frontiers | Frontiers in Computer Science

TYPE Original Research
PUBLISHED 10 December 2024
DOI 10.3389/fcomp.2024.1455860

OPEN ACCESS

EDITED BY
Marlies Temper,
St. Pölten University of Applied Sciences, Austria

REVIEWED BY
Tomasz Górski,
Gdynia Maritime University, Poland
Philipp Haindl,
St. Pölten University of Applied Sciences, Austria

*CORRESPONDENCE
Yasir Glani
✉ yasirglani@gmail.com
Luo Ping
✉ luop@tsinghua.edu.cn

RECEIVED 27 June 2024
ACCEPTED 06 November 2024
PUBLISHED 10 December 2024

CITATION
Glani Y and Ping L (2024) CodeGuard: enhancing accuracy in detecting clones within java source code.
Front. Comput. Sci. 6:1455860.
doi: 10.3389/fcomp.2024.1455860

COPYRIGHT
© 2024 Glani and Ping. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) and the copyright owner(s) are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.

CodeGuard: enhancing accuracy in detecting clones within java source code

Yasir Glani* and Luo Ping*

Software School, Tsinghua University, Beijing, China

Detecting code clones remains challenging, particularly for Type-II clones, with modified identifiers, and Type-III ST and MT clones, where up to 30% and 50% of code, respectively, are added or removed from the original clone code. To address this, we introduce *CodeGuard*, an innovative technique that employs comprehensive level-by-level abstraction for Type-II clones and a flexible signature matching algorithm for Type-III clone categories. This method requires at least 50% similarity within two corresponding chunks within the same file, ensuring accurate clone identification. Unlike recently proposed methods limited to clone detection, *CodeGuard* precisely pinpoints changes within clone files, facilitating effective debugging and thorough code analysis. It is validated through comprehensive evaluations using reputable datasets, *CodeGuard* demonstrates superior precision, high recall, robust F1 scores, and outstanding accuracy. This innovative methodology not only sets new performance standards in clone detection but also emphasizes the role *CodeGuard*'s can play in modern software development, paving the way for advancements in code quality and maintenance.

KEYWORDS

code clone detection, clone identification, software reliability, code quality assurance, software reuse

1 Introduction

In the dynamic world of software engineering, code cloning has emerged as a focus area of interest, highlighting the complex balance between the advantages and challenges of code reuse (Juergens et al., 2009). This practice, which typically involves copying and reusing code blocks with minimal or no modifications across different parts of a software project, serves as a double-edged sword. On one hand, it can significantly expedites the development process by allowing developers to utilize the existing code snippets . On the other, it poses substantial maintenance challenges and the potential for widespread errors. The discovery of a single error in a fragment of cloned code can necessitate laborious corrections in each occurrence of its reuse, leading to maintenance difficulties, inconsistencies, and an increased higher risk of bugs within the software system (Zakeri-Nasrabadi et al., 2023).

Further highlighting the significance of this issue, research indicates that cloned code comprises approximately 10%–15% of the total codebase in extensive software systems, illustrating its widespread prevalence (Kapser and Godfrey, 2006). Particularly in Common Business Oriented Language (COBOL) systems, the proportion of reuse code is even more significant, reaching around 50% (Ducasse et al., 1999). These findings confirm the prevalent use of code cloning in software development and underline the urgent need for advanced clone detection method. The method should effectively leverage the

advantages of code cloning while minimizing its associated inefficiencies. Consequently, managing code cloning has emerged as a crucial issue in enhancing software quality and maintainability, especially given the complexities of modern software development.

Detecting code clones represents a significant challenge in software development, particularly for methods that are efficient at identifying direct clones but struggle with more complex general clones. Type-I clones mirror the original code with only minor modifications to comments and whitespace, whereas Type-II clones extend these modifications to identifiers and literals. Typically, these direct clone types are detected with some success. However, the complexity significantly escalates with Type-III clones, Which futher divided into Strong Type(ST) and Moderate Type(MT) sub-type, which involve adding or removing upto 30% and %50 respectively of the original code. These advanced clones introduce substantial complexities, incorporating everything from minor modifications to major structural overhauls, thereby evading existed clone detection methods. Type-III (ST and MT) clones, categorized under general clones.

As exemplified in Figure 1, TempConverter conducts a basic Celsius to Fahrenheit conversion and evaluates comfort levels. In contrast, a direct clone named HeatIndexCalculator makes slight modifications—including adjustments to class names, methods, strings, numeric literals, whitespace, and comments—and notably integrates humidity assessment as mentioned in Type-I-II Clone. This functionality alerts users to low and high humidity levels, thereby evolving HeatIndexCalculator into a general clone with significantly broadened functionality by adding up to 41.6% of the new code as mentioned in Type-III Clone.

1.1 Background and motivation

For decades, numerous methods have been proposed to aim at detecting Type-I and Type-II code clones, utilizing text and token-based methods, and have proven effective. However, they struggle with Type-III (ST and MT) clone detection. Despite recent advances in AST, PDG-based, and hybrid detection methods, accurately identifying complex Type-III clone types remains challenging, including inefficiency, inability to detect complex clones, lower true and high false positive rates, and lack of precision in identifying Type-III (ST and MT) modifications within clones. This emphasizes the ongoing research need for a more innovative and effective approach to code clone detection. To fill this research gap, we propose *CodeGuard* which introduces a novel approach designed to detect and identify changes across direct (Type-I and II), general clone (Type-III ST and MT), addressing the significant gaps left by existing techniques and advancing the field of code clone detection.

