JJEE
Volume 9 | Number 2 | June 2023 | Pages 125-148

Jordan Journal of Electrical Engineering

ISSN (print): 2409-9600, ISSN (online): 2409-9619

Homepage: jjee.ttu.edu.jo

Software Code Bloats and Security Identification Model Based on Mikado Methodology: a Refactoring Practice

Taghi Javdani Gandomani[1*], Hamid Shabani Sichani[2], Behzad Soleimani Neysiani[3]

[1] Department of Computer Science, Shahrekord University, Iran
E-mail: javdani@sku.ac.ir
[2, 3] Department of Computer Engineering, Isfahan (Khorasgan) Branch, Islamic Azad University, Isfahan, Iran

Received: November 02, 2022 *Revised: January 01, 2023* *Accepted: January 06, 2023*

***Abstract*—** The term "code smell" or "bad smell" refers to a code that has been written incorrectly and reflects severe defects in software design. Some code smells cause, particularly, security vulnerabilities in software codes. Until now, identification of these codes is mainly done through software tools and not by process methods or models. Based on the Mikado methodology, this paper proposes a model that uses a syntax-metric parser engine to detect insecure software code bloats and security vulnerabilities. This model, named Touba, assesses and analyzes the discovered cases and provides an interactive method for code review and statistical analysis. Employing the proposed model in testing the Juliet Test Suites shows its outstanding performance in terms of the selected measures of precision, recall, and F-measure. The obtained results show that the proposed model has a better performance - compared to the existing tools - in terms of accuracy by 20.3%, recall by 16.76%, and F-measure by 18.61% on average. These results indicate the effectiveness of the proposed - security vulnerability identification - model as the main contribution of this investigation.

***Keywords*—** Code smell; Software security vulnerabilities; Refactoring; Mikado method.

1. INTRODUCTION

Static code analysis, aiming to detect and identify the vulnerabilities of the programming codes, is a process that has evolved in the past years and attracted the attention of software engineers and scientists. Meanwhile, various tools have been developed and used for static code analysis. However, most of these tools have considered codes written in Java language [1]. As a result of time and financial constraints and customer pressures, the software teams do not put enough accuracy into programming the required codes. The developed codes are subject to defects, resulting in severe problems in the future. Security problems, high maintenance costs, and code evolution are the outcomes of coding defects. Thus, considering the analysis of programming codes to detect the existing defects is very important. Software engineers take advantage of "code smell" as a sign to detect such defects [2].

In static code analysis, the presence of code smell indicates a defect in a source code under review. Different code smells represent different defects [3]. Thus, various classes have been defined for the code smells. Many researchers have investigated detecting, classifying, and resolving code smells. Security code smells are one of the important cases that can be

* Corresponding author
Article's DOI: 10.5455/jjee.204-1667422472

used to detect and identify the security defects of source codes. However, a few studies have studied identifying security defects using code smells. It seems that the introduced tools such as CodePro, FindBugs, and FindSecurityBugs are not very efficient. Also, in these studies, a proper methodology has not been used to identify the code smells [4].

In terms of code smell, the basis of bloaters' behavior and inherent nature are achieved by the mechanism of bloating, by leaving huge footprints during installation, extravagant use of system resources and providing useless features that users do not use. In computer programming, a bloater is the generation of source code or machine code that is unnecessarily verbose (large) and slow to infer, or in short, wastes resources. For this reason, there are security vulnerabilities that can be classified as bloating in terms of the type of behavior they exhibit. Therefore, assuming that it is done, in the continuation of the article, we can understand the relationship between different types of security breaches with code smells, especially the smell of the Bloaters type. Therefore, it is not possible to ignore the worries and problems caused by Bloaters, such as increasing the size of the program, inserting static code, excessive consumption of RAM and CPU, insecure holes, etc.

The research tries to answer the ambiguities and security issues. This study aims to resolve the above challenge (detecting and evaluating the security vulnerabilities and bloats). In this study, the syntax-metric parser engine, which is called the Touba security bloat mistakes detector hereafter, and comprises three steps of data pre-processing, primary analysis, and secondary analysis, is used. Finally, the steps are adapted to the Mikado method for modeling an agile method. Mikado is a simple method that is mainly used by software teams to improve their software codes in an iterative manner in refactoring phase. This method helps them to find hidden problems, vulnerabilities, and code smells.

The rest of this paper is organized as follows. Section 2 presents various code smell classes along with their details and the Mikado method. The literature review is presented in section 3. Section 4 presents the proposed method and related measures in the context of security vulnerabilities and bloats. Section 5 evaluates the proposed approach. Section 6 discusses the research constraints. Section 7 presents future suggestions, and finally, the paper is concluded in section 8.

2. BACKGROUND

The evolution of software products is one of the important challenges for software teams after the completion of the software development process. The generated codes should be reviewed to identify and resolve their problems and challenges due to various reasons. To detect the problems with the software codes, various techniques and methods are used [5]. Using the code smell is a technique to identify the problems and defects of software codes. Using code smell detection techniques can simplify the process of identifying the defects of software codes. Code smells exist in different forms, and each one can refer to a specific defect. In addition, there are various tools and methodologies in this context that try to simplify the process of identifying the defects of the software codes. In the following, the code classification and the Mikado method, one of the most popular methods in this context, are investigated.

2.1. Code Smell Classification

The literature review shows that previous studies have presented different forms of code smells. In one of these studies, Fowler and Beck [6] introduced 22 types of bad code smells. Since classification makes smells more understandable and represents the relationships between the smells more clearly, some studies have classified these smells. In one of the most famous studies, seven different units have been introduced for the code smells, including bloaters, object orientation abusers, change preventers, dispensible, encapsulators, couplers, and other smells [7]. Table 1 represents those units.

Table 1. Units of the code smells.

Unit	Code smells involved
Bloaters	Long method, large class, data clumps, primitive obsession, and long parameter list
Object orientation abusers	Switch statements, temporary field, refused bequest, and alternative classes with different interfaces
Change preventers	Divergent change, shotgun surgery, and parallel inheritance hierarchies
Dispensables	Lazy class, data class, duplicate code, dead code, and speculative generality
Encapsulators	Message chains and middle man
Couplers	Feature envy and inappropriate intimacy
Others	Incomplete library class and comments

According to Table 1, Bloaters represent something in the code that has grown so large that it cannot be effectively managed. The type of Object Orientation Abusers includes those in which the system does not take full advantage of object-oriented design capabilities. A common origin of this problem is programmers having prior experience in procedural programming and lack of training or understanding of object-oriented programming. Change Preventers is related to code structures, which significantly prevents software modification. Dispensables indicate unnecessary code that should be removed from the code. Classes that are not doing enough need to be removed or their responsibility needs to be increased. The Encapsulators deal with data communication mechanisms or encapsulation. The sixth category is Couplers smells which occur because of coupling issues in the code are included in this category. The other category contains the two remaining smells Incomplete Library Class, and Comments that do not fit into any of the categories [8].

2.2. The Mikado Method

The name of this method is adopted from the Mikado game. Mikado originated in Europe and is a game in which wooden bars are selected [9]. This method includes a process that improves the codes gradually, aiming to resolve the defects of the software codes. It has a suitable position in the static analysis of software codes due to its special characteristics. This method is of interest to experts in the refactoring of software codes. Table 2 describes some of its properties. Also, Fig. 1 shows the main steps of the Mikado method.

Table 2. Characteristics of the Mikado method.

No.	Characteristics
1	It fits nicely in an incremental process
2	It is very lightweight (pen and paper, or whiteboard)
3	It increases the visibility of the work.
4	It provides stability to the codebase while you are changing it.
5	It supports continuous deployments by finding a nondestructive change path.
6	It improves communication between people.
7	It enhances learning.
8	It aids reflection on the work done.
9	It leverages different competencies, abilities, and knowledge
10	It helps collaboration within a team.
11	It scales by enabling the distribution of the workload over the team
12	It is easy to use

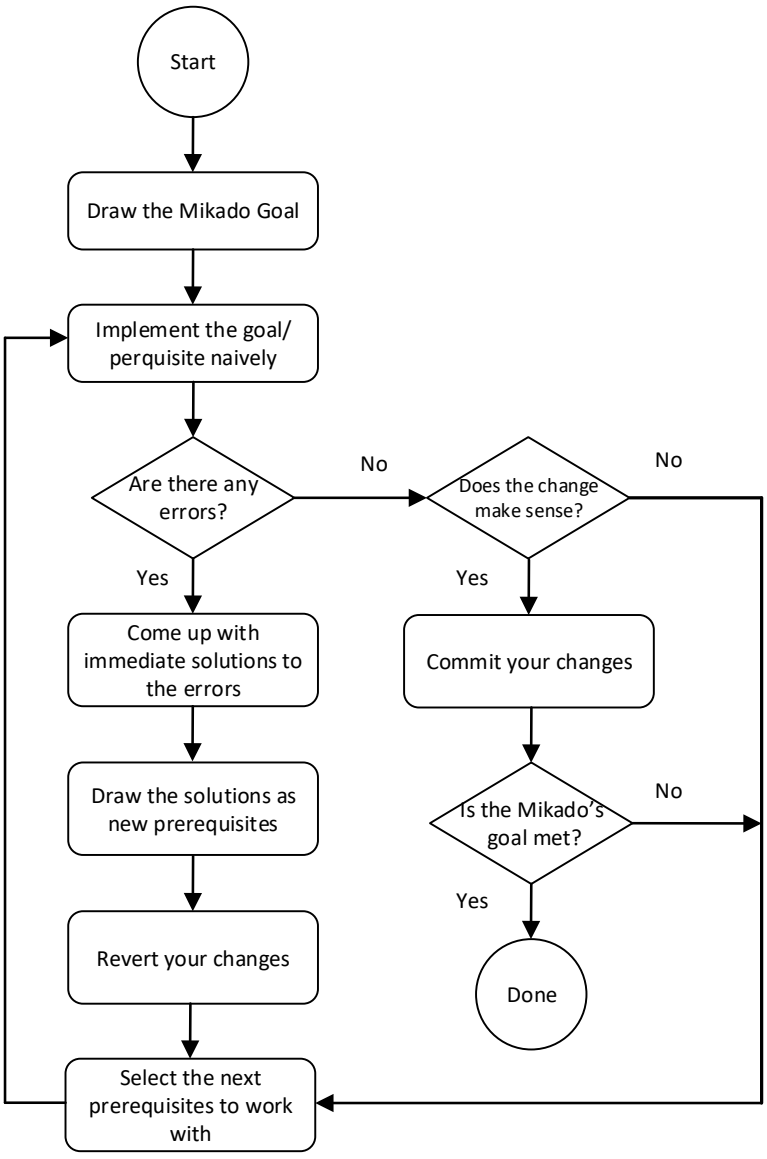


Fig. 1. The steps defined in the Mikado method.

