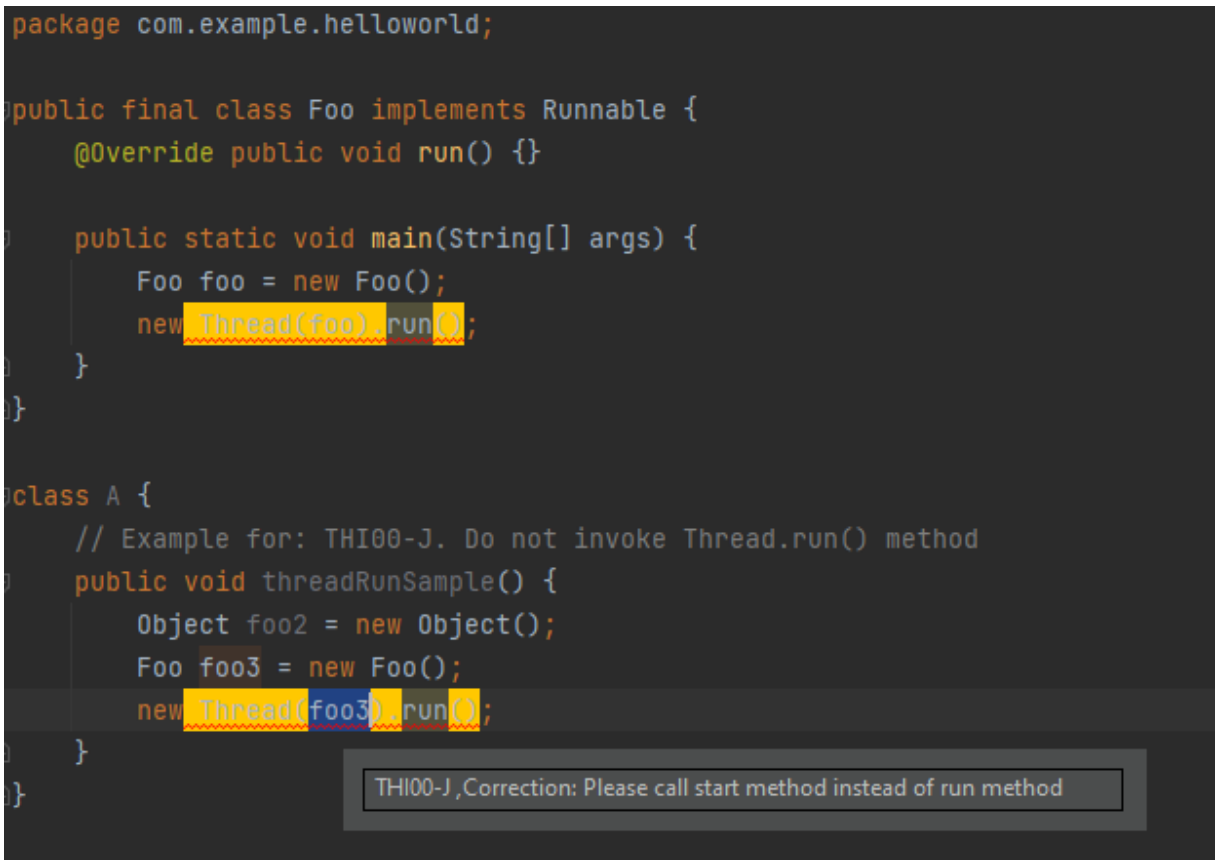


Figure 22: Ex 1: Detected violations of package level secure coding guideline

4.2.2 Solving the gap of complex secure coding guideline violation

Let's take the previously described example of complex secure coding guideline violation again.

IDS01-J. Normalize strings before validating them;

Here, as described earlier in the section (2.6.2 Nature of a complex secure coding guideline), a secure coding guideline rule creation language should provide below facilities:

1. The rule creation language should be able to keep track of the string variables ('strInput' in the below code example). Therefore, a rule creation language should have support for **store variables/ should support getters and setters**. Further, we might need **another set of variables to keep track of the execution order of the code**. Because in the below example, even though the 'strInput' is normalized, it is normalized after validating it. Therefore, tracking code execution order is very important. Checking for a normalization is not just enough.
2. A **variable** in the rule creation language to track whether the 'strInput' is normalized

might needed and initially this is ‘false’.

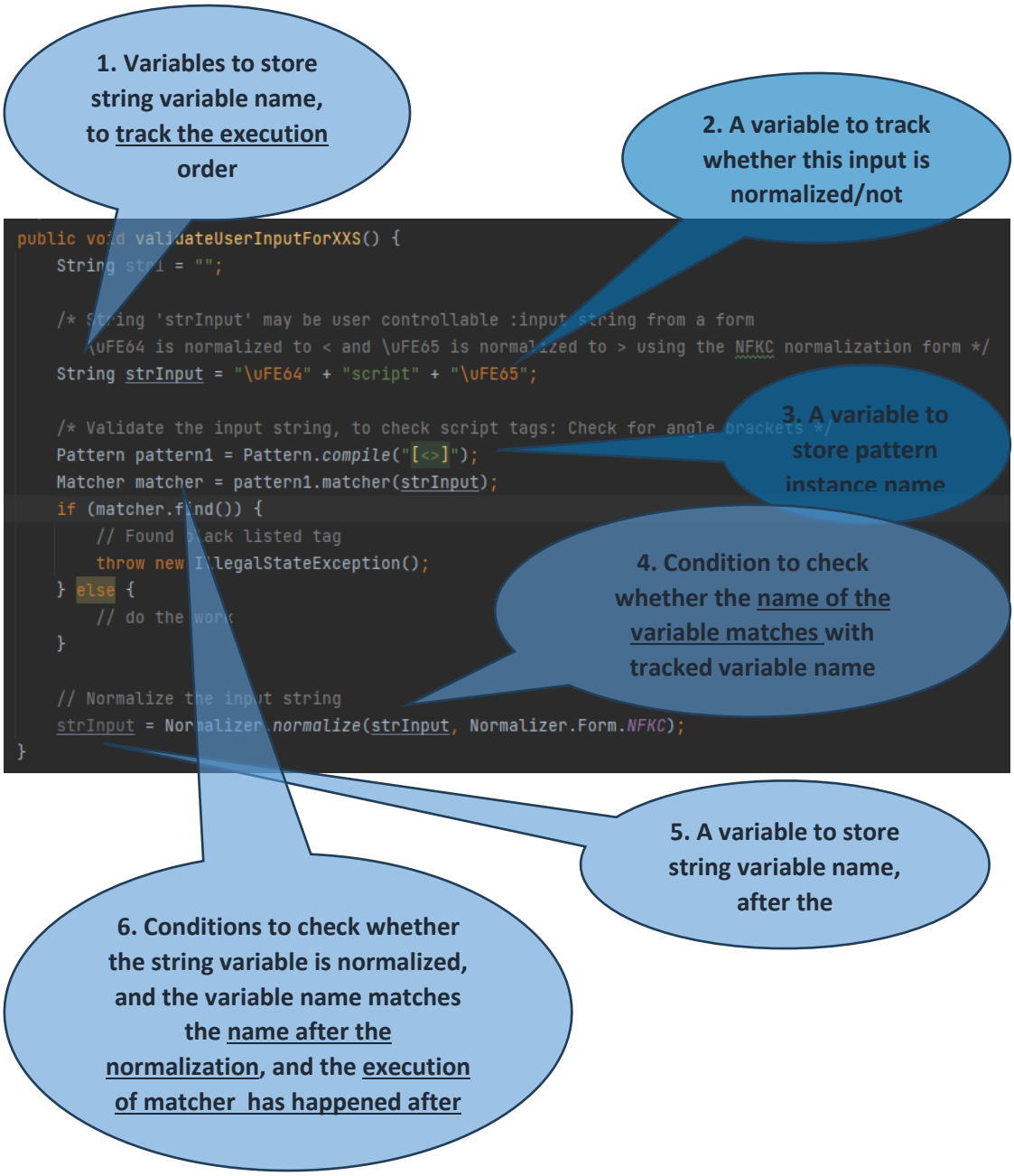


Figure 23: Ex2: A complex secure coding guideline violation

3. Another variable might be needed in the rule creation language to store the pattern instance name.
4. There might be a lot of string variables in a program inside even a method. As an example, can take the ‘str1’ variable in the below code. **Condition** to check whether the name of the variable inside normalization matches with tracked variable name of the ‘strInput’ might needed in the rule creation language.
5. Even though ‘strInput’ is normalized, it might be assigned to another new variable. Therefore, the rule must check for the new variable name after the normalization, in

the validation step. If a developer uses a new variable to store the normalized string, and if the developer accidentally validates the older string, that is again incorrect. Therefore, rule creation language might need **variables/ getters and setters** again to keep track of this.

6. Final step is:
    a. Check whether the input string is normalized. A **condition** needed to compare the stored track value is 'true', and a **getter** needed in the rule creation language to access the stored variable value.
    b. Check whether the string variable name inside the 'matcher' matches the string variable name after the normalization. A **condition** needed, to compare the stored string variable name after the normalization matches the string variable name inside the 'matcher'. A getter needed to access the stored variable value after the normalization.
    c. Another **condition** is needed, to check whether the execution of matcher has happened after the string normalization. The rule creation language might use **getters** of variables that are used to track the execution order of the code.
7. If everything matches for a violation, then the rule creation language might **output** the violation description, to display in the IDE.

AIML coding guideline: IDS01-J. Normalize strings before validating them

The AIML rule defined for detection of IDS01-J is in below (Figure 24: Ex 2: How AIML provide facilities to create complex secure coding guidelines - part 1, and Figure 25: Ex 2: How AIML provide facilities to create complex secure coding guidelines - part 2). The steps of the rule are described in step numbers.

1. A pattern matching, and this step is to detect a string variable initialization. If there is a string variable creation, processes inside 'think' tag of 'template' tag will be executed.
2. The string variable name is accessed based on the wildcard index.
3. Store the string variable name in a variable (name: strVar). Further, a variable to keep track of whether the string variable is normalized (name: strVarNormalized), a variable to store the variable name after string normalization (name: strVarAfterNormalize) are declared and initialized here.
4. Inside another category, a pattern matching to detect an object creation of the 'Pattern' class.

5. Access the pattern instance name, based on the wildcard index (index: 1) and stored in a variable (name: patternVar). This is also inside a ‘think’ tag.
6. Inside another category, a pattern matching to detect a string normalization.
7. Store the string variable name before normalization in a variable (name: strVarBeforeNormalize), based on the wildcard index (index: 2). Aim is to compare the previously declared string variable name in the source code. Further, the variable declared to keep track of the string variable name after normalization will be updated based on the wildcard (variable name: strVarAfterNormalize, wildcard index: 1).
8. A condition to compare the value of a variable with another variable value. This is to check whether the original string instance name matches the string variable name after the normalization (variable names: strVarBeforeNormalize, strVar). If the condition is a success, then do the work inside it.
9. If the condition is a success, change the value of the variable created to keep track of whether a string normalization happens to ‘true’ (name: strVarNormalized).
10. Another pattern matching, to detect a Java pattern matcher. If a ‘matcher’ is detected, proceed according to the ‘think’ tag of the ‘template’ tag.
11. Access the pattern instance name, based on the wildcard index (index: 2) and access the string variable name inside the matcher based on the wildcard index (index: 3).
12. Store above values in variables, to track variable names used in the matcher (names: patternInMatcher, strInMatcher) and initialize a new variable to keep track of whether the correct normalized string variable after normalization is used inside the matcher (name: correctStrVarNormalized) and the current value of this is ‘false’.
13. Conditions, to compare a variable with another variable (names: patternInMatcher, patternVar) to check whether the matcher happens based on the detected correct ‘pattern’ object. If this is a success, another variable-with-variable condition to check whether the correct string variable after normalization is used in the matcher (names: strInMatcher, strVarAfterNormalize).
14. If the above two conditions are successful (in step 14), a condition, to compare a static value with a variable value (variable: strVarNormalized, value to check: true). This is to check whether the string normalization has happened on the correct string variable. If this condition is a success, proceed ‘think’ tag inside it as per step 15.
15. Change the variable value created to keep track of whether the correct normalized string variable after normalization is used inside the matcher (name: correctStrVarNormalized) to ‘true’.