1.2 Contributions of the paper

This research introduces *CodeGuard*, making several significant contributions:

1. **Obfuscation handling**: *CodeGuard* introduces a multi-level abstraction strategy that effectively detects obfuscated clones and significantly improves Type-II clone detection.
2. **Intelligent matching**: *CodeGuard* leverages an intelligent matching algorithm that boosts detection accuracy and efficiency by focusing on relevant signature chunks rather than the entire database. Through flexible signature matching, it requires at least 50% chunk similarity and two aligned chunks within the same file to classify code as cloned, yielding substantial accuracy improvements, especially in detecting complex Type-III ST and Type-III MT clones.
3. **Post-cloning modifications**: Beyond basic detection, *CodeGuard* identifies changes made to cloned code, highlighting additions, deletions, or modifications with the use of Diff algorithm. This capability enables a nuanced understanding of how cloned code evolves, offering insights valuable for both software maintenance and security.
4. **Comprehensive validation**: Comprehensive evaluations validate that *CodeGuard* consistently surpasses other techniques across precision, recall, F1-score, and accuracy, highlighting its effectiveness and reliability in various clone detection scenarios.

In Section 2, we review the related work. Section 3 introduces the preliminary definitions used in this paper. Section 4 describes our proposed code cloning approach, *CodeGuard*. In Section 5, we detail the clone detection process. Section 6 discusses the evaluation metrics—precision, recall, F1-score, and accuracy—and compares *CodeGuard* with other code cloning methods. Section 7 explores the pros, cons, and limitations of our technique. Finally, Section 8 concludes the paper.

2 Related work

Detecting complex clones across extensive codebases has led to recent advances in clone detection aimed at improving software maintenance. This endeavor seeks to enhance software maintenance by improving the clone technique to precisely identify and locate clones in an extensive database. In recent years, various innovative methods have been proposed to address the nuanced challenges of clone detection. Which include lexical/text-based comparison, which is a foundational technique in clone detection; it analyzes source code as a text for line-by-line comparison. This method identifies exact clone matches through text similarity assessments. Studies (Ragkhitwetsagul and Krinke, 2017; Nakamura et al., 2016) have used these analyses to identify identical code fragments precisely. Yu et al. (2017) multi-granularity technique leverages Java bytecode for detecting Type-I, Type-II, and Type-III clones by converting source code into text. Chen et al. (2015) also employ NICAD for Android clone detection across types. Lyu et al. (2016)'s SuiDroid uses XML layout and the CTPH Hash algorithm for identifying Type-I, II, and III clones. Despite advancements, text-based methods face limitations from strict text matching.

Whereas token-based techniques, proposed by researchers (Glani et al., 2022, 2023; Giani et al., 2022), analyze source code by converting it into tokens for sequence processing to detect clones. Wang et al. (Wang et al., 2018) introduced CCAligner, utilizing

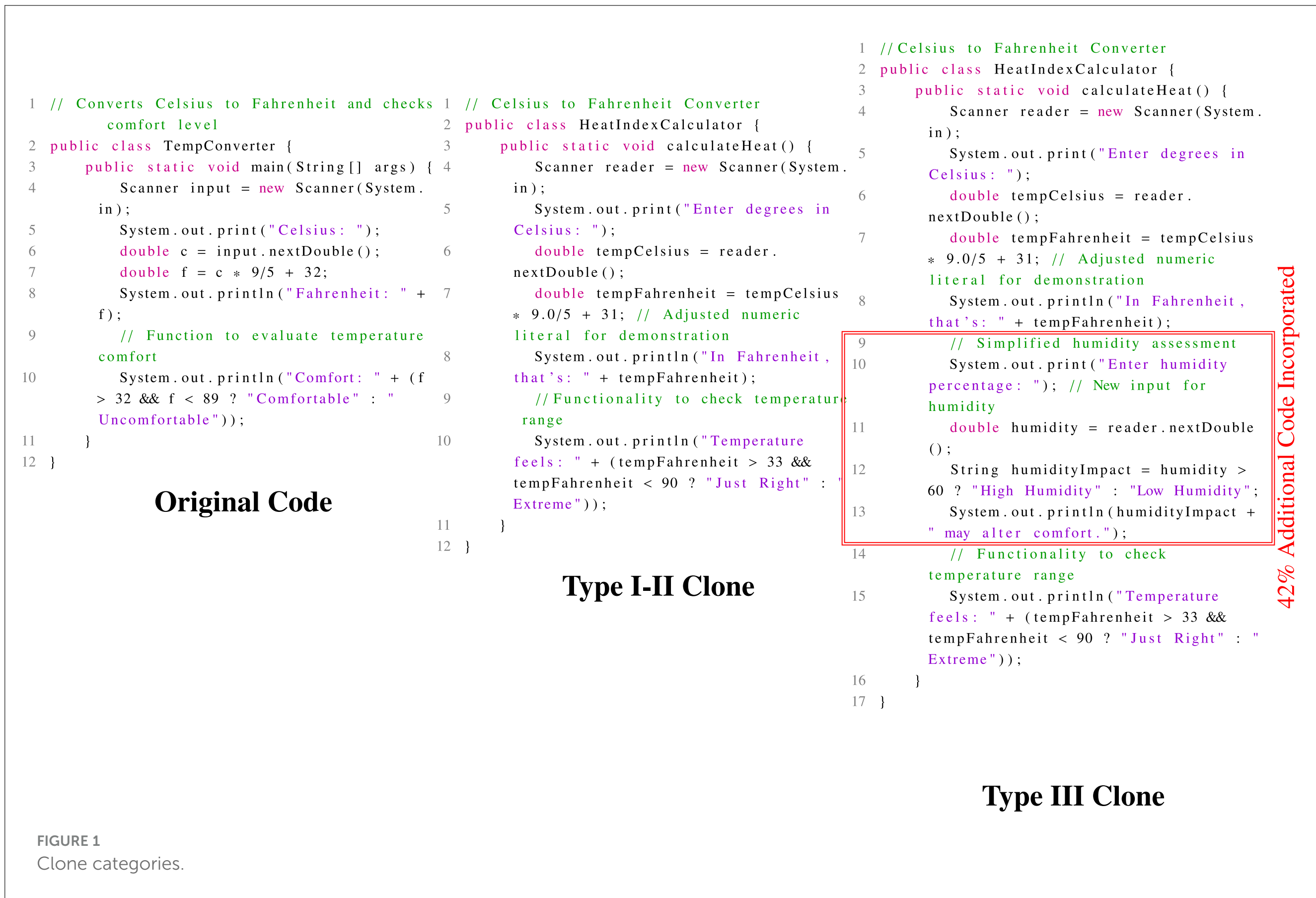


FIGURE 1
Clone categories.