There are four fundamental and known concepts that summarize the Mikado method [9]:

a) Set a goal: thinking and writing about the codes that should be changed. The concepts: 1) a starting point for the change; 2) an ending point or successful measures of the change are the basis of the experiment step.
b) Experiment: experiment is a procedure to validate a hypothesis. To achieve the goal of the experiments for the code change, the prerequisites are feedback. The goal and prerequisites are visualized.
c) Visualize: visualization is carried out when the goal and required prerequisites to achieve that goal are written. A graph is the only artifact of the Mikado method. The Mikado diagram shows the goal and all prerequisites needed to achieve that goal, telling what the next step is.
d) Undo: when an experiment separates the system to implement a goal or a prerequisite, and you have visualized the change that should be applied to the system to prevent this outcome, your changes should be undone to return to the previous state. In the Mikado method, you always visualize the prerequisites and then undo the separation of the changes.

The experiment, visualization, and undo processes for each prerequisite are iterated for the next layer of the prerequisites, and so on. In the following, matching and employing the Mikado phases in different steps of the proposed approach are described in detail.

3. RELATED WORK

The literature review demonstrates that various techniques and methods have been presented to identify code smells, where each one has a specific objective. Some of the most important relevant studies are examined in this section.

Gadient et al. [10] stated that the inter-component communication (ICC) is the common source of security vulnerabilities in the android programs. They proposed a lightweight static analysis tool on the Android Lint that analyzes the code being developed, providing just-in-time feedback in IDE about the presence of such smells in the code. In another study, Goseva-Popstojanova and Perhinschi [11] evaluated three common commercial static code analysis tools using the benchmarking test suite Juliet. They compared these tools in terms of their abilities to detect security vulnerabilities in C, C++, and Java. The results showed that despite recent advances in static code analysis methods employed in these tools, they could not detect security vulnerabilities efficiently.

Nunes et al. [12] studied choosing the right static analysis tools for finding vulnerabilities in web applications. They proposed a benchmark for comparing and evaluating static analysis tools in terms of their ability to detect security vulnerabilities. Using this benchmark, they showed that the best performance of static analysis tools depends on the deployment scenario and the vulnerability class being identified. However, this novel benchmark is a valuable tool to improve the abilities of static analysis tools, its implementation in practice is not easy.

Mumtaz et al. [13] investigated how removing bad code smells through refactoring can improve program security. Conducting several experiments using various security metrics, they showed that refactoring helps to improve the security of the software without affecting

the overall quality of the software systems. However, they did not propose a structured method in this regard.

According to Li et al. [14], ASIDE, ESVD, LAPSE+, SpotBugs, and FindSecBugs are five open-source IDE plug-ins that can identify and report vulnerabilities. The plug-ins were then assessed and compared in terms of the number of vulnerability categories they could detect, how they detected vulnerabilities, and how user-friendly their output was. The findings revealed that although some vulnerabilities, such as broken access control, are widely supported by most plug-ins, others are simply ignored.

According to Rachow [15], current approaches simply address code smells and design issues while neglecting the architectural impacts. The goal of this project was to create a decision-making framework that integrated architectural smell detection, appropriate refactoring selection, and impact analysis to prioritize refactoring that helped developers and software architects by measuring and comparing the required time and quality of the obtained software using control groups.

Fontana et al. [16] used correlation analysis to compare 19 code smells, six code smell categories, and four architectural smells in a case study. The goal of this investigation was to see if architectural smells are independent of code smells or if they may be derived from a group of code smells. The findings revealed that architectural smells are only associated with a small number of code smell events and that they cannot be derived from code smells.

Rahman and Williams [17] conducted an empirical study to help software experts to improve infrastructure quality as coded scripts (IaCs) that identify the source code attributes of defective infrastructures as IaC coding scripts. To discover the source code attributes associated with incomplete IaC scripts, qualitative analysis was employed in this work to confirm linked defects gathered from open-source software repositories. Their structural defect prediction models showed an approximate accuracy between 0.70 and 0.78 and a recall of 0.54 to 0.67. According to the findings, experts are advised to try hard to inspect and test IaC scripts, which include all ten features of the identified source code of IaC scripts.

In a systematic review, Kaur et al. [18] examined how code smells are prioritized in object-oriented software. Researchers noted that due to difficulties such as market pressure and time constraints, developers were often unable to eliminate all code smells and have to prioritize them. Also, Dos Reise et al. [19] investigated code smell detection methodologies and tools in another systematic literature review. They revealed that the most important smell detection techniques are search-based, metric-based, and sign-based approaches. Moreover, in another study, Kaur et al. [20] classified code smell detection strategies and tools based on simple and hybrid machine learning algorithms.

Koch et al. [21] developed three new smell detection approaches that use approximated spreadsheet structures to improve spreadsheet smells. Applying these approaches led to minimizing the amount of mistakenly reported smells and finding new code smells.

To promote secure programming techniques, Ghafari et al. [22] evaluated security-related studies and discovered avoidable vulnerabilities in Android devices and security code smells. They addressed the key vulnerabilities and smell in detail and explained how to eliminate or reduce them during development. In addition, they proposed a lightweight static analysis tool and assessed its performance to find various vulnerabilities on around 46,000 Apps hosted in the official Android market. In this study, 28 security code smells were found

and grouped into five classes while introducing the approaches for introducing secure programming.

The literature review shows that the main focus of the related cases and studies is on Java codes. This has been reported by several related work [23, 24]. For this reason, most related studies have focused on this language to compare their achievements with previous works. Moreover, only a few studies have investigated identifying security bloats. Thus, this research aims to provide an effective solution to fulfill this research and practical gap. In this context, a methodology and tool for a thorough assessment of security code bloating are described in the current article. The advantages and contributions of this work are explained in the next sections.

Most of the tools are designed based on the Abstract Syntax Tree parser model, and they also perform code parsing on Java codes, which must be compiled before code parsing. In comparison, the proposed model focuses on the syntax-metric parsing engine. Another advantage of the proposed model is to cover the search for possible insecurity cases that exist in the form of comments within the codes. In addition, the developed tool covers four programming languages. Using the proposed method in the current study, the codes in Java language do not need to be compiled for analysis. Moreover, as mentioned before, the existing models are often analyzed in Java language and compiled codes. In comparison, the proposed model proposed in our research has the ability to add criteria according to Common Weakness Enumeration (CWE) from the website https://cwe.mitre.org.

4. RESEARCH DESIGN

Through analysis of the insecure gaps of the code smell, a set of collected measurements in programming languages that mostly result in insecurity of the produced codes is offered in this section. For this purpose, the proposed way to match a measure for detecting code abnormalities is investigated in the following, and the security bloat classes are added as a new subclass to the smells, taking into account the necessity of describing the behavior of each metric. Since the security issue in information systems has been established, this term is commonly used in a variety of scenarios, which are explained by examining the behaviors displayed by these programs.

Some faults in this structure were discovered after reviewing numerous approaches and code smell types, as well as studying various refactoring methods, paving the path for more study and research in this area. We took steps to develop new concepts in this context and investigate the classifications offered by earlier researchers and the requirements of this context for the introduction of insecure bloat, which had been disregarded, due to a lack of study in the context of insecure codes in code smells. For the first time, the class of insecure bloaters is investigated in a novel way, aiming to expand the range of smells.

4.1. Overview of the Proposed Model: Touba

In this paper, a method for detecting insecure bloat vulnerabilities is proposed. This method is known as Touba Security Bloat Mistake Detector (TSBMD). Because of the model's special objective of detecting and addressing security bloat problems, the proposed solution has been given the name Touba, which refers to the Touba military march. This name was also

given to one of Japan's rulers in honor of his devotion to the Mikado. According to the type of analysis language, several metrics are evaluated for monitoring and detecting insecure bloat problems. These metrics are classified into two categories: similar and dissimilar metrics (homogeneous and heterogeneous metrics).

After determining the impact of employing similar and dissimilar metrics in identifying security concepts, this work proposes a system that allows users to use many metrics at the same time. In general, this approach scans the parameters used to construct the searched bloaters first, then summarizes the examples found using various metrics. The code's security is then calculated using these two sets of metrics, and the results are then combined. It is worth noting that the metrics in this study are calculated using a variety of input and output factors, which are introduced in the following parts. In addition, the suggested solution includes a novel engine for parsing code and aggregating the above-mentioned findings. Furthermore, the static analysis procedures have been adapted to the phases of the Mikado approach in order to make the proposed method compatible with it.