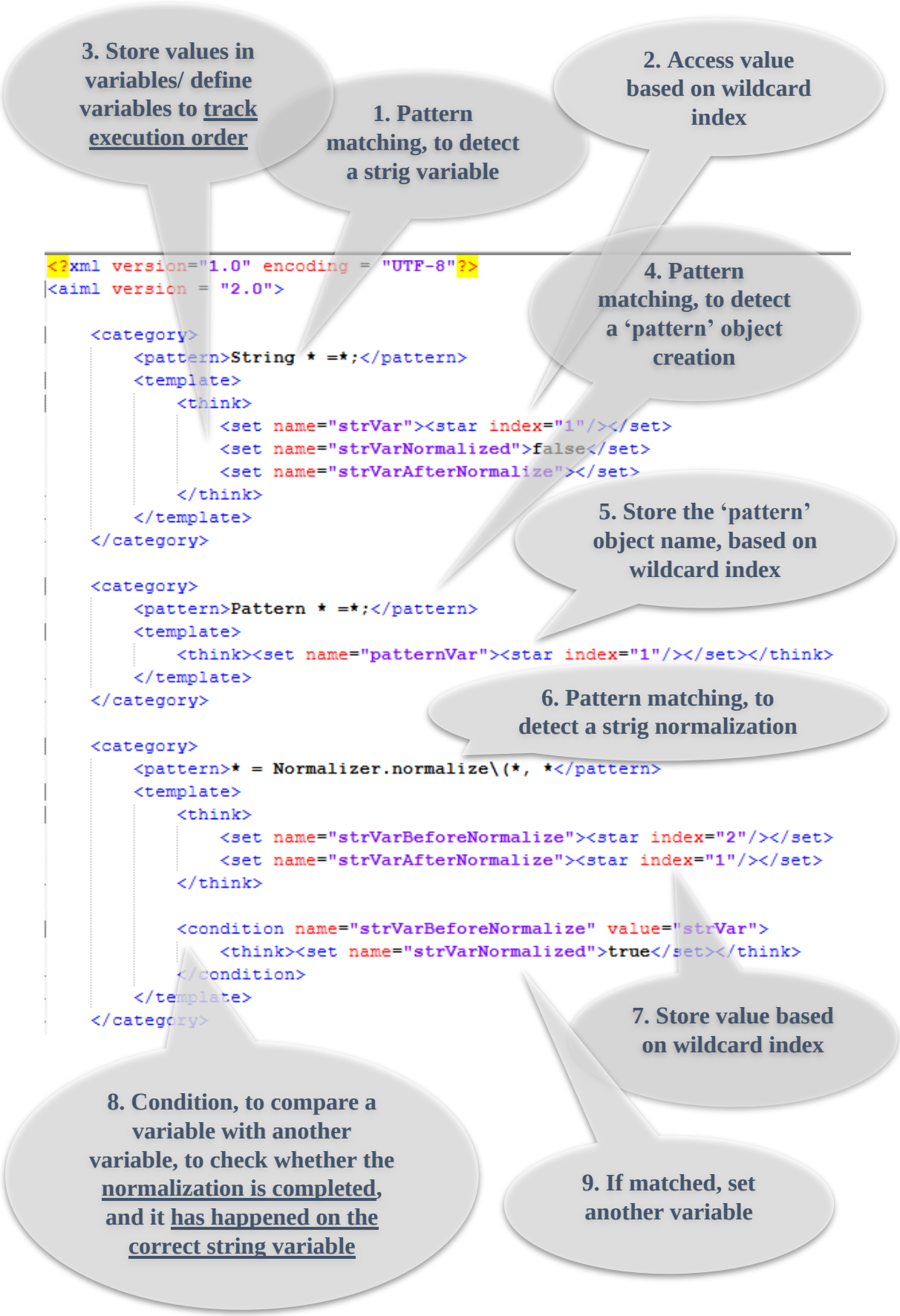


Figure 24: Ex 2: How AIML provide facilities to create complex secure coding guidelines - part 1

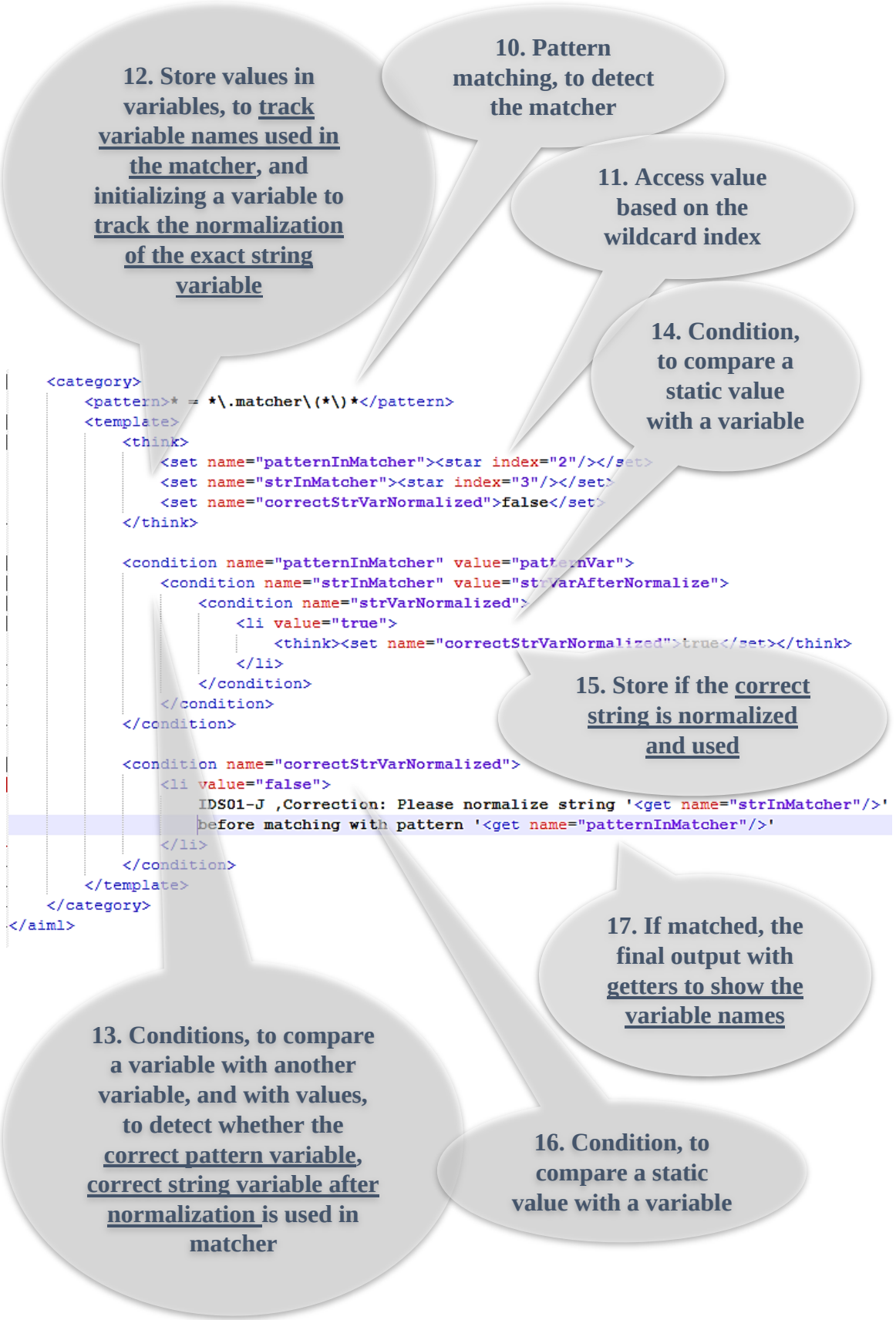


Figure 25: Ex 2: How AIML provide facilities to create complex secure coding guidelines - part 2

16. Before the output generation, another condition, to check the variable value created to keep track of whether the **pattern matcher has used normalized string variable,** and **it is the correct variable after normalization** (name: correctStrVarNormalized) is 'false'. If this is a success, proceed with step 17.
17. Final step to generate the output. The output consists of secure coding guideline rule number, description and also the variable names associated with the violation of the specified secure coding guideline rule.

Results: Detected violation of IDS01-J. Normalize strings before validating them

Scenario 1:

The correct string variable has normalized, before the matching (variable 'strInput' has normalized, and the normalization has happened before the patter matching), also the pattern matcher has used the correct string variable name after the string normalization (still the variable is 'strInput):

```
/*
 * Example for: IDS01-J. Normalize strings before validating them
 * This is to prevent accepting untrusted input strings
 * ex: an application's strategy for avoiding cross-site scripting (XSS) vulnerabilities may include
 * forbidding <script> tags in inputs
 */
public void validateUserInputForXXS() {
    String str1 = "";

    /* String 'strInput' may be user controllable :input string from a form
       \uFE64 is normalized to < and \uFE65 is normalized to > using the NFKC normalization form */
    String strInput = "\uFE64" + "script" + "\uFE65";

    // Normalize the input string
    strInput = Normalizer.normalize(strInput, Normalizer.Form.NFKC);

    /* Validate the input string, to check script tags: Check for angle brackets */
    Pattern pattern1 = Pattern.compile("[<>]");
    Matcher matcher = pattern1.matcher(strInput);
    if (matcher.find()) {
        // Found black listed tag
        throw new IllegalStateException();
    } else {
        // do the work
    }
}
```

Figure 26: Ex2: Plugin does not output anything for a correctly followed coding guideline

Results:

No output because no violations. Therefore, a true negative.

Scenario 2:

The correct string variable has normalized (variable ‘strInput’) also the pattern matcher has used the correct string variable name after the string normalization (still the variable is ‘strInput). But the normalization has happened after the pattern matching:

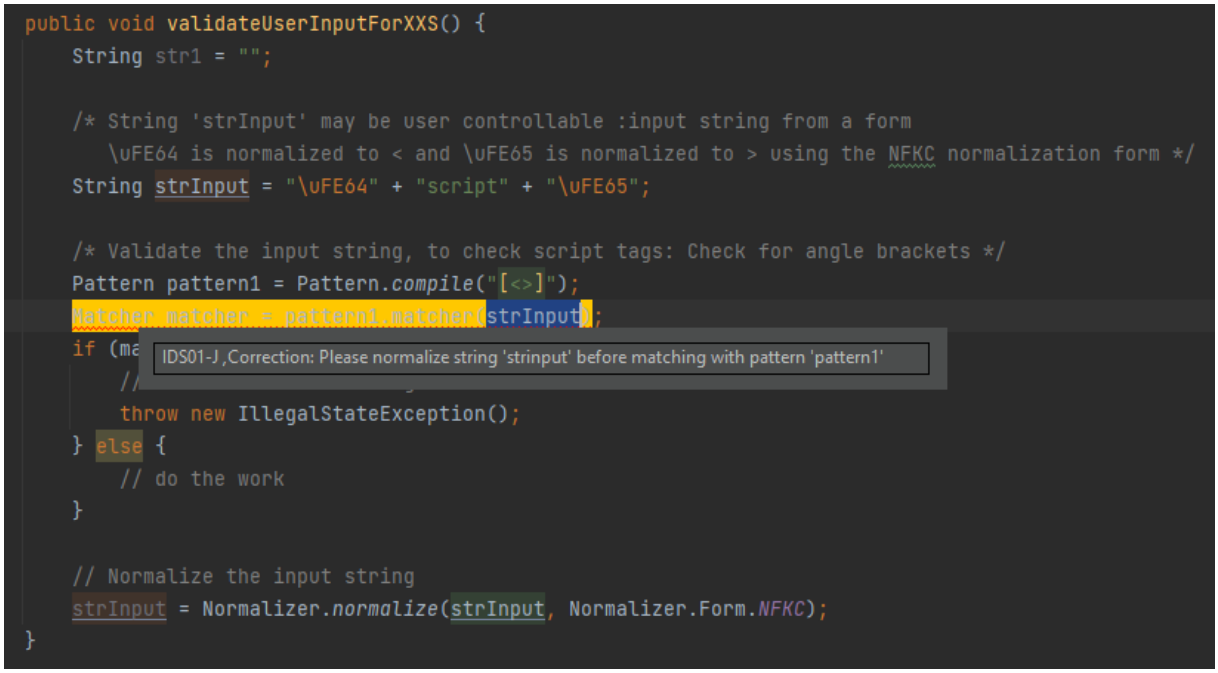


Figure 27: Ex2: Plugin shows the defined output for a coding guideline violation

Results:

Plugin shows the defined output because of the violation. Therefore, a true positive.

Scenario 3:

The correct string variable has normalized (variable: ‘strInput’) also the normalization has happened before the pattern matching. But the normalized value has assigned to another string variable (variable name: ‘str1’) and the pattern matcher has used the old string variable name. Thus, incorrect string variable has used in the pattern matcher.

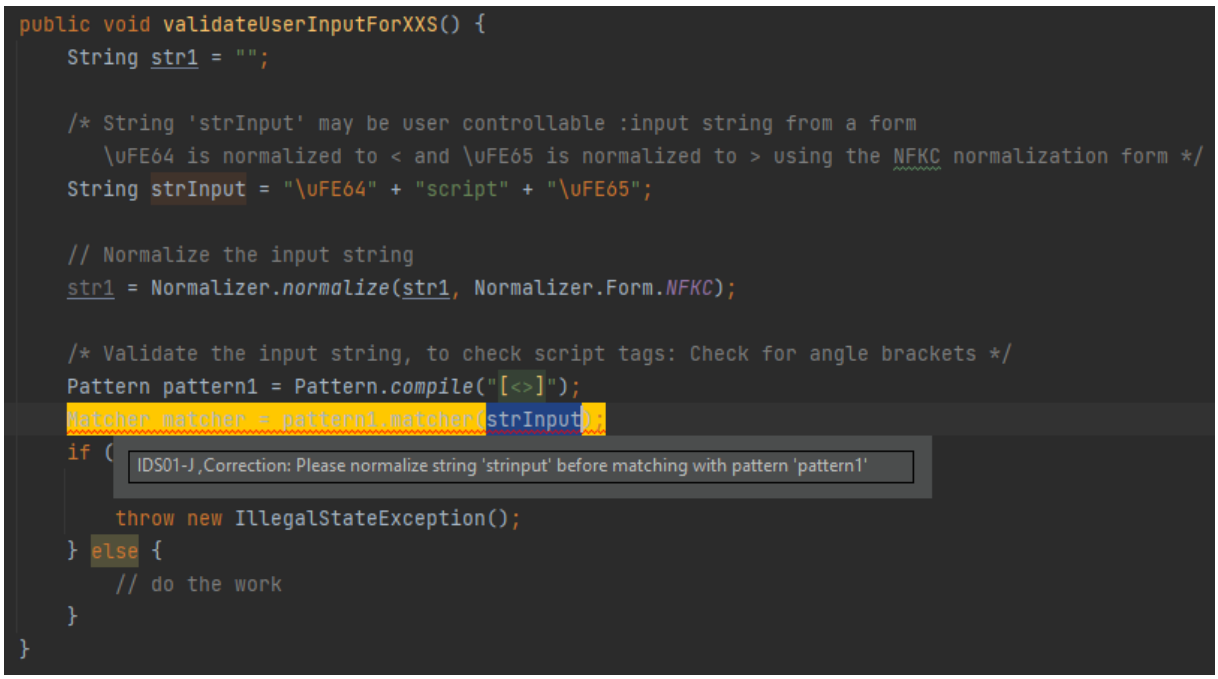


Figure 28: Ex2: Plugin shows the defined output for a coding guideline violation

Results:

Plugin shows the defined output because of the violation. Therefore, a true positive.