C and Java files to identify Type-I, II, and III clones, leveraging the inherent structure of code. Sajnani et al. (2016) developed SourcererCC, leveraging the IJaDataset and an inverted index for efficient clone query, identifying Type-I, II, and III clones. Chau and Jung (2020) Chau and Jung [13] enhance notation-based code cloning by incorporating an external-based identifier model, which improves the detection of Type-I, Type-II, and Type-III clones. Yuki et al. (2017) proposed a method to detect multi-grained clones using Java files and sequence alignment, demonstrating the adaptability and precision of token-based approaches in clone detection. Akram et al. (2020); Akram and Luo (2021) proposed IBFET and SQVDT, which utilize the ConQat method in the preprocessing stage. The authors later employed a 15-token overlapping chunk size. However, due to the larger chunk size and rigid signature matching, these techniques face limitations in detecting Type-III clones.

Tree-based techniques transform source code into Abstract Syntax Trees (ASTs) to analyze code similarity via tree node matching. Yang et al. (2018) proposed a function-level approach that leverages ASTs with defined node types to generate abstract representations of code, using the Smith-Waterman algorithm for local similarity scoring. This method achieves high precision in detecting cross-project code clones. Chodarev et al. (2015) proposed an AST algorithm for pattern recognition to detect type-I and II clones, and Pati et al. (2017) proposed a method for detecting type-I and II clones and leveraging AST alongside a Multi-Objective Genetic Algorithm (MOGA). Whereas Program Dependency Graph (PDG) techniques excel at detecting code clones by encapsulating both control and data flows within code. These methods generate PDGs and utilize isomorphic subgraph matching to pinpoint similar subgraphs effectively. Wang et al. (2017) introduced CCSharp, which utilizes PDGs alongside Frama-C2 to enhance graph generation. Similarly, Crussell et al.'s AnDarwin leverages large-scale PDGs by employing WALA for effective vector comparison.

Lastly hybrid-based clone detection method integrates two or more textual, token, PDG, and AST approaches to enhance detection capabilities. Singh et al. (2017) developed a technique for converting Java code into AST and PDG formats for identifying Types I, II, and III clones. Misu and Sakib (2017) introduced IDCCD, a combination of token and PDG methods for detecting clones within the IJa-Dataset. Uemura et al. (2017) merged token and metric methods in the Verilog HDL method for clone detection.

3 Preliminary definition

This section explores the fundamental concept of code cloning as proposed by Saini et al. (2018). We then outline our approach, organizing clones into direct and general clone categories. These categories are essential to asses our proposed code cloning technique.

Code fragment: is a sequence of statements, ranging from an entire function to a discrete block of the code, outlined by its source file F_i, with a start line S_{init} and an end line E_{term}.

Code similarity ratio (SR): Compares statement similarity between two fragments C_x and C_y, denoted by $SR_{x,y}$. The variation between $SR_{x,y}$ and $SR_{y,x}$.

Direct clone pair: Involves slight modifications such as whitespace, comments, or identifiers renaming within C_x, with equal length, defined as a direct pair $[C_x, C_y, SR_{x,y}]$.

- **Type-I clone:** A code fragment is considered a Type-I clone when the source code is copied without any modification, except for modifications to comments or empty lines.
- **Type-II clone:** A code fragment is classified as a Type-II clone when the source code is copied with systematic changes to identifiers, such as renaming methods, classes, data types, variable names, string literals, or numeric literals, while maintaining the overall structure and functionality.

Type-I and Type-II clones are types of direct clone pairs.

General clone pair: In this category, C_x originates from C_y but allows for transformations, including line insertions and deletions. Type-III (ST) clones fall into the general clone pair, and modifications in C_x can reach up to 30% of the original code, illustrating the flexibility and complexity of clone classification. Whereas Clone Type-III (MT), also known as large-gap clones, extends beyond Type-III (ST) clones by incorporating significant modifications—statement changes affecting as much as 50% of the original code, as outlined by Higo et al. (2002). Despite their resemblance to Type-III (ST) clones, Type-III MT are distinguished by their scale of modification. Thus, they are evaluated separately within our analysis.

4 Approach

This section outlines our *CodeGuard* technique and our proposed solution for code clone detection. The process begins with downloading and preprocessing reliable datasets, followed by code abstraction at various levels. We then create overlapping code chunks, converting them into unique signatures stored in an indexed pattern in a CloneVault for efficient retrieval. Following that, a diff algorithm is utilized to identify clone patterns and modifications, resulting in a comprehensive clone report. *CodeGuard* functions as a sequential pipeline, with each step building on the previous one. The overview is illustrated in Figure 2, our approach is designed for Java datasets, offering a systematic workflow from data acquisition to clone detection. The detailed preprocessing and processing phases are here as follows:

4.1 Data preparation

This subsection elaborates on the preprocessing phases applied to raw source code, aimed at optimizing it for precise clone detection. These phases systematically enhance the code's quality, forming the groundwork for in-depth analysis. Preprocessing is a fundamental phase in *CodeGuard* for enhancing source code analysis for clone detection, a method emphasized by Li et al. (2014). This phase elevates code quality by:

- **Data cleaning:** Removing outliers to reduce dataset noise.
- **Data reduction:** Streamlining the dataset, preserving its analytical value.
- **Data transformation:** Achieving uniformity through normalization and discretization.
- **Data integration:** Unifying data sources into one database.

An essential aspect of preprocessing involves filtering out non-functional code components that don't contribute to the operational behavior but could affect detection accuracy. These include metadata-centric package declarations, non-executable comments, and white spaces—which might compromise detection efficiency. By removing these unnecessary lines of code, we refine the source code, enhancing its clarity and reducing noise. This process strengthens the robustness and precision of our clone detection technique.