The suggested tool's high-level architecture is depicted in Fig. 2. The files are first received as input in this step. The engine separates the code, which is the primary section of static analysis, after satisfying the syntax-metric analysis criteria. This analysis can be adjusted according to the end user's rules, as a consequence of which probable bloaters in the code are identified, and the findings are visualized after analysis. After the process is completed, the tool outputs the analysis results in a statistical format.

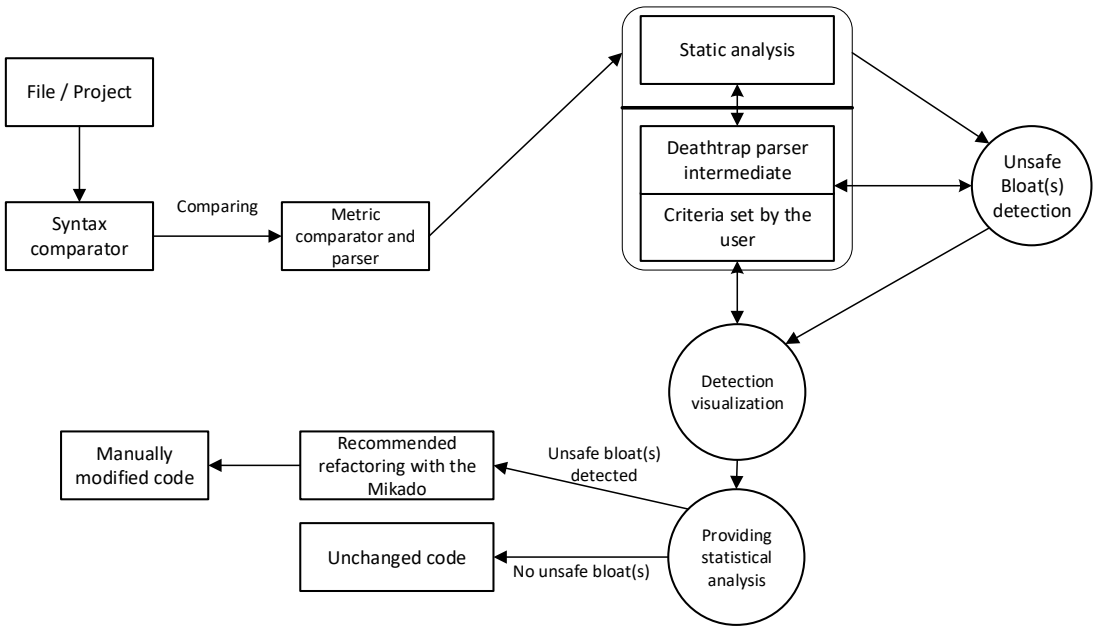


Fig. 2. High-level architecture of the proposed model.

To follow and match the proposed code analysis processes with Mikado phases, the following steps are taken:

a) The goal of the Mikado approach is to find and refactor the recognized insecure bloats, the descriptions and code lines about the detected bloaters from the goal step.

b) The Mikado method's experiment step involves parsing and analyzing all of the codes in a program. The tool provides numerous mappings to the incoming code that are actually the undo step during the code analysis.
c) The visualization process includes statistical analysis and graphics related to separation and constructing the final result.
d) In an agile method, undoing or repeating is an unavoidable element of the process; as a result, it is essential to repeat the identification operation after refactoring in order to locate the bloaters which have not been refactored.

The proposed method based on the Mikado method is depicted in Fig. 3.

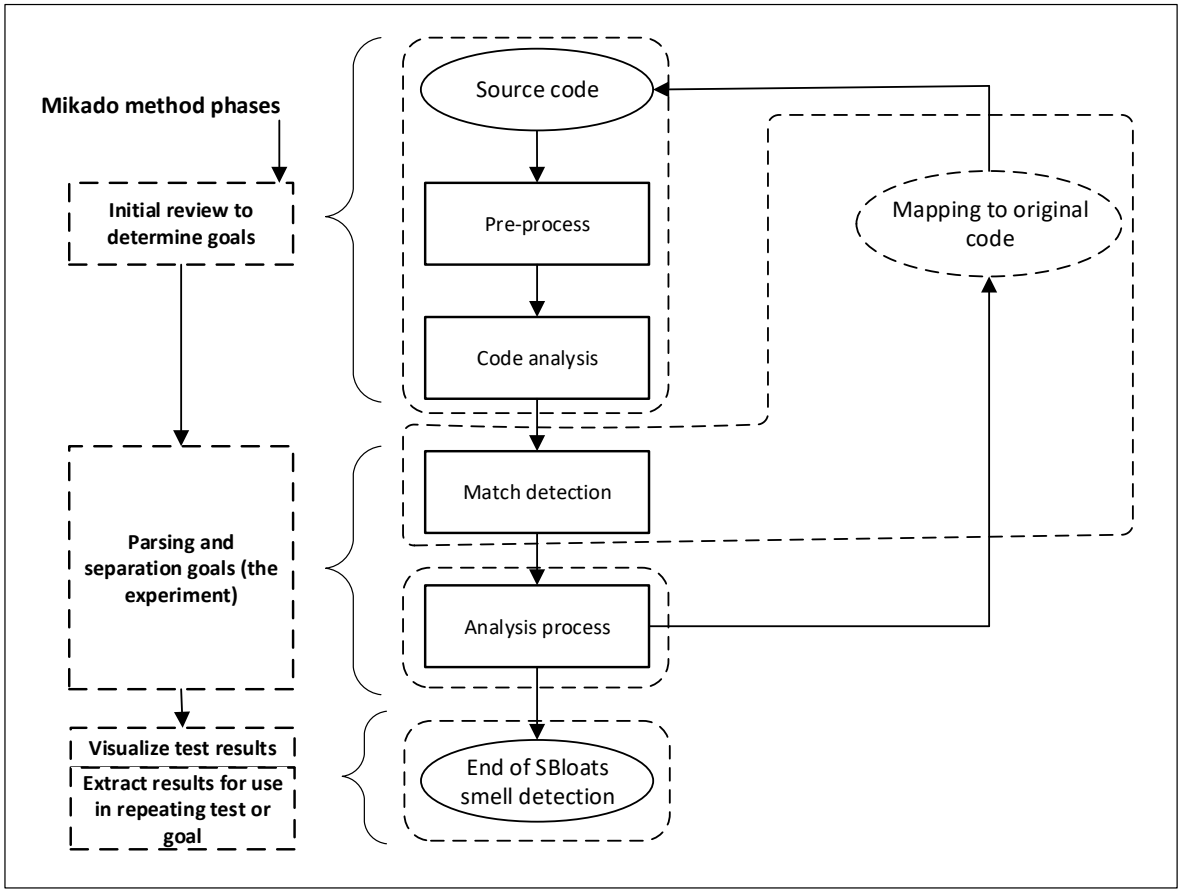


Fig. 3. Overview of the proposed method based on the Mikado method.

In all phases of the proposed model, including data pre-processing, primary analysis, and secondary analysis, there is access to the source code, and the analysis is done based on the accessible codes. Based on the codes received in the first phase, the required statistical data and charts are created in the last phase.

4.1.1. Inputs and Outputs of the Model

The proposed method requires particular inputs and generates proper outputs based on achieving the desired goals. The following are some of the inputs to the suggested method:

- Various files containing a sample programming language code that is being assessed. These files are used as Test Suites for analysis.

- The insecure signs and security bloaters' database. To compare and match the received codes with the identification metrics, the tool needs to read information of these metrics as input and match them with the received codes.
- Initial settings for scanning by the user.

The outputs of the proposed model include 14 parameters, which are listed in Table 3.

Table 3. Output parameters of the proposed model.

No.	Output parameter
1	The total number of project code lines
2	Total of comment lines
3	Number of lines without code
4	Total of lines
5	Number of potentially unsafe codes
6	Number of security signs
7	Percentage of file code contribution in the project
8	The total number of project code lines
9	Total of comment lines
10	Number of lines without code
11	Total of lines
12	Number of potentially unsafe codes
13	Number of security signs
14	Percentage of file code contribution in the project

4.1.2. Abbreviations

Table 4 lists the abbreviations used in the proposed model and its application.

Table 4. Abbreviations used in the proposed model.

Abbreviation	Phrase
PR	Project
UBC	Unsafe Bloat Code
SF	Single File
WS	WhiteSpace
CL	Comment Line
DTC	DeathTrap Code
ToB	Total of Bug
DTS	DeathTrap Signs
PUC	Percentage of Unsafe Code
PSC	Percentage of Safe Code
PCD	Percentage of Contribution Deathtrap
TUS	Total UnSecure
PFC	Percentage of File Contribution
LoC	Line of Code
ToL	Total of Line
Cmt	Comment

4.2. Details of the Proposed Model

The suggested approach, as shown in Fig. 4, is divided into three sections: data pre-processing, static code bloating analysis, and metric index classification processor, all of which are explained in this section.