4.2.3 Summary of the justification

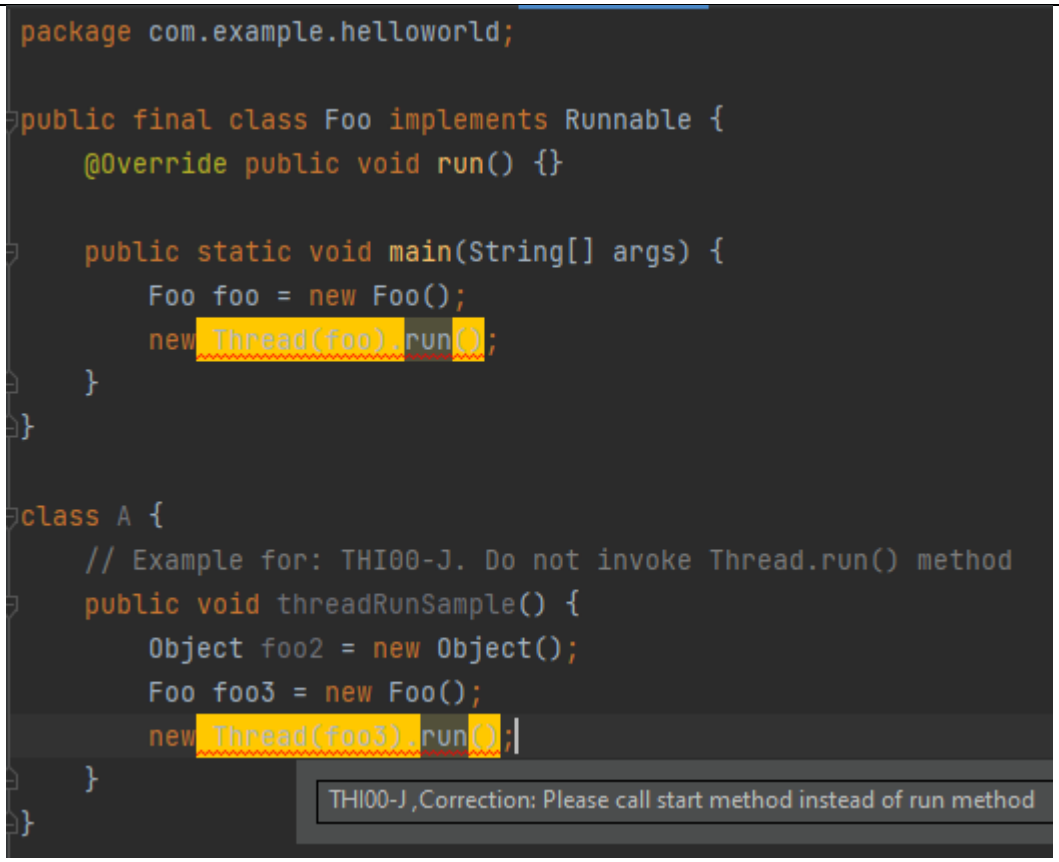
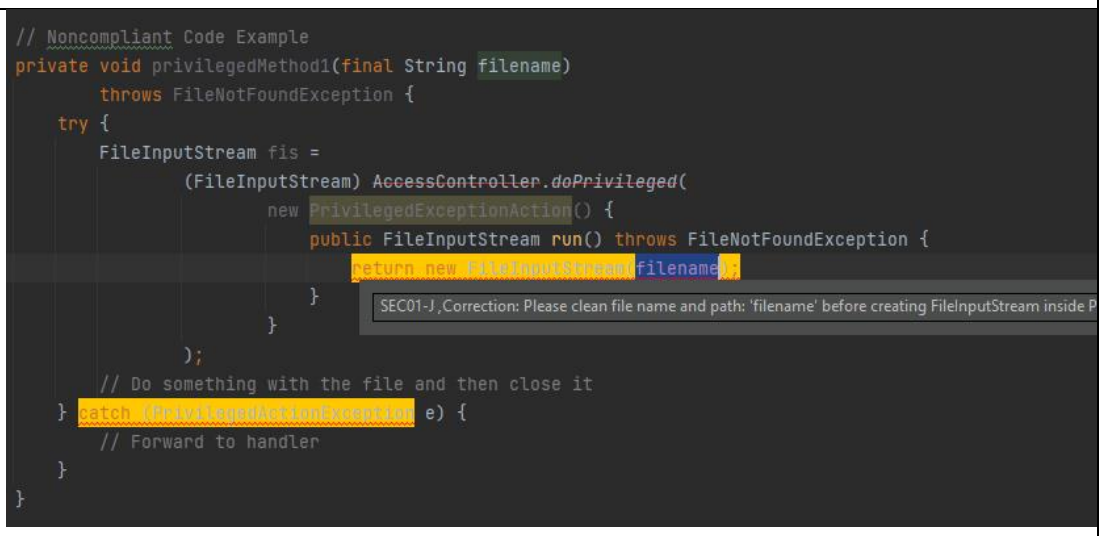
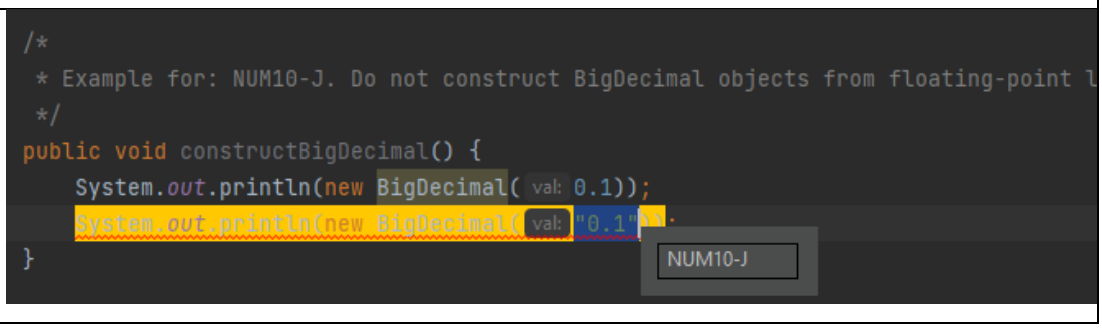
Therefore, can conclude that the proposed secure coding guideline rule defining language can define both package level secure coding rules and complex secure coding rules which needs the track of the control flow of the source code.

4.3 Custom secure rule-based evaluation

4.3.1 Evaluation

Secure coding guidelines research of Dasanayake et al. (Dasanayake, et al., 2019) has support for five package level rules, and these rules are hard coded. To demonstrate the ability of the selected rule creation mechanism of this research methodology, selected the same set of rules and created AIML rules as a verification. The below table includes the result outputs of the IDE. Created AIML rules are under the section Appendix I: AIML secure coding rules created for the evaluation

Rule name	Displayed output for a violation

THI00-J	
SEC01-J	
NUM10-J	

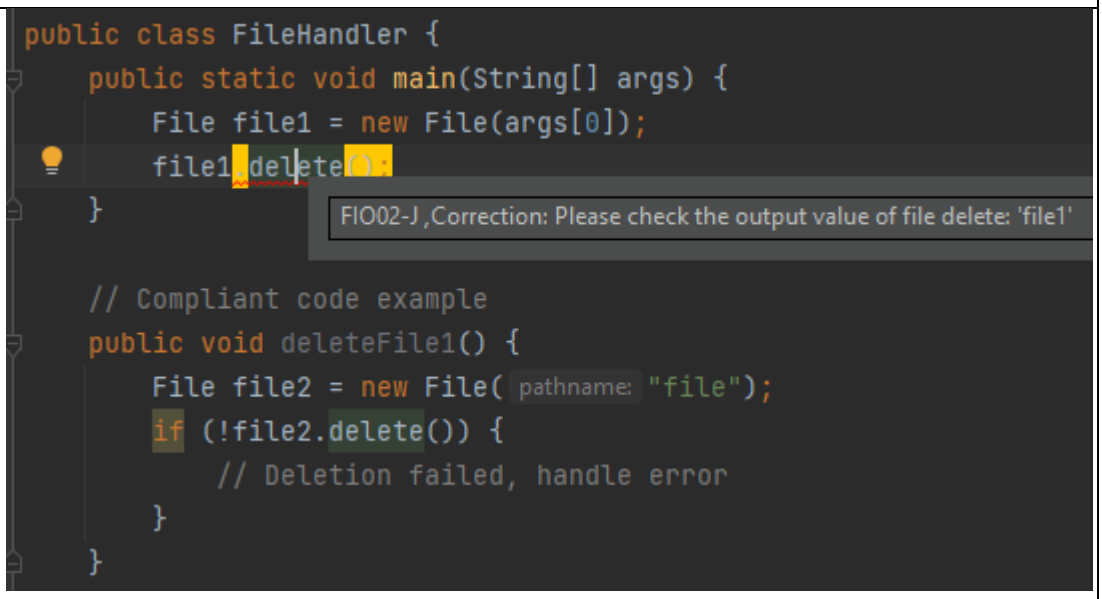
SEC07-J	
FIO02-J	

Table 8: Custom secure rule based evaluation

4.3.2 Summary

The above results give an indication that, even though the set of rules is not selective to the rule creator, features of the proposed rule creation mechanism facilitate creation of secure coding rules. Further, can conclude that the proposed plugin-based prototype can detect the non-biased set of rules created by the rule creator.

4.4 User based evaluation

4.4.1 Introduction

The user-based evaluation was carried out to assess the ability of knowledge sharing of the plugin-based framework and to assess other usability aspects. The evaluation was performed allowing a set of users to download the plugin and then giving them a questionnaire.

4.4.2 Questionnaire

The below is the set of questions and provided options added to the questionnaire to analyze the knowledge related to the awareness of secure coding guidelines, nature of secure coding

guidelines followed by users, rule creation ability of the suggested method, ability to detect violations of created rules and the overall ability of the tool in terms of knowledge sharing.

1. Are you a software developer? How do you rate yourself with respect to your experience?
    a. Not relevant
    b. Undergraduate/intern
    c. Beginner
    d. Intermediate
    e. Experienced
2. Are you aware of secure coding guidelines?
    a. Yes
    b. No
3. Do you follow any secure coding guidelines while coding?
    a. Yes
    b. No
4. What nature of the secure coding guideline do you follow?
    a. No
    b. Standard guideline (ex: OWASP, Oracle, SEI CERT)
    c. A guideline provided by the company/ A guideline you and your team created / Any other custom guideline.
5. Were you able to create custom secure coding guidelines for the framework?
    a. Didn't try it.
    b. Yes
    c. No
6. Did you encounter any issues while creating rules? If yes, kindly give a brief description.
7. Did you try your created rules, with example violations?
    a. Yes
    b. No
8. Did the plugin detect violations correctly?
    a. Yes
    b. No
    c. Not relevant
9. How do you rate this plugin in terms of Accuracy? (Whether mentioned violations

were detected)

a. Poor
b. Fair
c. Good
d. Excellent

10. How do you rate this plugin as a knowledge sharing tool? (Your team/organization can distribute a set of custom secure coding guidelines using this, therefore everyone in it can use that)
    a. Poor
    b. Fair
    c. Good
    d. Excellent

4.4.3 Analysis of results

Based on the feedback from several users, can see that a considerable number of developers are not following a secure coding guideline, even though they are aware of guidelines. Further, the analysis shows that most of the developers who follow a guideline follow a custom guideline, and not a standard one. Below are the overall results of the user-based evaluation, and this shows the plugin can create and detect custom secure coding guidelines, and this can act as a knowledge sharing tool.