4.1.1 Level-by-level abstraction

In our *CodeGuard* technique, we refine the direct tokenization method of cleaned source code to enhance clone detection, addressing Type-II clone identification challenges. Traditional tokenization methods may overlook actual Type-II clones due to typical modifications like variable or function etc renaming, as they implement direct tokenization. To address this, we employ a comprehensive level-by-level abstraction process, exemplified in Figure 3, with the weather report. Level-by-abstraction enhances *CodeGuard*'s ability to detect clones with common modifications, outlined as follows:

- **Level 0: No Abstraction**: At the initial level, the source code remains in its original cleaned form without any abstraction being implemented.
- **Level 1: Method and Class-Level Abstraction**: At the second level of abstraction, we refine the process by applying abstraction techniques to both method and class levels. After extracting the names of all classes and functions from their respective definition headers, each appearance of a method and class instance in the specific Java file is labeled as "CLASS_NAME" and "METHOD_NAME."
- **Level 2: String, Numerical-Literal and Variable-Level Abstraction**: At this level, we systematically extract the string literals, numeric literals, and variable names from each code file into distinct lists. Every instance of string, numeric literals, and variables within the cleaned code is then labeled with a unique identifier. Variable names are specifically labeled with the particular identifier "VAR_NAME". In contrast, string literals and numeric literals are assigned the identifiers "STRING_LITERAL" and "NUM_LITERAL", respectively.
- **Level 3: Data Type and Parameter-Level Abstraction**: Data types and method parameters are extracted from each code file and stored in a list. Each instance of a data type or method parameter within the code is tagged as "DATA_TYPE" and "PARAM_NAME" respectively. This enhancement strengthens the "CodeGuard" technique, make it resilient against common modifications at both the data type and method parameter levels.

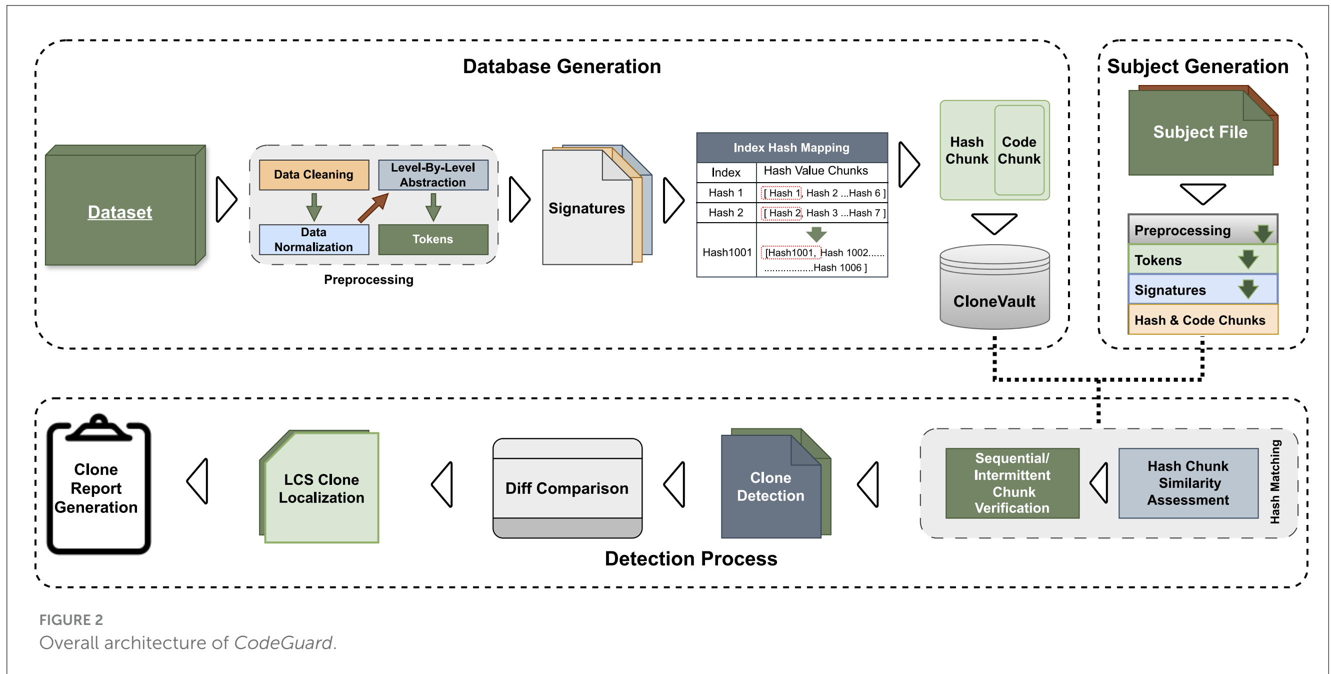


FIGURE 2
Overall architecture of *CodeGuard*.

No Abstraction

```
1  public class WeatherReporter {
2      int temperature;
3      public WeatherReporter(int temp) {
4          this.temperature = temp;
5      }
6      void report() {
7          System.out.println("Temperature: " + temperature + "C");
8      }
9      public static void main(String args) {
10         new WeatherReporter(25).report();
11     }
12 }
```

METHOD and Class-Level Abstraction

```
1  public class CLASS_NAME {
2      int temperature;
3      public CLASS_NAME(int temp) {
4          this.temperature = temp;
5      }
6      void METHOD_NAME() {
7          System.out.println("Temperature:" + temperature + "C");
8      }
9      public static void main(String args) {
10         new CLASS_NAME(25).METHOD_NAME();
11     }
12 }
```

String/NUMERICAL-Literal and Variable-Level Abstraction

```
1  public class CLASS_NAME {
2      int VAR_NAME;
3      public CLASS_NAME(int temp) {
4          this.VAR_NAME = temp;
5      }
6      void METHOD_NAME() {
7          System.out.println(STRING_LITERAL: + VAR_NAME +
8          STRING_LITERAL);
9      }
10     public static void main(String args) {
11         new CLASS_NAME(NUM_LITERAL).METHOD_NAME();
12     }
13 }
```

Data-Type and Parameter-Level Abstraction

```
1  public class CLASS_NAME {
2      DATA_TYPE VARIABLE_NAME;
3      public CLASS_NAME(DATA_TYPE PARAM_NAME) {
4          this.VARIABLE_NAME = PARAM_NAME;
5      }
6      void METHOD_NAME() {
7          System.out.println(STRING_LITERAL + VAR_NAME +
8          STRING_LITERAL);
9      }
10     public static void main(DATA_TYPE PARAM_NAME) {
11         new CLASS_NAME(NUM_LITERAL).METHOD_NAME();
12     }
13 }
```

FIGURE 3
Level-by-level abstraction implementation.