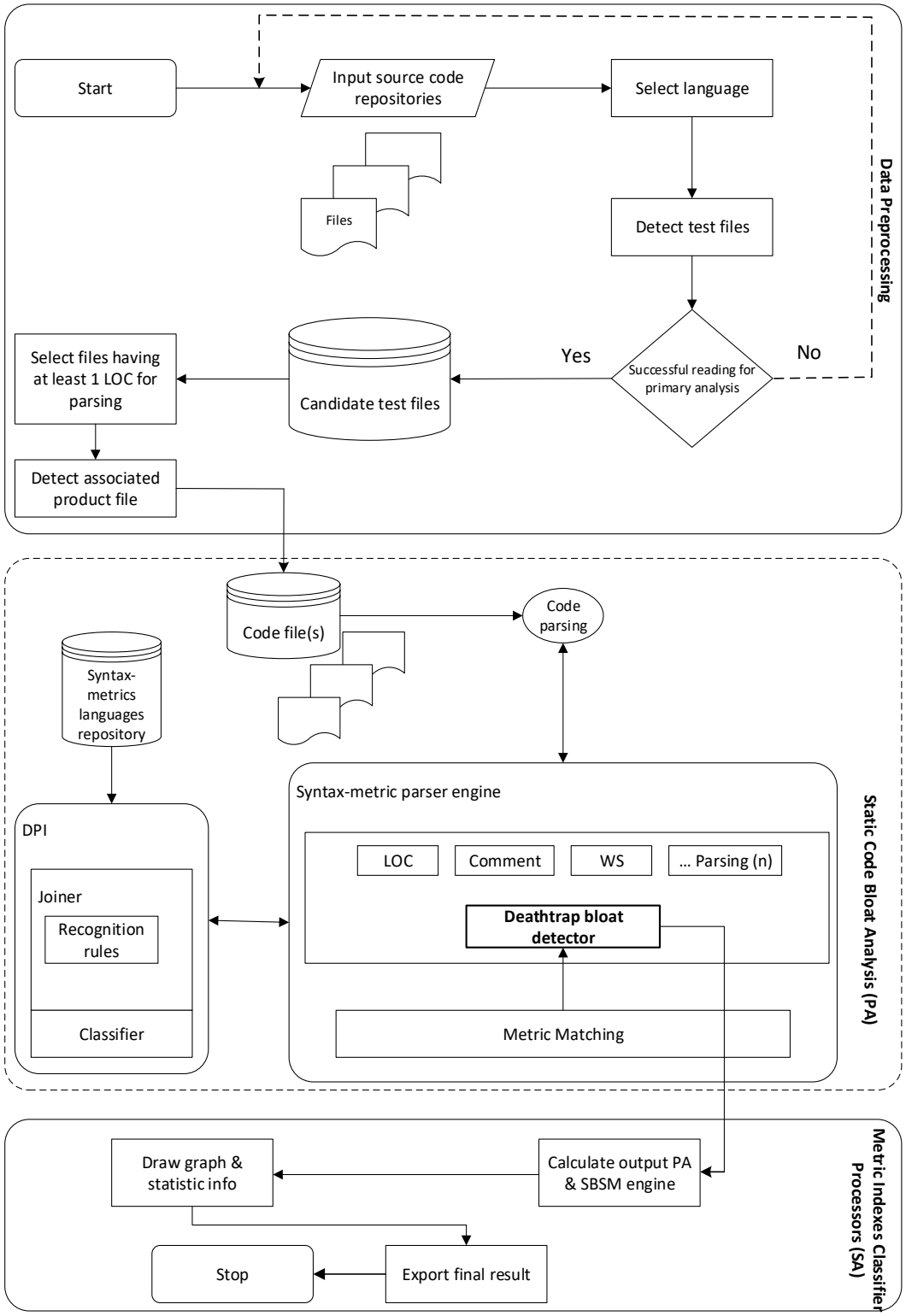


Fig. 4. Steps and phases of the proposed model.

4.2.1. Section 1: Data Pre-Processing

Static analysis tools and models work on the source code directly. This feature is advantageous for these tools because tools that operate on compiled code do not appropriately discover existing errors. The model suggested in this study works with source code and requires that the source codes be pre-processed before being used as analytic inputs.

The data pre-processing phase is the first stage in the future processes of data analysis. The circumstances and fundamental principles for processing the received data are reviewed for this purpose. If the selected pre-processing language does not match the files, the user must either adjust the intended language or introduce the files relevant to the selected language to the program using these guidelines. In subsequent phases, files that do not include at least one line of code will be excluded from the analysis process. All activities in this step are conducted automatically without human interaction after selecting the language and input files. The order of actions in the data pre-processing step is shown in Table 5.

Table 5. The sequence of work process in the data preprocessing phase.

No	Operation
1	Select the type of programming language
2	Read source code from a file or project as input
3	Check the compatibility of the selected project with the type of language being
4	Mismatch of input files with the type of target language and return in one or two
5	Separate user interface files from source code files
6	Array navigation of all files throughout the project to perform duplicate operations
7	Checking the validity of candidate files in terms of file type for code analysis and
8	Failure to qualify in number seven and return to stage two
9	List indexed test files for initial analysis
10	Check for at least one line of code in the measured file to continue the analysis
11	Final check of file extension related to programming language
12	Prioritize files for parsing by filename
13	Extract extensions of participating files in the project
14	Display a list of the number of filtered files allowed for analysis and processing in
15	Save pre-processing indexes in the temporary database

4.2.2. Section 2: Static Code Analysis and Detecting Insecure Bloaters

The initial data analysis is the second phase of the suggested model. The data processing findings from the previous phase are used as input in this step; that is, the codes are ready for analysis, and all of the essential conditions for analysis have been set. The potential faults from now on are unrelated to the preceding phase. The syntax-metric parser engine receives the revised codes. The separation is done simultaneously and in parallel on the syntax metrics and the insecure bloater detection metrics. The specified engine's insecure analysis portion uses a metric adaptor in conjunction with a database of similar and dissimilar metrics, which is detailed further below. To allow two-way communication between the adapter and the database, a deathtrap parser intermediate (DPI) is used. Bloater detection rules are used by the DPI to categorize the cases that are found. The sequencing of operations in this step is shown in Table 6.

Table 6. The sequence of work process in static code analysis phase.

No.	Operation
1	Create a tuple of files received from the pre-processing phase and metrics stored in the database
2	Check the occurrence of exceptions in analysis calculations
3	Exception error handling and processing
4	Issue a message according to the managed errors
5	Array navigation of a set of code lines throughout the file to perform operations on each code
6	Syntactic parsing process on the analyzed code line
7	Scrolling through the set of metric in the benchmark repository for each line of code
8	Checking the compliance of the code line with the criterion for measuring vulnerability
9	Compare each metric according to the target programming language
10	Analyze detected security vulnerabilities according to SQL injection criteria, deadlock, script injection through website, integer overflow, prime vector, etc.
11	Classify the criteria found in bloated code for secondary analysis

- **Insecure Metrics in the Detection Process**

Insecure bloats caused by haphazard management and the use of untrained programmers result in security gaps in the developed software, which can be exploited by hackers and unauthorized access. From the overall list of 927 Common Weakness Enumeration (CWE) of current software, the items discovered and selected as security bloaters are the most similar to the cases and behaviors of software bloats, according to several metrics introduced on the Mitre site [25]. Bloats are split into two kinds in the suggested method for this purpose: similar metrics (common) and dissimilar metrics (specific).

Intentionally or inadvertently, the existence of insecure bloats renders the code and executable file vulnerable. The process of damaging programs can be stopped, or their structure can be addressed by current vulnerability detection by recognizing numerous insecure bloated metrics in these two groups.

- **Insecure Similar Metrics**

The metrics that have been found as security bloater defects in more than one or all of the popular programming languages are bloater-related metrics in the process of detecting insecure code. Insecure bloats like these are used to characterize situations and behaviors that are typical in these languages. These are defects in software that cause security issues by lowering its quality and performance. The following are some of the most relevant evaluation measures in this class.

Table 7 illustrates the similar measures that were used in the proposed model's test case analysis.

- **Insecure Dissimilar Metrics**

Bloater dissimilar metrics are metrics that are not common among conventional coding languages in terms of behavior, states, structure, and type of vulnerabilities, and they are limited to a single language in the insecure code detection procedure. Table 8 demonstrates the dissimilar metrics employed in the suggested approach's test case analysis.

Table 7. Public metrics in the proposed model.

No.	Metric
1	Check for cross-site scripting (XSS)
2	SQL injection problems
3	Check for un-validated variables being executed via cmd line/system calls
4	Insecure network protocol (check for safe redirects and safe use of URLs)
5	Int overflow
6	Race conditions
7	TOCTOU vulnerabilities
8	Weak crypto algorithm
9	Weak crypto configuration
10	Exposed credentials
11	Brute-force and dictionary attacks
12	Hardcoded keys

Table 8. Private metrics in the proposed model.

No.	Metric
1	Check for turned off of .NET default validation
2	Enable or disable the config file to determine the .NET
3	Identify any initialization vector keys
4	Identify potential for deadlocking
5	random functions that are not cryptographically secure
6	Correct implementation of inherited SAML2 functions
7	Check for unsafe cloning implementation
8	Check for turned off of .NET default validation
9	Check for any issues related to servlets, such as bloat
10	Unsafe use of java.lang.Runtime.exec
11	Security check used in try ... catch blocks
12	Identify occurrences of realloc
13	Identify entry to the class destructor, report any exception
14	Check for printf format string vulnerabilities

Much of the identification of key metrics and variables use the Regular Expressions method; these terms are used to search for and match one or more specific search patterns. Following a specific pattern defined for each metric, vulnerabilities are identified and categorized. That is, if there is a specific type of security vulnerability, it is transferred to the defined category, and numerical calculations are performed based on it. In fact, the detection thresholds presented in each of the similar and dissimilar criteria operate based on the mechanism described. Criteria detection is based on a comparison of keywords and regular phrases.