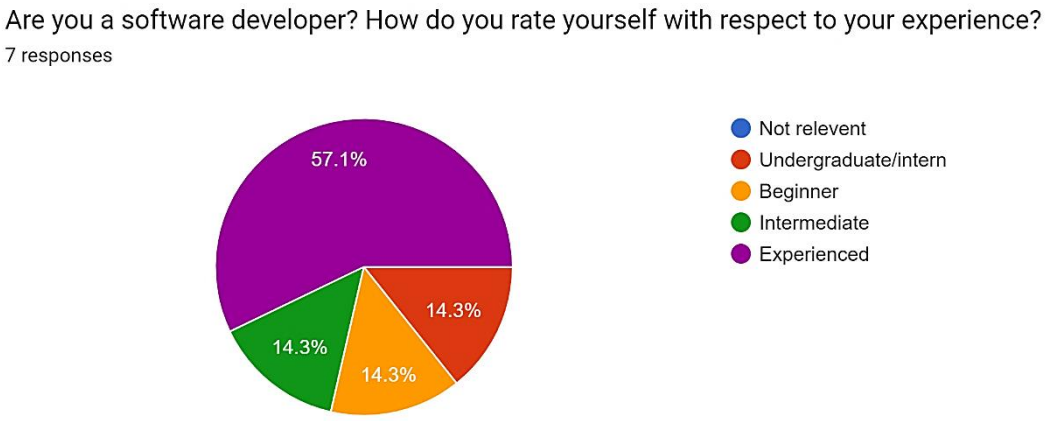


Figure 29: Responses for questionnaire: question 1

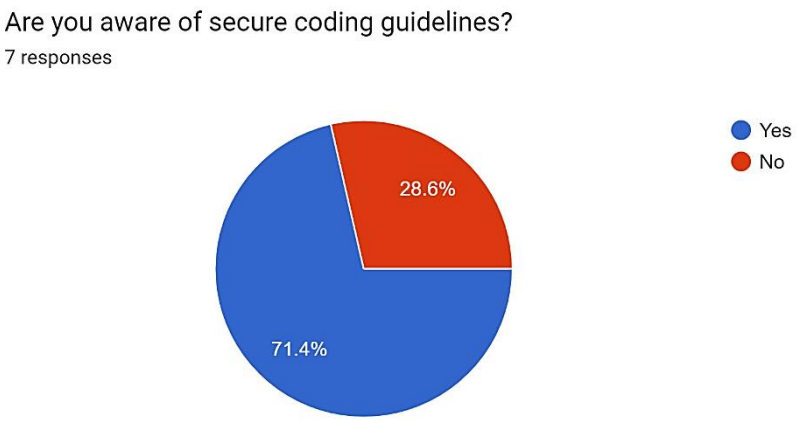


Figure 30: Responses for questionnaire: question 2

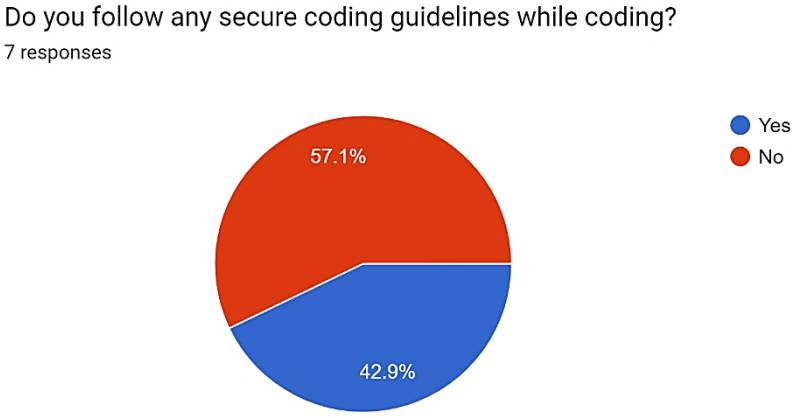


Figure 31: Responses for questionnaire: question 3

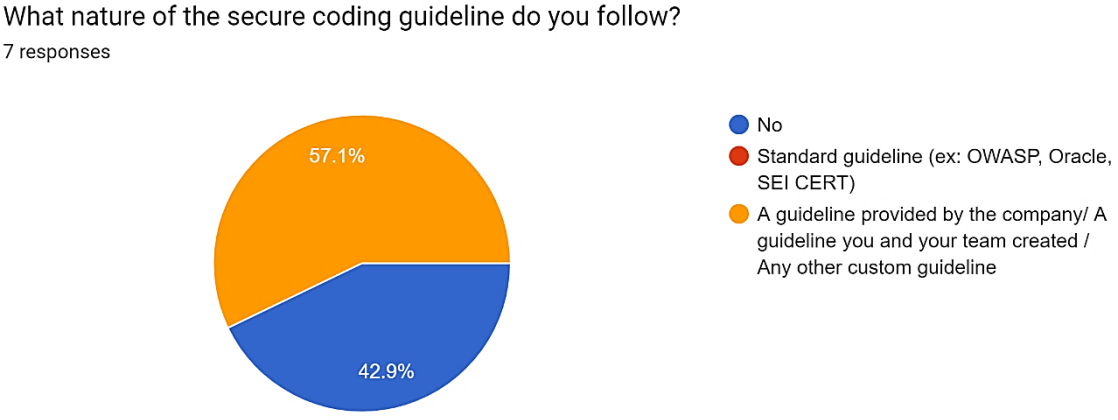


Figure 32: Responses for questionnaire: question 4

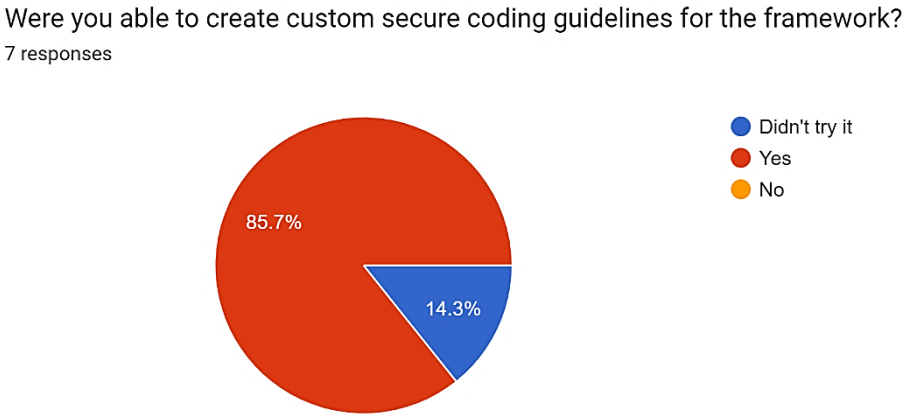


Figure 33: Responses for questionnaire: question 5

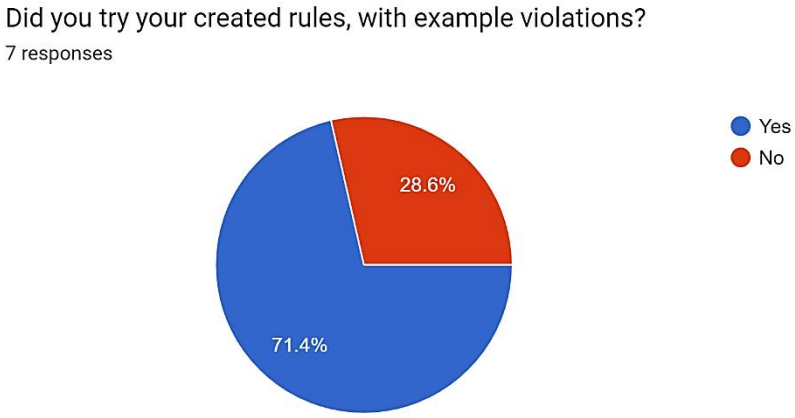


Figure 34: Responses for questionnaire: question 7

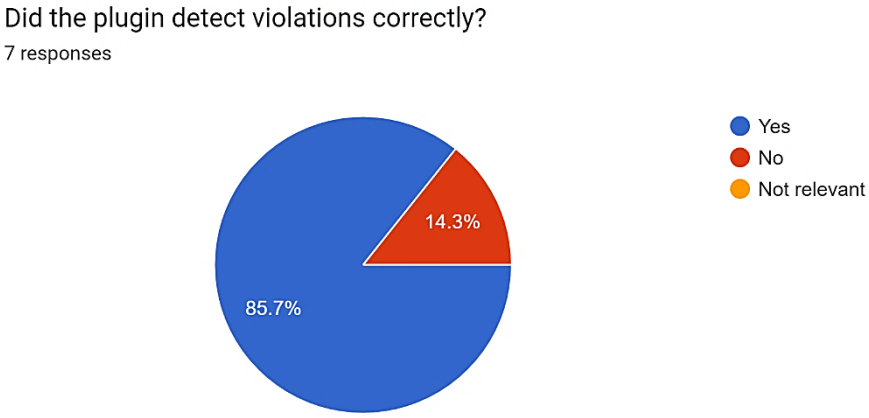


Figure 35: Responses for questionnaire: question 8

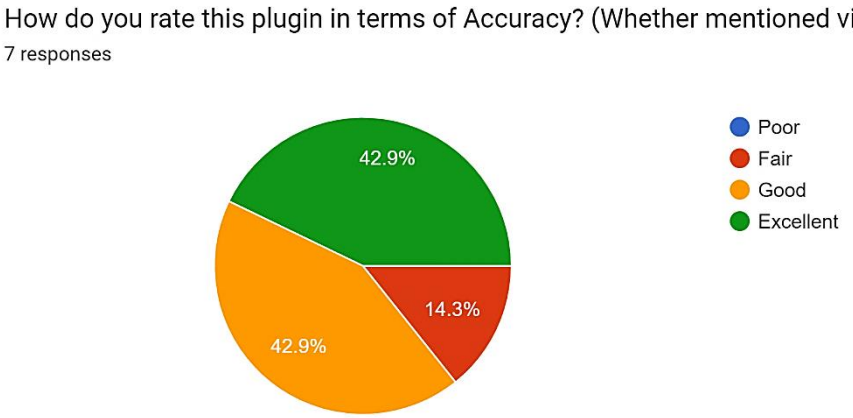


Figure 36: Responses for questionnaire: question 9

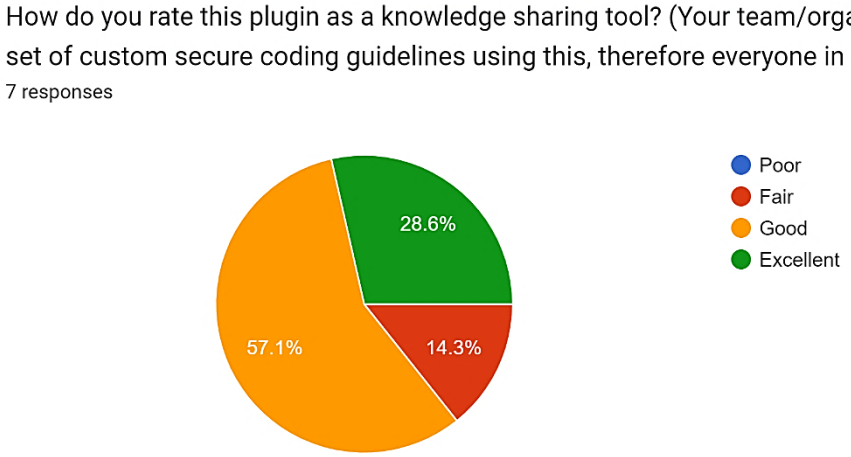


Figure 37: Responses for questionnaire: question 10

4.4.4 Conclusion

From the above user-based evaluation, it can be concluded that the overall usability aspects of the proposed system are good and can this kind of tool can act as a knowledge sharing medium.

4.5 Summary

This chapter focused on the evaluation of the proposed research methodology, along with the implemented prototype. The Justification section proves that the proposed rule creation mechanism can be used to create package-level and other complex secure coding guidelines which needs the track of the code execution order. Custom rule-based evaluation seconds this. Further, overall user-based evaluation also concludes that.

CHAPTER 5
CONCLUSION

As described in the chapter 1, this research work concerns the problem of *available on-the-fly detection supporting systems do not support creating complex custom rules which need the tracking of the control flow of the code* and *available on-the-fly detection supporting systems do not support creating custom rules for package level secure coding guideline violations* (Gasiba, et al., 2021) (Cremer, et al., 2020) (Dasanayake, et al., 2019) and no proper mechanism to *convert user-dependent complex or package level rules into algorithms* (Cremer, et al., 2020) (Dasanayake, et al., 2019) in existing research studies. We have proposed a new AIML based solution to resolve the above issues. As a prototype, this solution is implemented as a plugin, and it can be used offline once the user has the plugin installed. Evaluation and results chapter concludes that the proposed AIML based secure coding guideline rule defining language can define both package level secure coding rules and complex secure coding rules which needs the track of the execution order of the source code.

However, there are a few areas a future researcher may focus on. Firstly, AIML variable names are global. Therefore, if you accidentally use the same variable name for more than one rule, this will give you an incorrect validation output. Further, when the complexity of the rule becomes higher and when the number of steps of the code execution order becomes higher, the AIML rule gets complex. Therefore, future work may focus on a method to easy generation of rules, instead of writing in pure AIML. Furthermore, may focus on extending interpreter with XML new tags to support custom tags along with AIML tags.

APPENDICES

Appendix A: Source codes of this research study

I. Source code of the framework: https://github.com/ThilankaBowala/SecureCodingGuideline
II. Source code of the test project: https://github.com/ThilankaBowala/HelloWorld
III. JetBrains marketplace link: https://plugins.jetbrains.com/plugin/23904-securecodingguideline

Appendix B: Available GitHub repositories of AIML interpreters

Below are the currently available AIML interpreters written in java, developers have developed over past years for chatbots:

1. https://github.com/AIMLang/aiml-java-interpreter/tree/master
2. https://github.com/deepsarda/Aeona-Aiml/tree/main
3. https://github.com/karrarkazuya/aiml-java-interpreter/tree/master

Appendix C: Source codes and available plugins of previous research studies

I. Framework for secure coding (Dasanayake, et al., 2019)
    a. Idea plugin url: https://plugins.jetbrains.com/plugin/11265-framework-for-secure-coding
    b. Source code url: https://bitbucket.org/lasithd2/seproject_framework_for_secure_coding/src/master/
II. Sensei (Cremer, et al., 2020)
    a. Documentation: https://docs.sensei.securecodewarrior.com/intro.html

Appendix D: JetBrains official plugin development template and official documentation

I. Official plugin template repository of JetBrains: https://github.com/JetBrains/intellij-platform-plugin-template
II. Official documentation: https://plugins.jetbrains.com/docs/intellij/welcome.html

Appendix E: Available Commercial Static Analysis Tools

During the phase of literature review, studied available static analysis tools in the domain and available research studies to identify available gaps in the domain. But to refrain from changing the focus, added only a summary of available static analysis tools under CHAPTER 2
LITERATURE REVIEW, and added a descriptive version here.