Java keywords are remained same over all abstractions. Variables, strings, numeric literals, methods, and function names are frequently modified by developers to meet their specific code task requirements. Our level-by-level abstraction methodology is designed to detect common modified code and especially enhance the ability to detect Type-II clones.

4.2 Token generation

The abstraction of cleaned code is tokenised using semicolons, instead of new line (\n), to form tokens, addressing the issue of high false positive ratios when splitting by new lines, which often misidentifies non-clones as clones. Tokens are transformed into overlapping chunks with six-token window size, e.g., the first

chunk spans tokens 1–6 tokens, and the second chunk contain 2–7 token etc, a crucial step in *CodeGuard*. The chunk size directly affects database generation time, detection speed, and clone identification accuracy. Larger chunks increase preprocessing and database generation time, potentially decreasing accuracy, while too small chunks may raise the false positive rate.

4.3 Signature generation

Overlapping chunks are converted into xxHash64 signatures instead of transforming into MD5, FNV, and Murmur traditionally used hash signature for code cloning, known for their slower performance due to multiple rounds of compression as outlined in hash algorithm benchmarks.[1] The xxHash64 algorithm, proposed by Startin (2019), is chosen for its exceptional efficiency, generating signatures at RAM speed limits. This selection is paramount, not just for the algorithm's efficiency, but also for its quality and its low collision rate of 10^{18}, making identical signatures in a vast database nearly impossible. The xxHash64 algorithm begins with any 64-bit seed value, processing ASCII byte sequences in 32-byte chunks. Through several data mixing rounds and bitwise operations, it updates internal accumulators to capture the data's pattern, culminating in a single, highly reliable 64-bit hash value, which can be either in decimal form or hexadecimal form.

4.4 CloneVault

Signatures of overlapping chunks are stored in the CloneVault a specialized database table. It features four columns designed for efficient storage and retrieval:

- Entry ID: A unique identifier.
- Initial Signature: First signature value of each chunk, key to efficient indexing.
- Chunk Signature: Overlapping chunk's signature.
- File Path: The corresponding file path.

Efficient indexing, particularly of the Initial Signature, significantly accelerates clone retrieval. This database architecture facilitates streamlined detection, identification, and retrieval of code clones, boosting the overall effectiveness and efficiency of *CodeGuard*.

5 Clone detection process

For clone detection, we first subject the code to comprehensive preprocessing, involving data cleaning, normalization, and refinement, to strip away unnecessary lines of code and normalize it for deeper analysis. Subsequent to this step, we apply a systematic level-by-level abstraction to the code. Subsequently, we generate overlapping code chunks based on a six-line threshold and create xxHash64 signatures for these chunks. These signatures are subsequently stored and ready for comparison against our CloneVault to identify similarities effectively.

5.1 Clone identification

Our clone detection methodology efficiently compares subject code signatures with a pre-indexed CloneVault, eliminating the need to search through the entire database. By matching the first signature of each subject's overlapping chunk with the corresponding pre-indexed signature in the CloneVault, we then compare both sets of overlapping chunks for similarity. A successful match requires at least a 50% alignment in chunk signatures between the subject code and CloneVault entries. We use the following formula to determine match criteria:

$$SR_{x,y} = \begin{cases} 1, & \text{if } \exists\, 2\, H_{C_x} = H_{C_y} \in \text{DB with match} \geq 50\% \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where $SR_{x,y}$ denotes the similarity ratio between code fragments C_x and C_y, and H_{C_x} is the subject hash chunk and H_{C_y} is the database hash chunk. For a successful clone match, at least two chunks within the same file must surpass this 50% similarity threshold, regardless of their sequential order. This flexible matching allows us to identify clones with a high degree of similarity, even amidst significant code modification, including all sub-types of Type-III clones.