4.2.3. Section 3: Metric Index Classification Processor

The findings of the second stage, comprising the parameters retrieved by the syntax-metric parser engine, are collected, and the derived statistics are processed in the secondary analysis step, also known as calculating and processing the analysis results. As a result of the tool's output, statistical data and analytical graphs are generated and presented. The operations connected to the acquired results have been designed as relationships and

equations to improve the readability of the computations and make the symbols used in the suggested technique easier to understand. To this end, the suggested model's output parameters are computed using the relationships and equations below.

Eq. (1) is used to calculate the number of lines of a project. In this equation, *PR* is the whole project, *LOC* is the pure number of lines of a code, *ToL* is the total number of lines of a project without filtering, including white space (*WS*), and the comment lines (*CL*). *SF* represents the singular files, and the summation operation is carried out to the number of files of the project. The value of the involved parameters is obtained through the summation of every single value in the singular files of the source code, where SF_n represents the number of files that should be incorporated in this equation. In $\sum_{i=1}^{SF_n} ToL_{PR} - (WS_{PR} + CL_{PR})$, for the first to the nth singular file (*SF*), the total lines of the white space and the comment lines are obtained, and then the total number of lines is subtracted from WS_{PR} and CL_{PR}; the result is pure lines of code.

$$LOC_{PR}(SF) = \sum_{i=1}^{SF_n} ToL_{PR} - (WS_{PR} + CL_{PR}) \quad (1)$$

To obtain the total number of lines of a singular file with the symbol ToL_{SF}, three key parameters of the total number of lines of code (*LoC*), comments, and white space should be counted. The ToL_{SF} value of the total of the parameters obtained in each line is obtained for $i = 0, 1, \ldots n$ for lines of code $\|LoC_i\|$, $j=0,\ldots,m$, for the comment lines $\|Comment_j\|$, $K=0,..,q$, and for the white space lines, $\|WhiteSpace_k\|$. Since comment and white space might not exist in a singular file, these two parameters are counted from zero; Therefore, the number of lines in a file is calculated using Eq. (2).

$$TOL_{SF} = \|LoC_i\| + \|Comment_j\| + \|WhiteSpace_k\| \quad (2)$$

First, the number of lines of code, comment, and white space in each singular should be calculated to estimate the total number of lines in a project. In this process, the lines of code $LoC = (LoC_1, LoC_2, \ldots, LoC_n)$, comment lines $Cmt = (Cmt_1, Cmt_2, \ldots, Cmt_n)$, and white space lines $WS = WS_1, WS_2, \ldots, WS_n$) are a set of *n* digits obtained in each file, where LoC_j, Cm_{tj} and WS_j are equal to j^h the largest component in $SF=(SF_1, SF_2, \ldots, SF_n)$ and SF_n are equal to the number of files that should be measured. Thus, the final value and the sum of the mentioned values depend on the number of the singular files. It is evident that the main factor that determines this operation is calculated by the n value of the file for the whole project is calculated using Eq. (3).

$$TOL(SF_1, SF_2, \ldots, SF_n) = \sum_{j=1}^{n}(LoC_j + Comment_j + WhiteSpace_j) = (LoC_1+Cmt_1+WS_1) + (LoC_2+Cmt_2+WS_2) + \ldots + (LoC_n+Cmt_n+WS_n) \quad (3)$$

Eq. (4) is used to obtain the number of detected codes with a security vulnerability that their presence results in security bloat in the developed program. In this regard, DTC_{SF} represents the number of insecure bloaters for a file, which is denoted by the finder function *Cnt* and the value of the insecure bloating code (*UBC*). Then, the operation, $Cnt(UBC)_1$, $Cnt(UBC)_2, \ldots, Cnt(UBC)_n$ is calculated *m* times in $\sum_{i=1}^{m} Cnt(UBC)$.

$$DTC_{SF} = \sum_{i=1}^{m} Cnt(UBC) \quad (4)$$

The total amount of insecure codes discovered in a file or project determines the project's vulnerability. Eq. (5) is used to calculate the insecurity degree of a singular file represented by PUC_{SF}, and Eq. (6) is used to measure the insecurity degree of a code for the entire project, represented by PUC_{PR}, which is expressed in percent and requires two parameters of potentially insecure codes *DTC* and lines of code *LoC*, whose calculation is

described in the previous equations. How to obtain them was discussed in earlier relationships .

In Eq. (5), the LoC parameter is first calculated for all lines of code, and then the final value of the two primary parameters DTC_{SF} is divided by the total lines of code in the singular file LoC_{SF}. Finally, the values are expressed in percent.

$$PUC_{SF}= \left(\frac{DTC_{SF}}{\sum_{cnt=1}^{n} LoC_{cnt}}\right) * 100 \quad (5)$$

For all singular files SF_n, the total value of the parameters *LoC* and *DTC* are determined first in Eq. (6). The *DTC* value for all source code files is then divided by *LoC*, and the result is expressed in percent.

$$PUC_{pr} = \left(\frac{\sum_{cnt=1}^{SF_n} DTC}{\sum_{cnt=1}^{SF_n} LoC}\right) * 100 \quad (6)$$

The percentage of secure code, PSC_{PR}, is derived using Eq. (7) for the entire project.

$$PSC_{PR}= (100-PUC_{PR}) \quad (7)$$

In order to calculate the total number of errors detected (ToB_{SF}) in a file in terms of code (*DTC*) and Bloater insecure signs (*DTS*), the sum of these two parameters, $\sum_{cnt=1}^{n} DTC \cup DTS$, is calculated and divided by the count *LoC* (*CLoC*) in a singular file using Eq. (8).

$$ToB_{SF} = \frac{1}{CLoC} \sum_{cnt=1}^{n} (DTC \cup DTS) \quad (8)$$

DTS are lines of code that contain commands and codes of the analyzed programming language, but they are included in the code as comments. Signs are calculated based on the number of lines of comment that are considered security vulnerabilities.

Eq. (9), where $DTC_{a,n}$ is the n^{th} insecure bloater found by the a^{th} file divided by the total number of Bloaters detected in the whole project yields the percentage of Contribution Deathtrap (*PCD*) of a file in terms of the identified bloaters to the complete project.

$$PCD(SF_{a,i}) = \frac{\sum_{n=1}^{i} (DTC_{a,n})}{TotalCommitUnsafe} \quad (9)$$

The total insecurity of a project is determined as the total insecurity of the project to the total mistakes of the singular files, including discovered insecure bloaters and signs (*DTC*, *DTS*) for $i=1,2,\ldots,n$ the existing *TotalLoC* of the project using Eq. (10).

$$TUS(PR) = \frac{\sum_{i=1}^{n} (DTC,DTS)}{TotalLoC} * 100 \quad (10)$$

Eq. (11) calculates the contribution of each singular file of the project $PFC_{(PR)}$ to the total LoC of the project. ToL_{SFi} is the number of lines (code, comments, and white space) in the i^{th} singular file, divided by ToL_{PR}, or total calculated lines in a project, and expressed in percent.

$$PFC(PR) = \frac{TOL_{SF_i}}{ToL_{PR}} * 100 \quad (11)$$

The proposed method covers a wide range of security vulnerabilities. Among these, some are placed in the category of code smells and in the sub-group of bloaters. So, the engine is designed so that it is not exclusive to a specific type of vulnerability. Bloats have been mentioned as part of the vulnerabilities, and a number of criteria that can be used to detect them have been included in the proposed method.

The reason for naming the syntax-metric identification engine based on bloaters is that it has received more research attention. Highlighting bloats has been done due to the importance and more attention of this category. Like other security vulnerabilities, bloaters

are detected based on well-defined regular expressions and specific criteria patterns. For example, static codes are a typical example of bloaters.

4.3. Advantages of the Proposed Model

The suggested method differs from earlier methods in several ways. This strategy considers expanding the security bloats' application range and synchronizing the metrics scanning. The source code is examined rather than the compiled code. This paradigm also has the advantage of processing input from four programming languages: Visual Basic, C-Sharp, C++, and Java, rather than being limited to a single language like Java. It should be noted that most of the existing tools operate on Java codes only.

5. EVALUATION AND RESULTS

The proposed solution is implemented using the framework shown in Fig. 4. The program core and independent software components are created using Visual Studio .NET version 2015, C#, and Visual Basic programming languages, and data is stored using the open-source SQLite database management system. The proposed solution was implemented using the C#.NET environment as the principal programming language and is known as TSBMD in this article, as it was mentioned before. This tool contains a syntax-metric rule detection engine that works with source code programming languages like VB.NET, C#.NET, ASP.NET, C, C++, and Java. The aforementioned software is user-friendly, and numerous security metrics in the above languages can be added to the metrics pool and participate in the analysis process without the need for programming experience. A code editor is also included with the software to help with the process. The proposed tool categorizes the identified vulnerabilities based on the criteria considered in the metrics repository. Figs. 5 and 6 show the output of the software tool.