1. SpotBugs

SpotBugs is a tool that uses the concept of “bug patterns” to detect bugs in Java (spobugs, 2016-2022). It can be considered as the successor of *FindBugs* tool (Lenarduzzi, et al., 2023) (Dasanayake, et al., 2019). *SpotBugs* is a lightweight open-source analysis tool capable of finding a wide range of software bugs, including a number of security bugs. The tool supports more than 400 “bug patterns” with reference to the Open Web Application Security Project (OWASP) Top 10 and Common Weakness Enumeration (CWE) (Dasanayake, et al., 2019). This tool does not suggest any remediation but provides links to relevant Wikipedia articles (Cremer, et al., 2020).

SpotBugs allows the creation of third-party "detectors" to detect additional security bugs. These must be implemented through an API and then have to be compiled into a SpotBugs plugin, which is not that convenient (Cremer, et al., 2020). FindSecBugs is a popular security plugin for SpotBugs.

Despite its IDE integration, it is mostly used to scan after development due to its long scan times, which can take up to 20 minutes (Cremer, et al., 2020). Thus, in practice, this tool is integrated at later parts of the SDLC, which is again not that convenient.

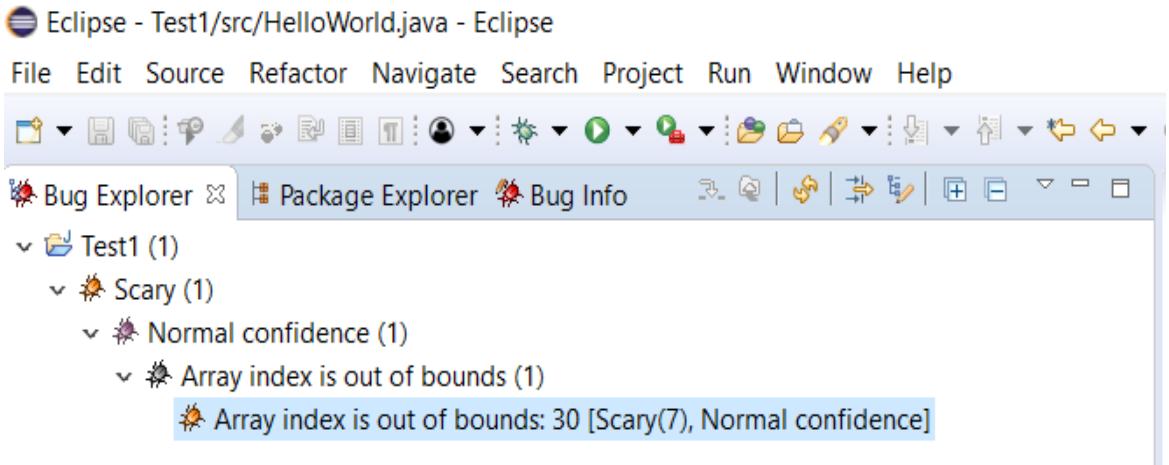


Figure 38: Detection of bugs using SpotBugs plugin (Dasanayake, et al., 2019)

2. SonarQube

SonarQube is a tool in Sonar family. To help perform continuous code inspections of the projects, the tool integrates into existing workflow and detects issues in code. The tool analyses 30+ different programming languages (SonarQube, 2008-2022). Anyway, the outcome of this analysis is reports of code quality measures and issues (instances where 'coding rules' were broken). Yet, what gets analyzed varies depending on the language. For certain languages, a static analysis should be performed on compiled code. (SonarQube, 2008-2022)

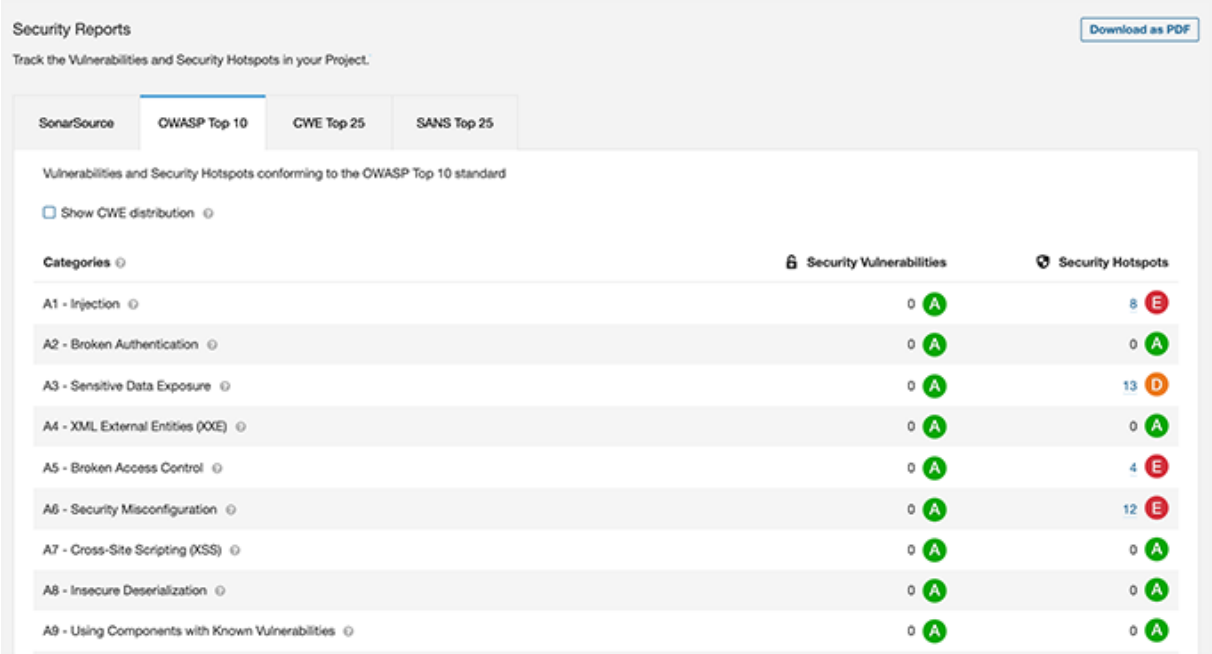


Figure 39: OWASP / CWE security reports by SonarQube (SonarQube, 2008-2022)

Only files that are recognized by that specific edition of the tool are loaded into the project during analysis. *Developer Edition* can analyze branches of the project and pull requests. (SonarQube, 2008-2022)

During analysis, the files provided to the analysis are analyzed, and then the resulting data is sent back as a report, at the end. Then the report is analyzed asynchronously in server-side. (SonarQube, 2008-2022) Tool provides dedicated reports to track Code Security against OWASP Top 10 and CWE Top 25.

To generate issues, SonarQube executes rules on source code. SonarQube LTS 6.7.7 detects a total of 413 rules which are grouped based on type and severity (Lenarduzzi, et al., 2023). There are four types of rules:

- Code Smell (Maintainability domain)
- Bug (Reliability domain)
- Vulnerability (Security domain)
- Security Hotspot (Security domain)

In the Rules page the user can discover all the existing rules or can create new rules based on provided templates. (Anon., 2008-2022)

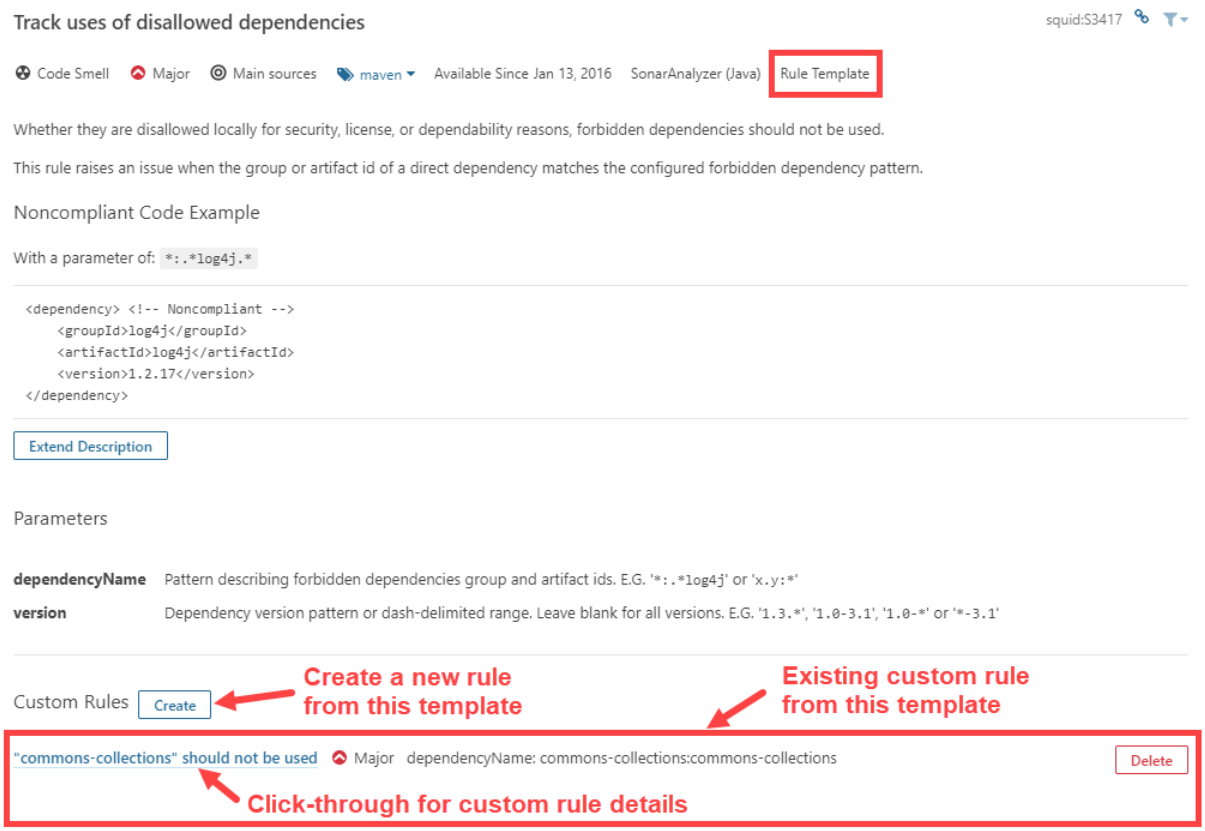


Figure 40: SonarQube – Rule template (Anon., 2008-2022)

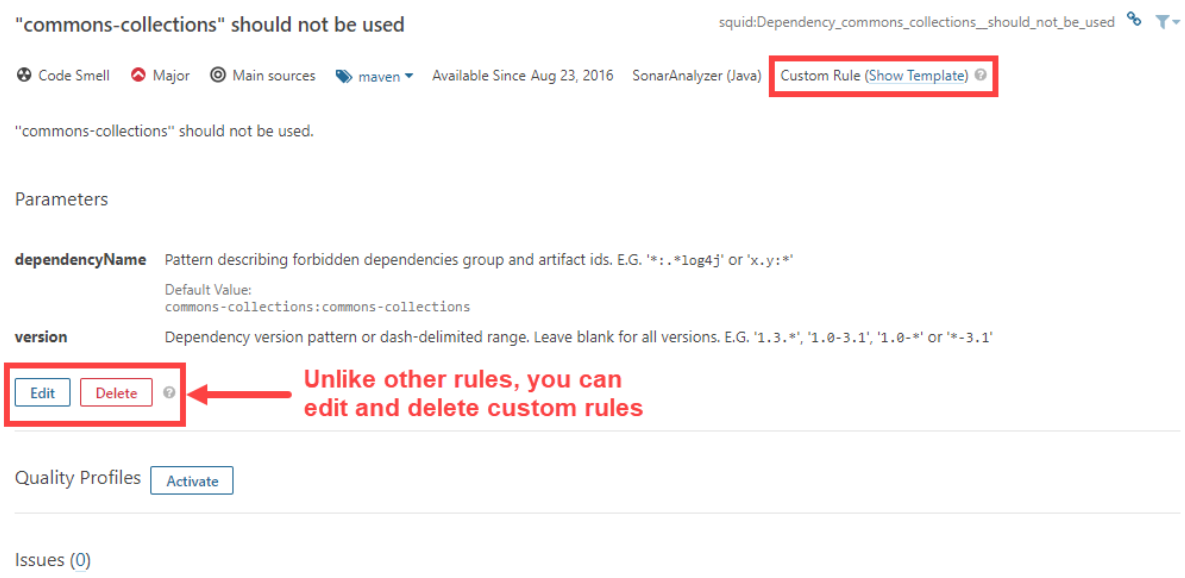


Figure 41: SonarQube - Creating a custom rule using the existing template, by editing the template (Anon., 2008-2022)

3. SonarLint

SonarLint is another tool recently introduced to the Sonar family. With this, Sonar family has 3 main tools namely SonarLint, SonarCloud, and SonarQube (SonarSource, 2008-2022). SonarLint exists as plugins for currently existing major IDEs namely Eclipse, IntelliJ IDEA, Microsoft Visual Studio, VS code and Atom (Anon., 2008-2022) (SonarSource, 2008-2022) It provides on the fly detection of source code quality issues. These issues have been classified into 3 categories as *Vulnerabilities*, *Bugs* and *Code smells*.

SonarLint comprises of several major features such as detecting common mistakes, tricky bugs and known vulnerabilities, offers on the fly instant feedback when the bugs are detected, gives guidance on countermeasures for such bugs, uncovering old issues, provides descriptions for errors that have arose in the source code, etc (Dasanayake, et al., 2019). But the on-the-fly feedback gives only after the Java class is *saved* and not while the user types the code in the IDE (Dasanayake, et al., 2019). Same as SpotBugs, SonarLint also incapable of detecting 'secure coding rule violations' (Dasanayake, et al., 2019) and thus cannot be considered as a solution for adhere SCG.