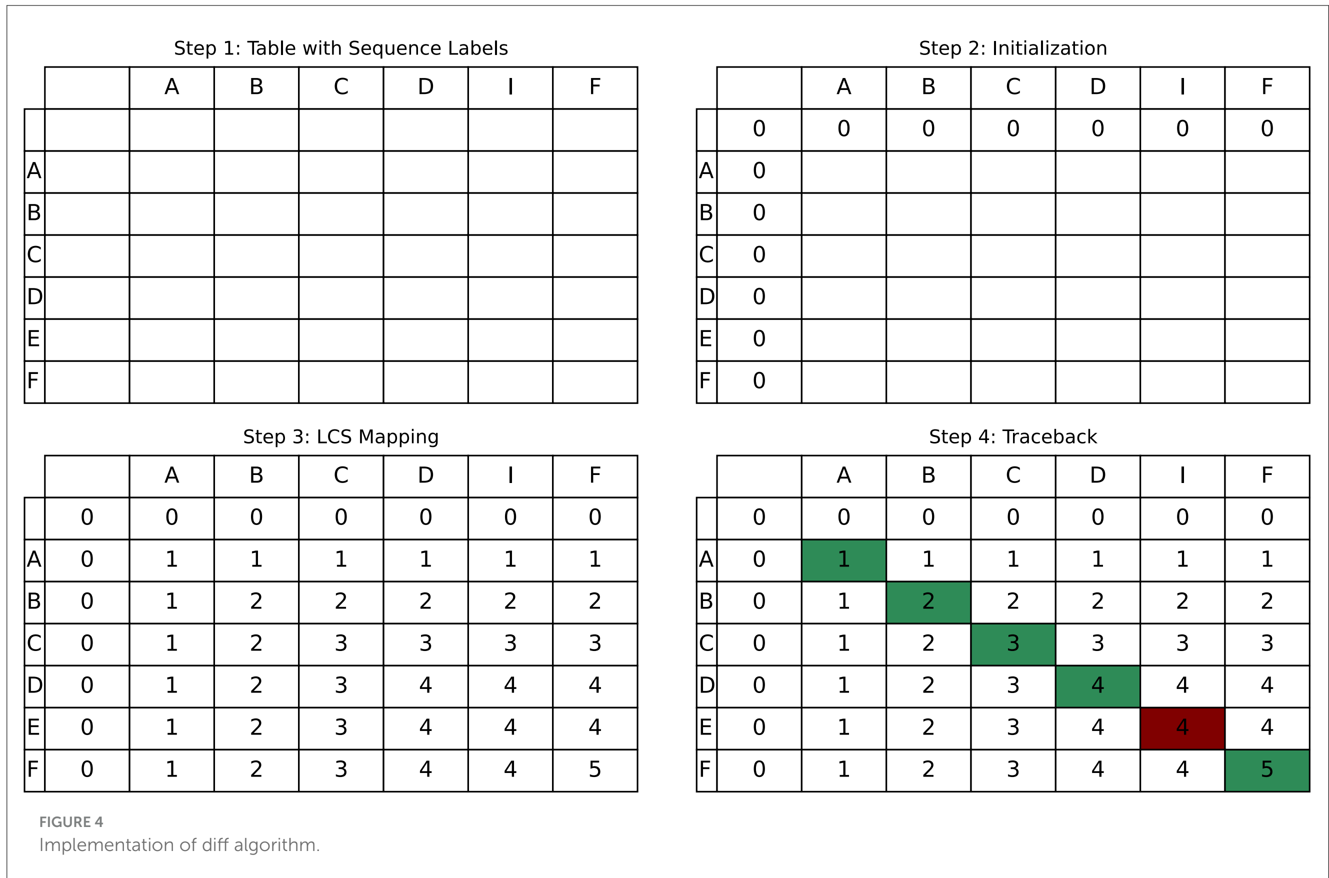
5.2 Precise detection using diff algorithm

After detecting potential clones, we utilize the Diff algorithm (Myers, 1986) to pinpoint changes made within copied code segments, advancing beyond existing clone detection techniques. This method excels at pinpointing modifications in Type-III sub-type clones by identifying the longest common subsequence (LCS) within the code, employing a dynamic programming approach for comparison. It begins the comparison of two sequences $X = [x_1, x_2, x_3, \ldots, x_m]$ and $Y = [y_1, y_2, y_3, \ldots, y_n]$, where X is the subject file, and Y is the java code file code stored in the database. Diff algorithm comprises four key steps, outlined as follows:

- **Step 1: Initialize the table**
  - Create a table L of size $(m + 1) \times (n + 1)$.
  - $L[i][j]$ stores the length of the LCS of $X[1..i]$ and $Y[1..j]$.
  - Set all $L[i][j]$ to 0, indicating no common subsequence found yet.

  A table L is set up with dimensions $(m + 1) \times (n + 1)$ for sequences X and Y, initializing all cells to zero, indicating no LCS found initially.
- **Step 2: Filling the table**
  - For each character comparison between X and Y, update $L[i][j]$ accordingly.
  - If $x_i = y_j$, then $L[i][j] = L[i-1][j-1] + 1$.

1 *Benchmarking the performance of hash functions*. Available at: https://github.com/Cyan4973/xxHash/tree/dev.



Step 1: Table with Sequence Labels

		A	B	C	D	I	F
A							
B							
C							
D							
E							
F							

Step 2: Initialization

		A	B	C	D	I	F
	0	0	0	0	0	0	0
A	0						
B	0						
C	0						
D	0						
E	0						
F	0						

Step 3: LCS Mapping

		A	B	C	D	I	F
	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
B	0	1	2	2	2	2	2
C	0	1	2	3	3	3	3
D	0	1	2	3	4	4	4
E	0	1	2	3	4	4	4
F	0	1	2	3	4	4	5

Step 4: Traceback

		A	B	C	D	I	F
	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
B	0	1	2	2	2	2	2
C	0	1	2	3	3	3	3
D	0	1	2	3	4	4	4
E	0	1	2	3	4	4	4
F	0	1	2	3	4	4	5

FIGURE 4
Implementation of diff algorithm.

- Else, $L[i][j] = \max(L[i-1][j], L[i][j-1])$, capturing the longest subsequence found so far.

Each character of X and Y is compared, incrementing $L[i][j]$ for matches, else take the max value from adjacent cells for mismatches.

- **Step 3: Traceback for LCS**
  - Starting from $L[m][n]$, trace back the path that led to the LCS.
  - Move diagonally back when a match ($x_i = y_j$) contributed to the current LCS length, indicating a part of the LCS. Else, move up or left, depending on which direction has the larger value.

Starting from $L[m][n]$, a traceback process determines the LCS, indicating matches and mismatches and guiding the LCS extraction.

- **Step 4: Extracting the LCS**
  - During traceback, when a diagonal move is made, record the matching character as part of the LCS.
  - Continue tracing back until reaching the start of the table ($L[0][0]$), compiling the LCS from the recorded characters.

The LCS is extracted by following the traceback path, where diagonal moves indicate matches included in the LCS.

The Diff algorithm efficiently compares Java code lines represented by sequences X = "ABCDEF" and Y = "ABCDIF," as shown in Figure 4. The algorithm initiates by setting up an empty matrix, progressing to populate it based on character matches and mismatches, following the LCS rules. Consecutive matches increase the value in the matrix, leading to an LCS value of 5, signifying five aligned characters. The analysis identifies "ABCDF" as the longest common subsequence and highlights "IE" as a variation, potentially marking a modified segment in cloned code. The green cells signify matched characters, and the red cells denote differences used to trace back the LCS, revealing the sequence "ABCDF" as the common subsequence and "IE" indicating cloned code modifications. Diff algorithm enhances the *CodeGuard* technique's ability to accurately pinpoint modifications within complex clone types.