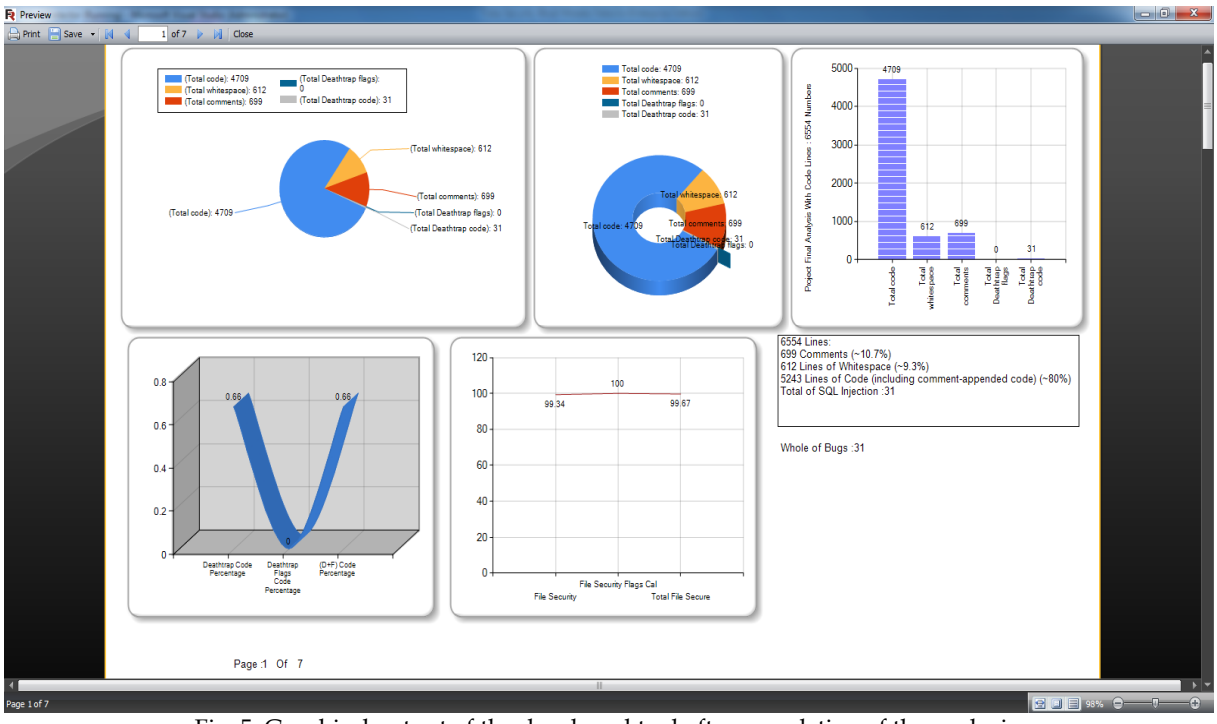
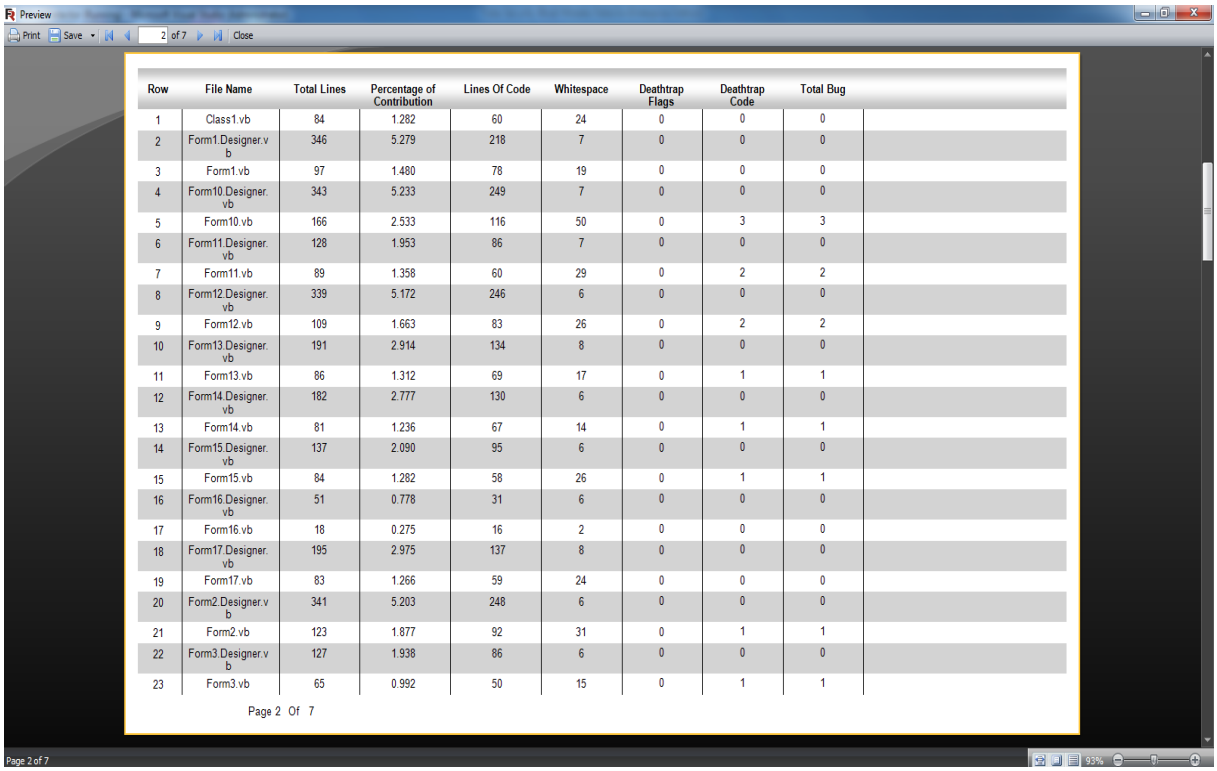


Fig. 5. Graphical output of the developed tool after completion of the analysis.



Row	File Name	Total Lines	Percentage of Contribution	Lines Of Code	Whitespace	Deathtrap Flags	Deathtrap Code	Total Bug
1	Class1.vb	84	1.282	60	24	0	0	0
2	Form1.Designer.vb	346	5.279	218	7	0	0	0
3	Form1.vb	97	1.480	78	19	0	0	0
4	Form10.Designer.vb	343	5.233	249	7	0	0	0
5	Form10.vb	166	2.533	116	50	0	3	3
6	Form11.Designer.vb	128	1.953	86	7	0	0	0
7	Form11.vb	89	1.358	60	29	0	2	2
8	Form12.Designer.vb	339	5.172	246	6	0	0	0
9	Form12.vb	109	1.663	83	26	0	2	2
10	Form13.Designer.vb	191	2.914	134	8	0	0	0
11	Form13.vb	86	1.312	69	17	0	1	1
12	Form14.Designer.vb	182	2.777	130	6	0	0	0
13	Form14.vb	81	1.236	67	14	0	1	1
14	Form15.Designer.vb	137	2.090	95	6	0	0	0
15	Form15.vb	84	1.282	58	26	0	1	1
16	Form16.Designer.vb	51	0.778	31	6	0	0	0
17	Form16.vb	18	0.275	16	2	0	0	0
18	Form17.Designer.vb	195	2.975	137	8	0	0	0
19	Form17.vb	83	1.266	59	24	0	0	0
20	Form2.Designer.vb	341	5.203	248	6	0	0	0
21	Form2.vb	123	1.877	92	31	0	1	1
22	Form3.Designer.vb	127	1.938	86	6	0	0	0
23	Form3.vb	65	0.992	50	15	0	1	1

Page 2 Of 7

Fig. 6. Statistical output of the developed tool after completion of the analysis.

In the artificial scenario, Juliet Test Suites [26] are offered in two components of the Java programming language and C++ to software developers for security evaluation. Thousands of test cases are included in each component, including similar functionalities with and without defects. These situations are referred to as bad and good codes, respectively. A common weakness enumeration (CWE) identifies defects in bad code, making defect detection simple. Each Juliet test case is designed to represent a CWE ID and includes good and bad codes. The defect reported by the CWE ID is present in the bad code. The matching good code is identical to the bad code, with the exception that it lacks the associated defect. Each test case focuses on a single sample defect; however, there may be more defects. Other defects in the test are disregarded to simplify the automated analysis outlined in the following section. Only the Java language and a few metrics in the field of code insecurity are used to evaluate the tool for the results to be presented in legitimate scientific forums.

The detailed data for the Juliet 1.2 test suite [27] is shown in Table 9. The number of various CWE identities covered by the test suite is listed in the CWE column. The LoC column shows the number of non-white lines in the code of each language that are without a remark.

Table 9. Juliet 1.2 specifications.

	CWEs	Test Cases	Files	LoC
C/C++	118	61387	102092	4719409 (C), 3882727 (C++)
Java	112	25477	41170	4565713

Because the Juliet Test Suites contain carefully identified defects, tool warnings can be automatically assessed. With the overall overview shown in Fig. 7, this section outlines the

analytical procedure. If their defect types are related, that is, their CWE IDs belong to the same group, and there is at least one warning spot in the allocated block, the tool alert is matched with a test case. The following are the details of the agreement calculation [28].

- If a related security vulnerability or unsafe bloat was in bad code, the tool had a true positive (TP).
- If no related security vulnerabilities or unsafe bloat was in bad code, the tool had a false negative (FN).
- If a related security vulnerability or unsafe bloat was in good code, the tool had a false positive (FP).
- If no related security vulnerabilities or unsafe bloat was in good code, the tool had a true negative (TN).

  Any unrelated security vulnerabilities were disregarded.