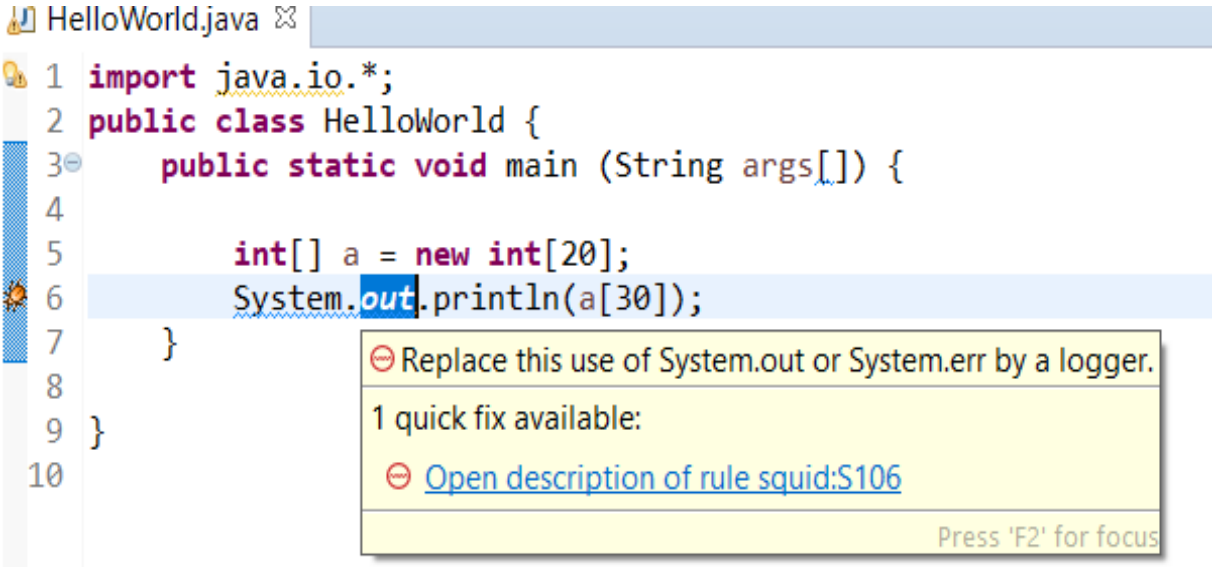


Figure 42: Detection of a code quality issue using the sonarLint plugin (Dasanayake, et al., 2019)

4. Fortify Static Code Analyzer (FSCA)

Fortify Micro Focus Fortify is an ecosystem that embeds application security testing into all stages of the development chain. It has several tools. Some of the tools related to this area are: Fortify Static Code Analyzer (FSCA), Fortify on Demand (FOD), and Fortify Security Assistant (FSA) (Cremer, et al., 2020).

FSCA performs static code analysis on the source code. It can be built into continuous integration and continuous delivery (CI/CD) tools. It supports 25 programming languages. But scanning takes several minutes. The results can be shown in a web interface or in integrations with many tools like bug tracking systems, ticketing systems, and code repositories (Cremer, et al., 2020).

FSCA supports over 1000 vulnerability categories and more than one million APIs. Rule creation should do using their custom Extensible Markup Language (XML) format in a text editor (Cremer, et al., 2020). This requires learning their syntax.

FOD provides similar features to FSCA. But it provides through a web portal (Cremer, et al., 2020).

FSA is a plugin for the IDE. The scan takes several minutes. Thus, during the scan, developer cannot make any code changes (Cremer, et al., 2020). Thus, any of this cannot be taken as a real time solution for adhering secure coding guidelines.

5. Tricorder

This is a program analysis platform integrated into the workflow of developers at Google

(Sadowski, et al., 2015). They support enhancing existing analysis. However, instead of a rule editor, rules must be written using programming languages. The results of this analyzer are shown in a review tool. (Cremer, et al., 2020)

6. Veracode

Veracode has two tools, named Veracode Static Analysis (VSA) and Veracode Greenlight (VG). VSA is a SaaS platform and VG is an IDE plugin. VSA performs static analysis scans on compiled bytecode. This results in analyzing frameworks and libraries used in the project as well. VSA provides integration with some tools and three main IDE namely IntelliJ, Visual Studio, and Eclipse. The tool provides facilities to scan the code and download the results.
Scans will take up to an hour. They provide guiding remediation by providing consultation. The consultant gives advice to the developer on whether the identified issue is a false positive or not. This can take up to three days (Cremer, et al., 2020). This means the feedback cycle is longer than other tools. Thus, this concludes that the tool cannot be taken as a solution for a real-time system. Further, the tool does not support custom rule generations (Cremer, et al., 2020).

7. Checkmarx Static Application Security Testing (CxSAST)

This tool supports a large variety of languages. This also has IDE plugins for Eclipse, Visual Studio, and IntelliJ. Plugins do not perform any local scans. It only allows uploading the source code to CxSAST and then providing the scan results in an interactive way with the IDE.
The tool provides support for extending rules through its extensive Query Language called CxQL (Checkmarx, n.d.). The rule writing tool is independent of the IDE and the scanning tool. Thus, requires long iterations to optimize created rules (Cremer, et al., 2020). Thus, can conclude that their rule-writing technique provides less support for new rules (Concea-Prisăcaru, et al., 2023).

8. Snyk

This is a tool designed to monitor and fix insecure dependencies (Cremer, et al., 2020). Though the tool will not look for vulnerabilities, it will mark dependencies with known vulnerabilities. Thus, this doesn’t cover secure coding guidelines.

9. The OWASP ASIDE/ESIDE

This project consists of two branches. The Application Security IDE (ASIDE) branch focuses on detecting software vulnerabilities. ESIDE branch focuses on helping fresh developers in acquiring secure programming knowledge and practices. Though ASIDE performs fast coding scans in Eclipse, the scans need to be started manually (Cremer, et al., 2020). Thus, can conclude that this tool cannot take as a real-time solution.

Appendix F: Taxonomy of Rule comparison/vulnerability verification methods

Rule verification mechanism	Related work	Limitations / Conclusions
AST with provided limited algorithms	Framework for Secure Coding: An algorithmic approach for real-time detection of secure coding guideline violations	– They have used *Javaparser* as the parser and *Abstract Syntax Tree (AST)* of the code after parsing.
IDE syntax checking features, with analysis techniques (taint analysis, data flow analysis, and control flow analysis)	Sensei - 2nd stage	– To check the rules, their tool reuses the *IDE syntax checking features*. When a developer writes new code, the IDE rebuilds the *Abstract Syntax Tree (AST)* and computes the changes compared to the previous version. – A limited *AST of the changes*, containing the necessary symbol information, is then *passed* on, allowing tools to *only analyze the changes*. – On this AST a combination of specialized light-weight versions of existing analysis techniques are used such as **taint analysis, data flow analysis, and control flow analysis** to verify the rules in real time.

Table 9: Rule comparison/verification methods

Appendix G: Method, class, and package level secure coding guidelines: examples

1. Method level secure coding guidelines example

Consider the below example, from the SEI CERT guidelines.

NUM09-J. Do not use floating-point variables as loop counters.

Description: Floating-point variables must not be used as loop counters (Carnegie Mellon University - Software Engineering Institute, 2018).

Noncompliant Code Example:

This noncompliant code example uses a floating-point variable as a loop counter. The decimal number 0.1 cannot be precisely represented as a float or even as a double (Carnegie Mellon University - Software Engineering Institute, 2018).

```
for (float x = 0.1f; x <= 1.0f; x += 0.1f) {
  System.out.println(x);
}
```

Figure 43: NUM09-J: Noncompliant Code Example

Compliant Solution:

This compliant solution uses an integer loop counter from which the desired floating-point value is derived (Carnegie Mellon University - Software Engineering Institute, 2018):

```
for (int count = 1; count <= 10; count += 1) {
  float x = count/10.0f;
  System.out.println(x);
}
```

Figure 44: NUM09-J: Compliant Solution

2. Class level secure coding guidelines example

Consider the below example, from the SEI CERT guidelines.

NUM03-J. Use integer types that can fully represent the possible range of unsigned data:

Description: The only unsigned primitive integer type in Java is the 16-bit char data type; all of the other primitive integer types are signed. To interoperate with native languages, such as C or C++, that use unsigned types extensively, any unsigned values must be read and stored into a Java integer type that can fully represent the possible range of the unsigned data. For example, the Java long type can be used to represent all possible unsigned 32-bit integer values obtained from native code (Carnegie Mellon University - Software Engineering Institute, 2018).

Noncompliant Code Example:

This noncompliant code example uses a generic method for reading integer data without considering the signedness of the source. It assumes that the data read is always signed and treats the most significant bit as the sign bit. When the data read is unsigned, the actual sign and magnitude of the values may be misinterpreted. (Carnegie Mellon University - Software Engineering Institute, 2018).

```
public static int getInteger(DataInputStream is) throws IOException {
  return is.readInt();
}
```

Figure 45: NUM03-J: Noncompliant Code Example

Compliant Solution:

This compliant solution requires that the values read are 32-bit unsigned integers. It reads an unsigned integer value using the readInt() method. The readInt() method assumes signed values and returns a signed int; the return value is converted to a long with sign extension. (Carnegie Mellon University - Software Engineering Institute, 2018):

```
public static long getInteger(DataInputStream is) throws IOException {
  return is.readInt() & 0xFFFFFFFFL; // Mask with 32 one-bits
}
```

Figure 46: NUM03-J: Compliant Solution

3. Package level secure coding guidelines example

Consider the below example, from the SEI CERT guidelines.

THI00-J. Do not invoke Thread.run()

Description: Thread startup can be misleading because the code can appear to be performing

its function correctly when it is actually being executed by the wrong thread. Invoking the Thread.start() method instructs the Java runtime to start executing the thread's run() method using the started thread. Invoking a Thread object's run() method directly is incorrect. When a Thread object's run() method is invoked directly, the statements in the run() method are executed by the current thread rather than by the newly created thread. Furthermore, if the Thread object was constructed by instantiating a subclass of Thread that fails to override the run() method rather than constructed from a Runnable object, any calls to the subclass's run() method would invoke Thread.run(), which does nothing. Consequently, programs must not directly invoke a Thread object's run() method. (Carnegie Mellon University - Software Engineering Institute, 2018).

Noncompliant Code Example:

This noncompliant code example explicitly invokes run() in the context of the current thread (Carnegie Mellon University - Software Engineering Institute, 2018):

```
public final class Foo implements Runnable {
  @Override public void run() {
    // ...
  }

  public static void main(String[] args) {
    Foo foo = new Foo();
    new Thread(foo).run();
  }
}
```

Figure 47: THI00-J: Noncompliant Code Example

Compliant Solution:

This compliant solution correctly uses the start() method to tell the Java runtime to start a new thread (Carnegie Mellon University - Software Engineering Institute, 2018):

```
public final class Foo implements Runnable {
  @Override public void run() {
    // ...
  }

  public static void main(String[] args) {
    Foo foo = new Foo();
    new Thread(foo).start();
  }
}
```

Figure 48: THI00-J: Compliant Solution

Appendix H: Classification of secure coding rules

Based on the literature, the secure coding rules of each main category have been classified as follows, for the 'level' of the secure coding rule (Dasanayake, et al., 2019).

Main Category	Sub category	Level
Input validation and Data Sanitization (IDS)	IDS01-J	Package
	IDS03-J	Package
	IDS04-J	Package
	IDS06-J	Method
	IDS07-J	Method
	IDS08-J	Method
	IDS11-J	Method
Declarations and Initialization (DCL)	DCL00-J	Class
	DCL01-J	Package
	DCL02-J	Package
Expression (EXP)	EXP00-J	Method
	EXP02-J	Method
	EXP04-J	Package
	EXP05-J	Class
Numeric Types and Operations (NUM)	NUM01-J	Method
	NUM02-J	Method
	NUM03-J	Class
	NUM04-J	Method
	NUM07-J	Method
	NUM09-J	Method
	NUM10-J	Package
	NUM12-J	Method

Characters and Strings (STR)	NUM14-J	Method
	STR00-J	Method
	STR01-J	Method
	STR02-J	Method
	STR03-J	Method/ Package
	STR04-J	Method
Object Orientation (OBJ)	OBJ01-J	Class
	OBJ02-J	Package
	OBJ04-J	Class
	OBJ05-J	Class
	OBJ07-J	Package
	OBJ08-J	Package
	OBJ09-J	Class
	OBJ10-J	Class
Methods (MET)	MET00-J	Method
	MET01-J	Method
	MET02-J	Package
	MET03-J	Class
	MET04-J	Package
	MET05-J	Package
	MET06-J	Package
	MET07-J	Package
	MET08-J	Class
	MET09-J	Class
	MET10-J	Package
	MET12-J	Package

Exceptional Behavior (ERR)	ERR00-J	Method / Package
	ERR01-J	Package
	ERR02-J	Method
	ERR03-J	Method
	ERR04-J	Method
	ERR05-J	Method
	ERR07-J	Method
	ERR09-J	Method
Visibility and Atomicity (VNA)	VNA00-J	Class
	VNA02-J	Class
	VNA05-J	Class
Locking (LCK)	LCK00-J	Class
	LCK01-J	Class
	LCK02-J	Package
	LCK03-J	Package
	LCK04-J	Class
	LCK05-J	Class
	LCK08-J	Class
Thread APIs (THI)	THI00-J	Package
	THI03-J	Class or Package
	THI05-J	Package
Thread Pools (TPS)	TPS00-J	Class
	TPS01-J	Package
Thread-Safety Miscellaneous (TSM)	TSM01-J	Method, class
	TSM02-J	Method
	TSM03-J	Method

Input Output (FIO)	FIO00-J	Method
	FIO01-J	Package
	FIO02-J	Package
	FIO03-J	Package
	FIO04-J	Package
	FIO05-J	Package
	FIO06-J	Package
	FIO07-J	Method
	FIO08-J	Package
	FIO09-J	Package
	FIO10-J	Package
Serialization (SER)	SER00-J	Package
	SER01-J	Package
	SER02-J	Package
	SER03-J	Package
	SER04-J	Method
	SER05-J	Package
	SER09-J	Package
Platform Security (SEC)	SEC02-J	Package
	SEC04-J	Method
	SEC06-J	Package
	SEC07-J	Package
Runtime Environment (ENV)	ENV02-J	Method
	ENV03-J	Method

Appendix I: AIML secure coding rules created for the evaluation

The below SEI CERT secure coding rules are created for evaluation purposes (Carnegie Mellon University - Software Engineering Institute, 2018).