6 Evaluation

In our comparative evaluation, *CodeGuard* was assessed alongside prominent token-based and hybrid token-based methods, including DroidMD (Akram et al., 2021), Vuddy (Kim et al., 2017), and AYAT (Giani et al., 2022), VCIPR (Akram et al., 2019) which align closely with our hybrid token-based method. To highlight *CodeGuard*'s versatility and provide a comprehensive assessment, we broadened our analysis to encompass various methodologies, such as the hybrid-PDG method proposed by Song et al. (2020), the hybrid-AST based method by Yang et al.

TABLE 1 Datasets.

Dataset	Files	Methods	Lines
D4J-Dataset	85,173	278,110	10,631,318
IJA-Dataset	100,000	252,022	51,763,980
GTH-Dataset	100,000	172,307	7,768,509

(2018), and text-based technique ICDT.[2] This comprehensive comparative analysis emphasizes *CodeGuard*'s robustness across different cloning detection methods.

6.1 Dataset

To assess the effectiveness of our *CodeGuard* technique, we conducted an evaluation using the IJADataset (Svajlenko and Roy, 2015), GTH-Dataset (Allamanis and Sutton, 2013), and D4J-Dataset (see footnote 2) datasets highly regarded in the field of code clone detection. As mentioned in Table 1, IJA-Dataset ranges from the smallest file containing 50 lines of Java code to the largest comprising 1,600 lines, summing up to a total of 51,763,980 lines across 1,593,159 methods.

The D4J-Dataset includes 85,173 Java files, 278,110 methods, and 10,631,318 lines of code. The GTH-Dataset contain 100,000 Java files, 172,307 methods and 7,768,509 lines of code. The IJA-Dataset is approximately 4.87 times larger than the D4J-Dataset and 6.66 times more than GTH-Dataset in terms of the number of lines of code, ensuring a robust evaluation of *CodeGuard* against diverse and extensive codebases.

6.2 Experimental setup

To assess our code clone detection technique, we established a testing environment with 200 Java files, initially categorized as non-clones. We then selected 100 diverse functions, ranging from 10 to 150 lines of code, from our diverse dataset and organized them into five distinct groups to represent a range of clone types. This setup facilitated the simulation of both direct and general clone modifications across all Type-I, Type-II, Type-III ST, Type-III VST, and Type-III MT (LG) types.

For Type-I modifications, comments and empty lines are systematically modified at specific intervals to simulate uniform changes without bias. For Type-II, identifiers like class names, method names, parameters, constructors, string literals, and variables are systematically renamed, whereas numerical literals are incremented to ensure modifications are evenly spread across files.

For Type-III ST (0%–10%), Type-III VST (11%–30%), and Type-III MT (31%–50%), modifications involve inserting blocks of code within the original code based on predefined systematic intervals. These modifications correspond to modifying a specified percentage of the original code's length, ensuring systematic variations across the dataset. Each modification type adheres to a patterned approach, using calculated intervals or patterns for modifications, thus minimizing randomness and providing a consistent, unbiased modification across all the files.

Following that, we injected 100 diverse functions representing each five clone types—Type-I, Type-II, Type-III ST, Type-III VST, and Type-III MT (LG)—into subsets of Java files, initially non-clones. Across five groups, each with 200 files, functions are injected into 100 files (each initially containing 200 non-clone Java files) per group, thus transforming them into known clones. This selection process ensured a comprehensive assessment across clone types. The injection was carefully executed to ensure that the functions were injected at syntactically appropriate positions within the code to maintain the integrity of the original file's structure while adding new functions.

The selection and injection of clone functions into non-clone files were methodically executed by shuffling and pairing each non-clone file with a unique clone function from the respective group, ensuring an equitable distribution and minimizing bias. This deliberate methodology ensures that each modified file, now prefixed with "Clone-," accurately reflects its cloned status (e.g., "Clone-Example.java"), while the rest remain as non-clones. This systematic approach simulates realistic software development scenarios.

6.2.1 Evaluation metrics analysis

In this section, we conduct a comprehensive evaluation of clone detection evaluation metrics, placing our *CodeGuard* technique in direct comparison with leading clone detection techniques. By leveraging essential evaluation metrics such as precision, recall, F1-score, accuracy, and detection time, it prepares the groundwork for an extensive analysis of these performance indicators in the context of code cloning. The comprehensive findings, summarized in Tables 2, 3 and average results graphical representation presented in Figure 5, set the stage for an in-depth discussion on these metrics. This analysis not only emphasizes the competitive performance of *CodeGuard* but also sheds light on its efficiency and effectiveness in identifying code clones, distinguishing it from other techniques.

Precision

Precision is a critical metric in clone detection that measures the proportion of true identifications (TP, correctly identified clones) against the sum of true positives and false positives (FP, incorrect clone identifications). It is calculated as:

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}} \quad (2)$$

High precision indicates a technique's ability to correctly classify clones, highlighting its specificity in distinguishing true clones (TP) from false positives (FP). Our *CodeGuard* technique consistently demonstrated exceptional precision across various clone types, averaging results from the IJA-Dataset, D4J-Dataset, and GTH-Dataset. For Type-I clones, *CodeGuard* achieved a precision of 0.98, which is 1.18 times better than AYAT (0.833) and more than double

2 *D4j code dataset*. Available at: https://www.kaggle.com/datasets/zavadskyy/lots-of-code?select=java.txt.

TABLE 2 Evaluation metrics for direct clones, where D.Time denotes detection time.