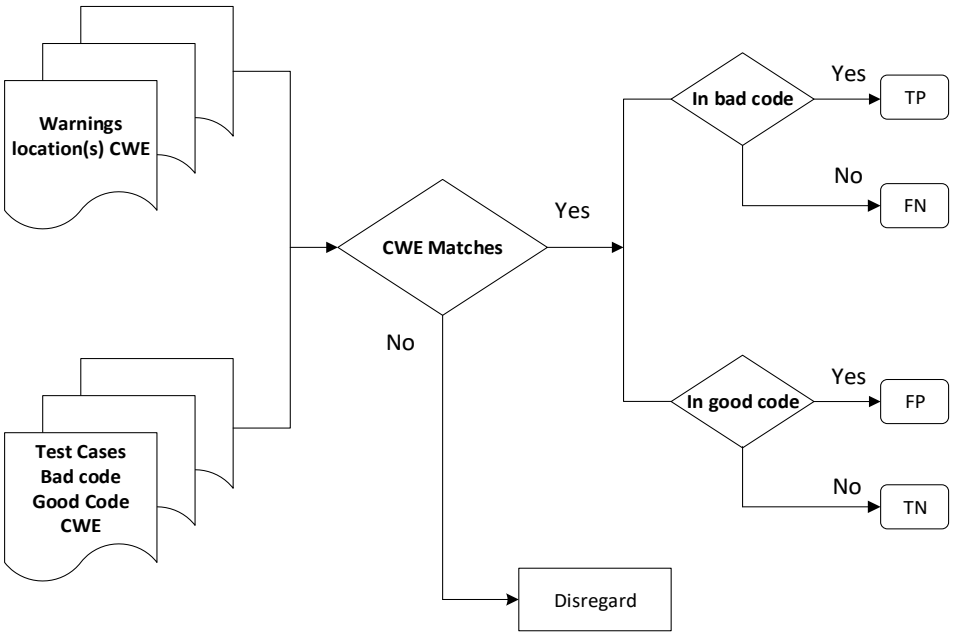


Fig. 7. The evaluation process for synthetic test suites.

The following are the formulas for calculating precision, recall, and F-Measure, as shown in Eqs. (12) to 14.

$$Precision = \frac{TP}{(TP+FP)} * 100 \quad (12)$$

$$Recall = \frac{TP}{(TP+FN)} * 100 \quad (13)$$

$$F\ Measure = 2 * \frac{Precision*Recall}{Precsision+Recall} \quad (14)$$

The first scenario test examines the scenarios that were chosen from the Juliet Test Suites. In this part, five tests from a total of 112 test cases written in Java are chosen to assess the effectiveness of the proposed tool. CWE83 SQL injection, CWE78 OS command injection, CWE83 XSS attribute injection, CWE190 Integer overflow, and CWE80 XSS injection are among the tests. The given tool, which is based on the method of this study, is compared to the most well-known tools in order to assess the efficiency of the proposed method. Some tools are demonstrated below.

The University of Maryland created FindBugs. It comes as a standalone application as well as an Eclipse plug-in [29]. In addition to static parsing, CodePro Analytix is only accessible as an Eclipse plug-in and provides several security check functions [30]. In the Linux platform, the FindSecurityBugs plug-in is utilized as a standalone application. Additionally, this tool includes a valuable feature that allows extra rules (add-on, double) to be supplied to extend the tool's experiments [31]. SonarQube is a program that, in addition to Java, supports a variety of other languages and may be used in conjunction with IDEs and other external tools [32]. The SQL injection test is used to identify the relevant defects [33, 34].

Because different tools have different limitations in CWE identification, the Juliet Test Suites are chosen proportionally to the same test for comparison, and the comparison in one test could be between many tools in one test and between two tools in another. Because other tools do not support the CWE test, this is the case. According to the CWE ID and the measuring tools in each parameter, Table 10 categorizes positive and negative samples. Table 11 compares the proposed model's precision, recall, and F-measure to those of other tools

In general, the results of using the proposed model and implementing the Touba Security Bloating Mistake Detector Tool show that the proposed model is effective in detecting security defects and has passed the Juliet test cases successfully. It should be noted that the lower performance of TSBMD compared to the other tools is because CWE80 uses a particular regular expression. Nonetheless, TSBMD was able to detect security vulnerabilities to a reasonable extent in CWE80. The findings also suggest that the proposed model and tool can be utilized as a solution and a code evaluation tool, reflecting the program's difficulties in being used by developers (development team) and stakeholders of software systems that have been developed or are being produced. Fig. 8 shows the precision, recall, and F-Measure of the proposed model compared to the existing tools in the test case CSEs.

Table 10. Statistical results of the data.

CSW ID	Tools	Total bad test cases	Total good test cases	TP	FP	FN	TN
CWE 89	TSBMD	2220	8165	1409	1200	811	6965
	CodePro	2220	8165	795	4347	1425	3818
	FindBugs	2220	8165	1200	3097	1020	5068
	SonarQube	2220	8165	888	1200	1332	6965
CSE 78	TSBMD	444	1047	437	527	7	520
	FindSecurityBugs	444	1047	400	679	44	368
CWE 80	TSBMD	666	1566	327	95	339	1471
	CodePro	666	1566	409	38	257	1528
	FindBugs	666	1566	491	213	175	1353
CWE 83	TSBMD	333	783	108	37	225	746
	CodePro	333	783	27	56	306	727
	FindBugs	333	783	18	43	315	740
CWE 190	TSBMD	2553	9456	1363	2302	1190	7154
	FindBugs	2553	9456	135	1090	2418	8366

Table 11. Comparison of precision, recall, and F-measure of the proposed model with other tools.

CWE ID	Tools	Precision	Recall	F-measure	Improved TSBMD compared to			
					Tools	Precision	Recall	F-measure
CWE89	TSBMD	54.01	63.47	58.36				
	CodePro	15.46	35.81	21.60	CodePro	38.54	27.66	36.76
	FindBugs	27.93	54.05	36.83	FindBugs	26.08	9.41	21.53
	SonarQube	42.53	40.00	41.23	SonarQube	11.48	23.47	17.13
					AVG.	25.37	20.18	25.14
CWE78	TSBMD	45.33	98.42	62.07				
	FindSecurityBugs	37.07	90.09	52.53	FindSecurityBugs	8.26	8.33	9.55
					AVG.	8.26	8.33	9.55
CWE80	TSBMD	77.49	49.10	60.11				
	CodePro	91.50	61.41	73.50	CodePro	-14.01	-12.31	-13.38
	FindBugs	69.74	73.72	71.68	FindBugs	7.74	-24.62	-11.57
					AVG.	-3.13	-18.47	-12.48
CWE83	TSBMD	74.48	32.43	45.19				
	CodePro	32.53	8.11	12.98	CodePro	41.95	24.32	32.21
	FindBugs	29.51	5.41	9.14	FindBugs	44.97	27.03	36.05
					AVG.	43.46	25.68	34.13
CWE190	TSBMD	37.19	53.39	43.84				
	FindBugs	11.02	5.29	7.15	FindBugs	26.17	48.10	36.69

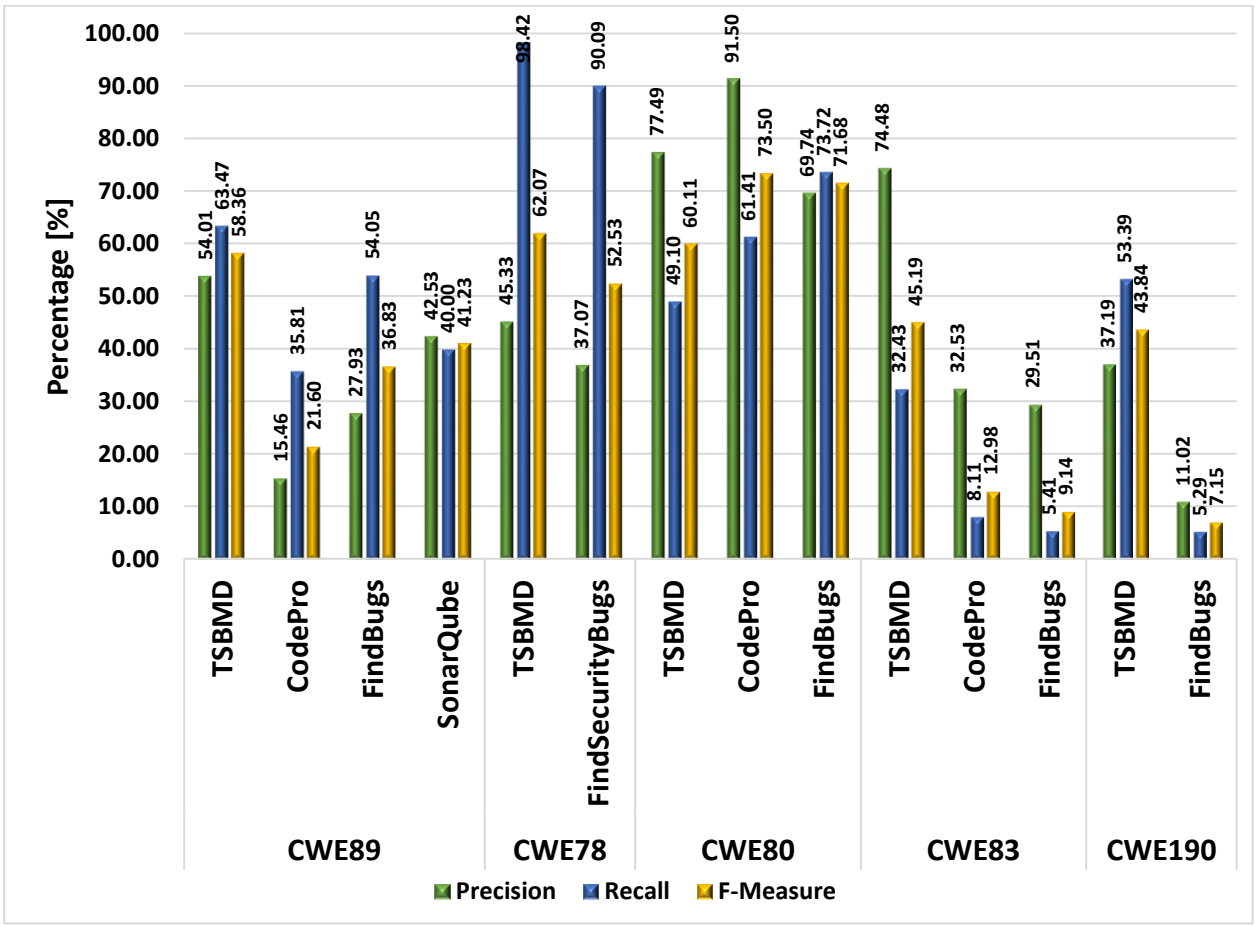


Fig. 8. Comparison diagram of precision, recall, and f-measure of the proposed model with other tools in the test case CWEs.