1. THI00-J: Do not invoke Thread.run()

```
<?xml version="1.0" encoding = "UTF-8"?>
<aiml version = "2.0">

    <category>
        <pattern>*class * implements Runnable*</pattern>
        <template>
            <think>
                <set name="runnableClassName"><star index="2"/></set>
                <set name="runnableInstanceName"></set>
                <set name="calledRun">false</set>
                <set name="invokedThreadRun">false</set>
            </think>
        </template>
    </category>
<category>
    <pattern>* * = new *\(\);</pattern>
    <template>
        <think>
            <set name="className"><star index="3"/></set>
            <set name="instanceName"><star index="2"/></set>
        </think>

        <condition name="className" value="runnableClassName">
            <think><set name="runnableInstanceName"><get name="instanceName"/></set></think>
        </condition>
    </template>
</category>
    <category>
        <pattern>* Thread\(*\).run\(\)*</pattern>
        <template>
            <think>
                <set name="threadRunnableName"><star index="2"/></set>
                <set name="calledRun">true</set>
            </think>

            <condition name="threadRunnableName" value="runnableInstanceName">
                <condition name="calledRun">
                    <li value="true">
                        <think><set name="invokedThreadRun">true</set></think>
                    </li>
                </condition>
            </condition>

            <condition name="invokedThreadRun">
                <li value="true">
                    THI00-J ,Correction: Please call start method instead of run method
                </li>
            </condition>
        </template>
    </category>
</aiml>
```

Figure 49: AIML example rule 1

2. SEC01-J: Do not allow tainted variables in privileged blocks

```
<?xml version="1.0" encoding = "UTF-8"?>
<aiml version = "2.0">

    <category>
        <pattern>* * *\(final String *\)</pattern>
        <template>
            <think>
                <set name="finalStringParameter"><star index="4"/></set>
                <set name="PrivilegedBlock">false</set>
                <set name="PrivilegedAction">false</set>
                <set name="PrivilegedActionReturnedUnsafe">false</set>
                <set name="pathOfReturnedFileInput"></set>
            </think>
        </template>
    </category>

    <category>
        <pattern>* AccessController.doPrivileged^\(^</pattern>
        <template>
            <think>
                <set name="PrivilegedBlock">true</set>
            </think>
        </template>
    </category>
  <category>
      <pattern>^new^PrivilegedExceptionAction\(^\)^</pattern>
      <template>
          <think>
              <set name="PrivilegedAction">true</set>
          </think>
      </template>
  </category>
    <category>
        <pattern>return^new^FileInputStream\(*\);^</pattern>
        <template>
            <think>
                <set name="pathOfReturnedFileInput"><star index="3"/></set>
            </think>

            <condition name="pathOfReturnedFileInput" value="finalStringParameter">
                <condition name="PrivilegedAction">
                    <li value="true">
                        <think><set name="PrivilegedActionReturnedUnsafe">true</set></think>
                    </li>
                </condition>
            </condition>

            <condition name="PrivilegedActionReturnedUnsafe" value="PrivilegedBlock">
                <li value="true">
                    SEC01-J ,Correction: Please clean file name and path:
                    '<get name="finalStringParameter"/>' before creating FileInputStream
                    inside Privileged block
                </li>
            </condition>
        </template>
    </category>

</aiml>
```

Figure 50: AIML example rule 2

3. NUM10-J: Do not construct BigDecimal objects from floating-point literals

```
<?xml version="1.0" encoding = "UTF-8"?>
<aiml version = "2.0">
    <category>
        <pattern>*new^BigDecimal^\(^"*"^\)*</pattern>
        <template>NUM10-J</template>
    </category>
</aiml>
```

Figure 51: AIML example rule 3

4. SEC07-J: Call the superclass's getPermissions() method when writing a custom class loader

```
<?xml version="1.0" encoding = "UTF-8"?>
<aiml version = "2.0">

    <category>
        <pattern>*class * extends *</pattern>
        <template>
            <think>
                <set name="extendedClass">true</set>
                <set name="getPermissionsOverridden">false</set>
                <set name="createdNewPermissions">false</set>
                <set name="violatedSEC07_J">false</set>
            </think>
        </template>
    </category>

    <category>
        <pattern>^protected^PermissionCollection^getPermissions^\(^CodeSource*\)*</pattern>
        <template>
            <think>
                <set name="getPermissionsOverridden">true</set>
            </think>
        </template>
    </category>
```

```
    <category>
        <pattern>*=^new ^Permissions^\(^\)*</pattern>
        <template>
            <think>
                <set name="createdNewPermissions">true</set>
            </think>

            <condition name="extendedClass">
                <li value="true">
                    <think>
                        <condition name="getPermissionsOverridden">
                            <li value="true">
                                <think>
                                    <condition name="createdNewPermissions">
                                        <li value="true">
                                            <think><set name="violatedSEC07_J">true</set></think>
                                        </li>
                                    </condition>
                                </think>
                            </li>
                        </condition>
                    </think>
                </li>
            </condition>

            <condition name="violatedSEC07_J">
                <li value="true">
                    SEC07-J,Correction: Please call super.getPermissions() method
                    instead of calling Permissions constructor
                </li>
            </condition>
        </template>
    </category>
</aiml>
```

Figure 52: AIML example rule 4

5. FIO02-J: Detect and handle file-related errors

```
<?xml version="1.0" encoding = "UTF-8"?>
<aiml version = "2.0">

    <category>
        <pattern>File * = new File\(^\);</pattern>
        <template>
            <think>
                <set name="fileVar"><star index="1"/></set>
                <set name="safeDeleted">false</set>
                <set name="fileUnsafeDelete">false</set>
                <set name="deletedObjectName"></set>
            </think>
        </template>
    </category>

    <category>
        <pattern>^if^\(\!*\.delete\(^\)\)^</pattern>
        <template>
            <think>
                <set name="deletedObjectName"><star index="1"/></set>
                <set name="safeDeleted">true</set>
            </think>
        </template>
    </category>
```

```
    <category>
        <pattern>*.delete^\(^\)^;</pattern>
        <template>
            <think>
                <set name="deletedObjectName"><star index="1"/></set>
                <set name="safeDeleted">false</set>
            </think>

            <condition name="deletedObjectName" value="fileVar">
                <condition name="safeDeleted">
                    <li value="false">
                        <think><set name="fileUnsafeDelete">true</set></think>
                    </li>
                </condition>
            </condition>

            <condition name="fileUnsafeDelete">
                <li value="true">
                    FIO02-J ,Correction: Please check the output value of
                    file delete: '<get name="deletedObjectName"/>'
                </li>
            </condition>
        </template>
    </category>
</aiml>
```

Figure 53: AIML example rule 5

REFERENCES

[1]. The OWASP Foundation, 2017. *OWASP Top Ten.* [Online] Available at: https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf [Accessed 23 09 2021].

[2]. Abeyrathna, A. et al., 2020. A security specific knowledge modelling approach for secure software engineering. *Journal of the National Science Foundation of Sri Lanka,* 48(1), pp. 93-98.

[3]. Ahmed, I. & Singh, S., 2015. AIML Based Voice Enabled Artificial Intelligent Chatterbot. *IJUNESST,* 28 02, Volume 8, pp. 375-384.

[4]. AIML Foundation, 2018. *AIML Docs.* [Online] Available at: http://www.aiml.foundation/doc.html [Accessed 11 11 2023].

[5]. Alenezi, M. & Almuairfi, S., 2019. Security Risks in the Software Development Lifecycle. *International Journal of Recent Technology and Engineering (IJRTE),* 8(3), pp. 7048-7055.

[6]. Alwan, A. & Andersson, J., 2022. *Secure Application Development.* s.l.:s.n.

[7]. Anon., 2008-2022. *SonarQube docs 9.6 - Rules.* [Online] Available at: https://docs.sonarqube.org/latest/user-guide/rules/ [Accessed 8 9 2022].

[8]. Anon., 2011. *Google Code Archive - Free A.L.I.C.E. AIML Set.* [Online] Available at: https://code.google.com/archive/p/aiml-en-us-foundation-alice/ [Accessed 14 11 2022].

[9]. Armerding, T., 2021. *What is the cost of poor software quality in the U.S.?,* s.l.: Synopsis.

[10]. Barnisin, M., 2022. *AIML 2.1 Chatbot Design Language Interpreter in Python.* Brno, Spring: Department of Machine Learning and Data Processing - Faculty Of Informatics - Masaryk University.

[11]. Barnišin, M., 2022. *pyaiml21 documentation.* [Online] Available at: https://pyaiml21.readthedocs.io/en/latest/ [Accessed 12 11 2023].

[12]. batiaev, 2019. *AIMLang - aiml-java-interpreter.* [Online] Available at: https://github.com/AIMLang/aiml-java-interpreter/blob/master/core/src/main/java/org/aimlang/core/consts/WildCard.java [Accessed 28 01 2024].

[13]. Baxter, I. D. et al., 1998. *Clone detection using abstract syntax trees.* Bethesda, Mayland, IEEE Computer Society, pp. 368-377.

[14]. Ben-Kiki, O., Evans, C. & Ingerson, B., 2005. YAML Ain't Markup Language (YAML™) Version 1.1. 18 1.

[15]. Ben-Kiki, O., Evans, C. & Net, I. d., 2009. *YAML Ain't Markup Language (YAML™).* [Online] Available at: https://www.earthdata.nasa.gov/s3fs-public/imported/YAML%201.2%20Spec.pdf [Accessed 20 10 2022].

[16]. Carnegie Mellon University - Software Engineering Institute, 2018. *SEI CERT Oracle Coding Standard for Java.* [Online] Available at: https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java [Accessed 04 05 2022].

[17]. Carnegie Mellon University - Software Engineering Institute, 2023. *IDS01-J. Normalize strings before validating them.* [Online] Available at: https://wiki.sei.cmu.edu/confluence/display/java/IDS01-J.+Normalize+strings+before+validating+them [Accessed 09 02 2024].

[18]. Checkmarx, n.d. *Attachments.* [Online] Available at: https://checkmarx.atlassian.net/wiki/pages/viewpageattachments.action?pageId=5406733 [Accessed 13 10 2022].

[19]. Chiu, T., 2020. *How to Find Additional Hidden Vulnerabilities During DAST Testing.* [Online]
Available at: https://www.k2io.com/how-to-find-additional-hidden-vulnerabilities-during-dast-testing/
[Accessed 5 5 2022].

[20]. Concea-Prisăcaru, A.-I., Nițescu, T.-A. & Sgârciu, V., 2023. SDLC and the importance of software security. *U.P.B. Sci. Bull., Series C,* 85(1), pp. 117-130.

[21]. Cremer, P. D., Desmet, N., Madou, M. & Sutter, B. D., 2020. Sensei: Enforcing Secure Coding Guidelines in the IDE. *Software: Practice and Experience,* 50(9), p. 1682–1718.

[22]. Dagiene, V., Gudoniene, D. & Burbaite, R., 2015. Semantic Web Technologies for e-Learning: Models and Implementation. *Informatica (Netherlands),* 26(2), pp. 221-240.

[23]. Dasanayake, S. L., Mudalige, A. & Perera, M. L. T., 2019. *Framework for Secure Coding: An algorithmic approach for real-time detection of secure coding guideline violations,* Colombo: University of Colombo School of Computing.

[24]. Dawson, M., Burrell, D. N., Rahim, E. & Brewster, S., 2010. Integrating software assurance into the software development life cycle (SDLC). *Journal of information systems technology & planning,* 3(6), pp. 49-53.

[25]. ENISA - European Union Agency for Cybersecurity, 2018. *ENISA - European Union Agency for Cybersecurity.* [Online]
Available at: https://www.enisa.europa.eu/publications/info-notes/is-software-more-vulnerable-today
[Accessed 5 May 2022].

[26]. Eriksson, M. & Hallberg, V., 2011. *School of Computer Science and Engineering - Royal Institute of Technology.* [Online]
Available at: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=636a2b04d98c0af8e9d6f59148352dd63af4f0c1
[Accessed 8 11 2023].

[27]. Fernandez, E. B. & Mujica, S., 2011. Model-based development of security requirements. *CLEI Electronic Journal - On-line ISSN 0717-5000,* 14(3).

[28]. Gasiba, T. E., Lechner, U., Pinto-Albuquerque, M. & Fernandez, D. M., 2021. Awareness of Secure Coding Guidelines in the Industry - A first data analysis. *arXiv,* 6 1.1(2101.02085).

[29]. Gasiba, T. E., Lechner, U., Pinto-Albuquerque, M. & Mendez, D., 2021. *Is Secure Coding Education in the Industry Needed? An Investigation Through a Large Scale Survey.* s.l., Software Engineering Education and Training (ICSE-SEET).

[30]. Gedeon, T. et al., 2019. Code Summarization with Abstract Syntax Tree. *Neural Information Processing,* Volume 1143, pp. 652-660.

[31]. GitLab, 2019. *2019 Global Developer Report: DevSecOps finds security roadblocks divide teams.* [Online] Available at: https://about.gitlab.com/blog/2019/07/15/global-developer-report/ [Accessed 12 10 2022].

[32]. Goldstein, A., 2021. *All About WhiteSource's 2021 Open Source Security Vulnerabilities Report.* [Online] Available at: https://www.whitesourcesoftware.com/resources/blog/2021-state-of-open-source-security-vulnerabilities-cheat-sheet/# [Accessed 5 May 2022].

[33]. Google, 2013. *Google Code Archive - Program AB.* [Online] Available at: https://code.google.com/archive/p/program-ab/ [Accessed 14 11 2022].

[34]. Google, 2014. *Google Code Archive - ALICE 2.0.* [Online] Available at: https://code.google.com/archive/p/aiml-en-us-foundation-alice2/ [Accessed 14 11 2022].

[35]. Hanif, H. & Maffeis, S., 2022-07. *VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection.* s.l., IEEE, pp. 1-8.

[36]. Humayun, M., Jhanjhi, N., Almufareh, M. F. & Khalil, M. I., 2022. Security Threat and Vulnerability Assessment and Measurement in Secure Software Development. *Computers, Materials & Continua,* 71(3), pp. 5039-5059.

[37]. Jacob, J., 2006. *A Rule Markup Language and Its Application to UML.* Berlin, Heidelberg, Springer, pp. 26-41.

[38]. Jacob, J. F., 2008. Chapter 2 - RML and its application to UML. In: *Domain specific modeling and analysis.* s.l.:s.n., pp. 23-40.

[39]. Khakpour, N. et al., 2018. *Synthesis of a Permissive Security Monitor.* Cham, Springer International Publishing, p. 48–65.

[40]. Khan, R. . A., Khan, S. U., Khan, H. U. & Ilyas, M., 2022. Systematic Literature Review on Security Risks and Its Practices in Secure Software Development. *IEEE Access*, 14 January, Volume 10, pp. 5456-5481.

[41]. Khin, N. N. & Soe, K. M., 2020 February. *University Chatbot using Artificial Intelligence Markup Language.* Yangon, Myanmar, IEEE, pp. 1-5.

[42]. Khin, N. N. & Soe, K. M., 2020. University Chatbot using Artificial Intelligence Markup Language. *2020 IEEE Conference on Computer Applications, ICCA 2020,* 27-28 February.

[43]. Lenarduzzi, V. et al., 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software,* April.Volume 198.

[44]. Lipner, S., 2014. The trustworthy computing security development lifecycle. *Annual Computer Security Applications Conference, ACSAC*, pp. 2-13.

[45]. Lomio, F. et al., 2022. Just-in-time software vulnerability detection: Are we there yet?. *Journal of Systems and Software,* 01 06, Volume 188, p. 111283.

[46]. Malvisi, F., 2014. *Development of a Framework for AIML Chatbots inHTML5 and Javascript.* s.l.:s.n.

[47]. Marietto, M. d. G. B. et al., 2013. *Artificial Intelligence MArkup Language: A Brief Tutorial.* [Online] Available at: http://arxiv.org/abs/1307.3091 [Accessed 21 10 2022].

[48]. McGhee, E., Krobatsch, T. & Milton, S., July 2022. *NetInfra - A Framework for Expressing Network Infrastructure as Code.* Boston, MA, USA, ACM Digital Library, pp. 1-7.

[49]. McGraw, G., 2005. Chapter 1. Defining a Discipline. In: G. McGraw, ed. *Software Security - Building security in.*. Cambridge, MA: Addison-Wesley Professional, pp. 24-58.

[50]. McGraw, G., 2005. Chapter 12. A Taxonomy of Coding Errors. In: G. McGraw, ed. *Software Security - Building security in.* Cambridge, MA: Addison-Wesley Professional, pp. 273 - 295.

[51]. McGraw, G., 2005. Chapter 3. Introduction to Software Security. In: G. McGraw, ed. *Software Security - Building security in.* Cambridge, MA: Addison-Wesley Professional, pp. 100-118.

[52]. McGraw, G., 2005. Chapter 6. Software Penetration Testing. In: G. McGraw, ed. *Software Security - Building security in.* Cambridge, MA: Addison-Wesley Professional, pp. 180-192.

[53]. McGuire, M., 2022. *What is the cost of poor software quality in the U.S.?.* [Online]
Available at: https://www.synopsys.com/blogs/software-security/poor-software-quality-costs-us/
[Accessed 28 02 2023].

[54]. Mehla, S. & Jain, S., 2019. *Rule Languages for the Semantic Web.* Singapore, Springer, pp. 825-834.

[55]. Microsoft, 2024. *Security Engineering.* [Online] Available at: https://www.microsoft.com/en-us/securityengineering/sdl/about [Accessed 04 03 2024].

[56]. Neamtiu, I., Foster, J. S. & Hicks, M., 2005. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. 17 May.

[57]. Ochodek, M. et al., 2020. Recognizing lines of code violating company-specific coding guidelines using machine learning. *Empirical Software Engineering*, 01 01, 25(1), pp. 220-265.

[58]. ORACLE, 2023. *Secure Coding Guidelines for Java SE.* [Online] Available at: https://www.oracle.com/java/technologies/javase/seccodeguide.html [Accessed 03 03 2024].

[59]. OWASP, 2010. *Secure Coding Guidelines.* [Online] Available at: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf [Accessed 03 03 2024].

[60]. Petrosyan, A., 2023. *Common IT vulnerabilities and exposures worldwide 2009-2023,* s.l.: Statista.

[61]. Rasheed, S., Dietr, J. & Tahir, A., 2019. *Laughter in the Wild: A Study into DoS Vulnerabilities in YAML Libraries.* s.l., IEEE, pp. 342-349.

[62]. RuleML, 2023. *RuleML.* [Online] Available at: https://www.ruleml.org/ [Accessed 23 02 2024].

[63]. Sadowski, C. et al., 2015. *Tricorder: Building a Program Analysis Ecosystem.* s.l., s.n.

[64]. Satu, M. S., Parvez, M. H. & Shamim-Al-Mamun, 2015. *Review of integrated applications with AIML based chatbot.* s.l., IEEE Xplore, pp. 87-90.

[65]. Secure Code Warrior, 2000-2022. *Sensei by Secure Code Warrior - IDE plugin.* [Online] Available at: https://plugins.jetbrains.com/plugin/14015-sensei-by-secure-code-warrior-/ [Accessed 14 10 2022].

[66]. Secure Code Warrior, 2019-2021. *Targets.* [Online] Available at: https://docs.sensei.securecodewarrior.com/ref/targets.html [Accessed 21 02 2024].

[67]. Shippey, T. J., 2015. Exploiting Abstract Syntax Trees To Locate Software Defects. 7 May.

[68]. Siavvas, M., Gelenbe, E., Kehagias, D. & Kehagias, D., 2018. *Static Analysis-Based Approaches for Secure Software Development.* s.l., Springer, Cham, pp. 142 - 157.

[69]. Silva, D. & Mendonça, D., 2021. *SCPL: A Markup Language for Source Code Patterns Localization.* New York, NY, USA, Association for Computing Machinery, p. 127–132.

[70]. Software Assurance Forum for Excellence in Code (SAFECode), 2018. *Fundamental Practices for Secure Software Development: Essential Elements of a Secure Development Lifecycle Program Third Edition Fundamental Practices for Secure Software Development,* s.l.: Software Assurance Forum for Excellence in Code (SAFECode).

[71]. SonarQube, 2008-2022. *Code Security,.* [Online] Available at: https://www.sonarqube.org/features/security/ [Accessed 8 9 2022].

[72]. SonarQube, 2008-2022. *SonarQube docs 9.6 - Analyzing source code - overview.* [Online] Available at: https://docs.sonarqube.org/latest/analysis/overview/ [Accessed 8 9 2022].

[73]. SonarQube, 2008-2022. *SonarQube docs 9.6 Language overview.* [Online] Available at: https://docs.sonarqube.org/latest/analysis/languages/overview/ [Accessed 8 September 2022].

[74]. SonarSource, 2008-2022. *SonarLint - What's new.* [Online] Available at: https://www.sonarsource.com/products/sonarlint/whats-new/ [Accessed 13 10 2022].

[75]. SonarSource, 2008-2022. *SonarSource static code analysis.* [Online] Available at: https://rules.sonarsource.com/ [Accessed 13 10 2022].

[76]. spobugs, 2016-2022. *spotbugs documentation - Introduction.* [Online] Available at: https://spotbugs.readthedocs.io/en/stable/introduction.html [Accessed 12 10 2022].

[77]. Stefanovska, Z., Jakimoski, K. & Stefanovski, W., 2022. Optimization of Secure Coding Practices in SDLC as Part of Cybersecurity Framework. *Journal of Computer Science Research*, 01 04, Volume 4, pp. 31-41.

[78]. Sterling, K., 2021. *Program-Y.* [Online] Available at: https://github.com/keiffster/program-y/wiki/ [Accessed 12 11 2023].

[79]. Synopsys Editorial Team, 2020. *What is the secure software development life cycle (SDLC)? | Synopsys.* [Online] Available at: https://www.synopsys.com/blogs/software-security/secure-sdlc/ [Accessed 29 04 2022].

[80]. Synopsys, 2019. *Eclipse IDE for Java.* [Online] Available at: https://www.openhub.net/p/eclipse/analyses/latest/languages_summary [Accessed 24 9 2021].

[81]. synopsys, 2019. *Secure Coding Guidelines.* [Online] Available at: https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/secure-coding-guidelines-datasheet.pdf [Accessed 14 January 2022].

[82]. Synopsys, 2022. *[Analyst Report] 2022 The Cost of Poor Quality Software.* [Online] Available at: https://www.synopsys.com/software-integrity/resources/analyst-reports/cost-poor-quality-software.html [Accessed 03 09 2023].

[83]. Synopsys, 2023. Open Source Security and Risk Analysis 2023. *Open Source Security and Risk Analysis report,* April.

[84]. Tantithamthavorn, C., Ihara, A., Hata, H. & Matsumoto, K., 2014. Impact Analysis of Granularity Levels on Feature Location Technique. In: D. Z. a. Z. Jin, ed. *Requirements Engineering.* New York: Springer-Verlag, pp. 135-149.

[85]. Tantithamthavorn, C., Ihara, A., Hata, H. & Matsumoto, K., 2014. *Impact Analysis of Granularity Levels on Feature Location Technique.* Berlin, Heidelberg, Springer, pp. 135-149.

[86]. Tricentis, 2017. *The Software Fail Watch: 2016 in Review.* [Online] Available at: https://www.tricentis.com/blog/1-1-trillion-in-assets-impacted-by-software-defects-a-software-testing-fail/ [Accessed 14 January 2022].

[87]. Veracode, n.d. *Open Source Risk.* [Online] Available at: https://www.veracode.com/sites/default/files/pdf/resources/ipapers/everything-you-need-to-know-open-source-risk/index.html [Accessed 08 11 2023].

[88]. W3C, 2013. *W3C Recommendation 26 November 2008.* [Online] Available at: https://www.w3.org/TR/xml/ [Accessed 22 02 2024].

[89]. Wallace, R. S., 2003. *The Elements of AIML Style,* s.l.: ALICE A. I. Foundation.

[90]. Wallace, R. S., 2009. The Anatomy of A.L.I.C.E.. In: R. Epstein, G. Roberts & G. Beber, eds. *Parsing the Turing Test: Philosophical and Methodological Issues in the Quest for the Thinking Computer*. Dordrecht: Springer Netherlands, pp. 181-210.

[91]. Wartschinski, L. et al., 2022. VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python. *Information and Software Technology,* 01 04, Volume 144, p. 106809.

[92]. Welty, C., 1997. *Augmenting abstract syntax trees for program understanding.* Incline Village, NV, USA, IEEE Computer Society, pp. 126-133.

[93]. Wijesiriwardana, C. et al., 2020. Secure Software Engineering: A Knowledge Modeling based Approach for Inferring Association between Source Code and Design Artifacts. *International Journal of Advanced Computer Science and Applications,* 11(12), pp. 708-716.

[94]. Wu, Y. et al., July 5, 2022. *VulCNN: an image-inspired scalable vulnerability detection system.* New York, NY, USA, Association for Computing Machinery.

[95]. YAML Org, 2021. *YAML Ain't Markup Language (YAML™) version 1.2 revision 1.2.2.* [Online] Available at: https://yaml.org/spec/1.2.2/ [Accessed 11 11 2023].

[96]. YAML, 2001-2009. *YAML Ain't Markup Language (YAML™) version 1.2 - Revision 1.2.2 (2021-10-01).* [Online]

Available at: https://yaml.org/spec/1.2.2/ [Accessed 14 10 2022].

[97]. YAML, 2009-2022. *YAML: YAML Ain't Markup Languag.* [Online] Available at: https://yaml.org/ [Accessed 14 10 2022].

[98]. YAML, n.d. *A New YAML Specification.* [Online] Available at: https://yaml.com/blog/2021-10/new-yaml-spec/ [Accessed 4 10 2022].