6. LIMITATIONS

Because of its nature, this study has some limitations; where the most significant ones are as follows:

- Pattern standardization (Benchmark): since bloater search and detection in each language is based on keywords, graphemes, and various functions, pattern standardization is necessary. As a result, the number of metrics that can be used to compare and search bloaters in the target languages is restricted.
- Tool comparison: there are only a few security tools that can be compared to and evaluated against the proposed tools and methodologies.
- Incomplete metric coverage: security vulnerability detection tools and software, in general, do not cover all of the metrics reported in this study. As a result, various tools are employed to assess certain indicators.
- Measurable programming language: as mentioned before, because security vulnerability analysis tools focus on the mentioned language, it is not possible to compare logical advancements and evaluate the existing tools due to a lack of thorough study.
- Test cases (formal test): artificial test cases for insecurity analysis tools have been confirmed. There are valid tests in the context of C++ and Java, such as the Juliet test, and sufficient documentation to guide the user in using them; however, there is no documentation for using these test cases and determining the number of positive and negative samples by their developer, despite having test cases for other languages such as C#, Python, PHP, and so on.

7. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

Code smell, as a serious sign of security vulnerabilities, has been considered by software researchers. The current study proposed a model that uses a syntax-metric parser engine to detect security vulnerabilities. This model is rooted in the Mikado methodology. Moreover, based on the proposed method, a software tool has been developed to show its real performance in detecting security vulnerabilities. Evaluation of this model by Juliet test cases revealed that the proposed tool outperformed other tools in four out of five CWE scenarios. In general, the proposed method showed a better performance compared to the existing tools in terms of accuracy by 20.3%, recall by 16.76%, and F-measure by 18.61% on average.

The necessity of applying security detection metrics for bloat detection becomes evident when considering the conducted studies and the analysis and assessment outcomes of the tools offered in this paper. However, most of the existing tools focused on employing metrics and approaches regarding the different forms of code smell presented in earlier studies. So, making some changes to existing tools and integrating procedures may improve the results clearly. As a result of the debate above, the following can be examined in future studies.

- Investigating common metrics of insecure bloaters in programming languages in related research fields in order to assess the quality of output code generated by various parameters and apply changes to make them more compatible with this field.
- Due to the extent of integrated software development environments and lack of comprehensive studies, identified insecure scenarios are fairly limited; however, more

programming languages can be added to current tools to find security factors if the target languages are carefully examined.

- One method for simplifying the operation environment and its interaction with the processed IDE is to create a plug-in in the same development environment, which simplifies the operation when identifying smells and allows programmers to identify insecure smells in their code without having to switch between two environments.
- In the context of insecure code, artificial intelligence and machine learning can be utilized to improve identification and refactoring, allowing the system to learn how to provide the best methods for reconstructing the insecure code.

REFERENCES

[1] S. Habchi, N. Moha, R. Rouvoy, "Android code smells: From introduction to refactoring," *Journal of Systems and Software,* vol. 177, pp. 110964, 2021.

[2] M. Gholami, F. Daneshgar, G. Beydoun, F. Rabhi, "Challenges in migrating legacy software systems to the cloud — an empirical study," *Information Systems,* vol. 67, pp. 100-113, 2017.

[3] P. Louridas, "Static code analysis," *IEEE Software,* vol. 23, no. 4, pp. 58-61, 2006.

[4] J. Al Dallal, A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: a systematic literature review," *IEEE Transactions on Software Engineering,* vol. 44, no. 1, pp. 44-69, 2018.

[5] M. Laguna, Y. Crespo, "A systematic mapping study on software product line evolution: from legacy system reengineering to product line refactoring," *Science of Computer Programming,* vol. 78, no. 8, pp. 1010-1034, 2013.

[6] K. Beck, M. Fowler, G. Beck, "Bad smells in code," in *Refactoring: Improving the Design of Existing code*Westford, Massachusetts: Addison Wesley Longman Inc., pp. 75-88, 1999.

[7] S. Singh, S. Kaur, "A systematic literature review: Refactoring for disclosing code smells in object oriented software," *Ain Shams Engineering Journal,* vol. 9, pp. 2129-2151, 2018.

[8] M. Mantyla, J. Vanhanen, C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," *International Conference on Software Maintenance*, Amsterdam, The Netherlands, pp. 381-384, 2003.

[9] O. Ellnestam, D. Brolund, *The Mikado Method*, Manning Publications Co., 2014.

[10] P. Gadient, M. Ghafari, P. Frischknecht, O. Nierstrasz, "Security code smells in Android ICC," *Empirical Software Engineering,* vol. 24, no. 5, pp. 3046-3076, 2019.

[11] K. Goseva-Popstojanova, A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology,* vol. 68, pp. 18-33, 2015.

[12] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, M. Vieira, "Benchmarking static analysis tools for web security," *IEEE Transactions on Reliability,* vol. 67, no. 3, pp. 1159-1175, 2018.

[13] H. Mumtaz, M. Alshayeb, S. Mahmood, M. Niazi, "An empirical study to improve software security through the application of code refactoring," *Information and Software Technology,* vol. 96, pp. 112-125, 2018.

[14] J. Li, S. Beba, M. M. Karlsen, "Evaluation of open-source ide plugins for detecting security vulnerabilities," *Proceedings of the Evaluation and Assessment on Software Engineering*, pp. 200-209, 2019.

[15] P. Rachow, "Refactoring decision support for developers and architects based on architectural impact," in *2019 IEEE International Conference on Software Architecture Companion*, Hamburg, Germany, pp. 262-266, 2019.

[16] F. Fontana, V. Lenarduzzi, R. Roveda, D. Taibi, "Are architectural smells independent from code smells? An empirical study," *Journal of Systems and Software,* vol. 154, pp. 139-156, 2019.

[17] A. Rahman, L. Williams, "Source code properties of defective infrastructure as code scripts," *Information and Software Technology,* vol. 112, pp. 148-163, 2019.

[18] A. Kaur, S. Jain, S. Goel, G. Dhiman, "Prioritization of code smells in object-oriented software: a review," *Materials Today: Proceedings,* 2021.

[19] J. dos Reis, F. e Abreu, G. de Figueiredo Carneiro, C. Anslow, "Code smells detection and visualization: a systematic literature review," *Archives of Computational Methods in Engineering,* pp. 1-48, 2021.

[20] A. Kaur, S. Jain, S. Goel, G. Dhiman, "A review on machine-learning based code smell detection techniques in object-oriented software system (s)," *Recent Advances in Electrical and Electronic Engineering,* vol. 14, no. 3, pp. 290-303, 2021.

[21] P. Koch, B. Hofer, F. Wotawa, "On the refinement of spreadsheet smells by means of structure information," *Journal of Systems and Software,* vol. 147, pp. 64-85, 2019.

[22] M. Ghafari, P. Gadient, O. Nierstrasz, "Security smells in Android," in *2017 IEEE 17Th International Working Conference on Source Code Analysis and Manipulation*, Shanghai, China, pp. 121-130, 2017.

[23] A. Gupta, N. Chauhan, "A severity-based classification assessment of code smells in Kotlin and Java application," *Arabian Journal for Science and Engineering,* vol. 47, no. 2, pp. 1831-1848, 2022.

[24] N. Lambaria, T. Cerny, "A data analysis study of code smells within Java repositories," *Annals of Computer Science and Information Systems,* vol. 32, pp. 313-318, 2022.

[25] T. Enumeration, "About CWE," 2021. < https://cwe.mitre.org/about/index.html>

[26] N. Technology, "SAMATE Reference Dataset," 2017. <http://samate.nist.gov/SRD/>

[27] T. Boland, P. Black, "Juliet 1. 1 C/C++ and Java test suite," *Computer,* vol. 45, no. 10, pp. 88-90, 2012.

[28] A. Delaitre, B. Stivalet, P. Black, V. Okun, T. Cohen, A. Ribeiro, "Sate v report: Ten years of static analysis tool expositions," 2018.

[29] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *2013 35th International Conference on Software Engineering*, San Francisco, CA, USA, pp. 672-681, 2013.

[30] T. Xie, *Improving Automation in Developer Testing: State of the Practice*, North Carolina State University, Dept. of Computer Science, 2009.

[31] H. Shahriar, K. Riad, A. Talukder, H. Zhang, Z. Li, "Automatic security bug detection with findsecuritybugs plugin," in *Conference on Cybersecurity Education, Research and Practice*, Kennesaw State University, USA, 2019.

[32] G. Campbell, P. Papapetrou, *SonarQube in Action*, Manning Publications Co., 2013.

[33] M. Adil, I. A. Sumra, "Using model checking to detect SQL injection vulnerability in Java code," *Engineering Science and Technology International Research Journal,* vol. 3, no. 4, pp. 33-41, 2019.

[34] T. Paananen, *Analyzing Java EE Application Security with SonarQube*, Master JAMK University of Applied Sciences, Finland, 2016.