Techniques		Code clone Type-I					Code clone Type-II				
Technique	Dataset	Precision	Recall	F1-Score	Accuracy	D.Time	Precision	Recall	F1-Score	Accuracy	D.Time
0.90 Our technique	D4J-Dataset	0.99	1.0	0.99	0.99	1.08	0.99	1.0	0.99	0.99	1.07
0.95	IJA-Dataset	1.00	0.95	0.97	0.97	4.83	1.00	0.95	0.97	0.97	4.82
0.90	GTH-Dataset	0.95	1.00	0.97	0.97	0.79	0.95	1.0	0.97	0.97	0.85
Vuddy	D4J-Dataset	1.00	0.91	0.95	0.91	0.28	1.00	0.89	0.94	0.89	0.21
	IJA-Dataset	0.51	0.54	0.52	0.51	0.90	0.51	0.54	0.52	0.51	0.55
	GTH-Dataset	0.52	0.59	0.55	0.53	0.44	0.52	0.59	0.55	0.52	0.27
Song et al.	D4J-Dataset	0.65	0.93	0.77	0.71	0.79	0.65	0.94	0.77	0.72	0.78
	IJA-Dataset	0.61	0.84	0.71	0.66	2.10	0.51	0.55	0.53	0.51	2.18
	GTH-Dataset	0.55	0.67	0.61	0.56	0.63	0.52	0.60	0.56	0.53	0.59
Yang et al.	D4J-Dataset	0.46	0.63	0.53	0.44	1.46	0.44	0.58	0.50	0.41	1.13
	IJA-Dataset	0.48	0.60	0.53	0.47	5.22	0.49	0.63	0.55	0.48	4.71
	GTH-Dataset	0.47	0.65	0.55	0.47	0.86	0.49	0.69	0.57	0.47	0.83
VCIPR	D4J-Dataset	0.57	0.04	0.07	0.51	0.92	0.57	0.04	0.51	0.97	0.62
	IJA-Dataset	0.38	0.05	0.09	0.48	2.11	0.43	0.06	0.11	0.49	0.34
	GTH-Dataset	0.97	0.36	0.53	0.68	0.47	0.83	0.05	0.09	0.52	0.47
ICDT	D4J-Dataset	0.53	0.57	0.55	0.54	0.54	0.51	0.52	0.51	0.51	0.23
	IJA-Dataset	0.56	0.56	0.56	0.14	0.53	0.49	0.43	0.46	0.49	0.24
	GTH-Dataset	0.60	0.64	0.62	0.61	0.48	0.59	0.58	0.58	0.58	0.19
AYAT	D4J-Dataset	0.83	0.86	0.84	0.84	3.40	0.83	0.90	0.87	0.86	2.79
	IJA-Dataset	0.86	1.00	0.93	0.92	44.91	0.70	0.30	0.49	0.61	61.16
	GTH-Dataset	0.81	1.00	0.90	0.89	1.82	0.81	1.00	0.90	0.89	1.49
DroidMD	D4J-Dataset	0.62	0.08	0.14	0.52	0.93	0.64	0.09	0.16	0.52	0.40
	IJA-Dataset	0.78	0.14	0.24	0.55	3.24	0.43	0.03	0.06	0.49	1.15
	GTH-Dataset	0.92	0.44	0.59	0.70	0.29	0.88	0.28	0.42	0.62	0.28

the precision of Yang et al., showing an improvement of 2.09 times (0.98 vs. 0.47). In Type-II clones, it was 1.26 times more precise than AYAT (0.98 vs. 0.78) and exhibited a 1.85 times enhancement over ICDT (0.98 vs. 0.53). For Type-III S clones, it surpassed Song et al. by 1.77 times (0.98 vs. 0.553) and outperformed Vuddy by 1.45 times (0.98 vs. 0.676). Additionally, *CodeGuard* improved its precision over Song et al. by 1.75 times (0.98 vs. 0.56) and maintained a 1.18 times lead over AYAT (0.98 vs. 0.833). In the complex Type-III MT clones, *CodeGuard* demonstrated an improvement of 2.14 times over Yang et al. (0.976 vs. 0.456) and exceeded Vuddy by 1.45 times (0.976 vs. 0.673).

Recall

Recall is an important metric for evaluating clone detection performance, focusing on a technique's ability to identify all true clones within a dataset. It measures the proportion of actual clones detected, with higher recall indicating a technique's enhanced ability to detect all true clones. Whereas a false negative (FN) represents a true clone that is incorrectly identified as a non-clone. Recall is calculated using the following formula:

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}} \tag{3}$$

In evaluating recall, *CodeGuard* consistently outperformed across all clone types. Averaging results from the IJA-Dataset, D4J-Dataset, and GTH-Dataset, *CodeGuard* achieved a recall of 0.983 for Type-I clones, surpassing the highest recall of AYAT by 1.03 times (0.983 vs. 0.953) and exceeding the lowest recall of VCIPR and DroidMD by 6.5 times and 4.5 times, respectively (0.98 vs. 0.15 and 0.22). For Type-II clones, it maintained dominance with a recall of 0.983, outperforming AYAT by 1.34 times (0.983 vs. 0.733) and exceeding the recall of VCIPR and DroidMD by approximately 20 times and 7.5 times, respectively (0.98 vs. 0.05 and 0.13). In Type-III ST clones, *CodeGuard* achieved a recall of 0.95, exceeding AYAT by 1.02 times (0.95 vs. 0.933) and outperforming VCIPR and DroidMD by approximately 24 times and 7.9 times, respectively (0.95 vs. 0.04 and 0.12). For Type-III MT clones, *CodeGuard* showed a recall of 0.927, slightly surpassing AYAT by 0.98 times

TABLE 3 Evaluation metrics for general clones, where D.Time denotes detection time.

Techniques		Code clone type-III-ST					Code clone type-III-MT				
Technique	Dataset	Precision	Recall	F1-Score	Accuracy	D.Time	Precision	Recall	F1-Score	Accuracy	D.Time
0.90 Our technique	D4J-Dataset	0.99	1.0	0.99	0.99	1.13	0.99	1.0	0.99	0.99	1.44
0.95	IJA-Dataset	1.00	0.90	0.94	0.95	4.92	1.00	0.89	0.94	0.94	4.90
0.90	GTH-Dataset	0.95	0.95	0.95	0.95	0.80	0.94	0.89	0.91	0.92	0.82
Vuddy	D4J-Dataset	1.00	0.86	0.92	0.86	0.28	1.00	0.83	0.91	0.83	0.29
	IJA-Dataset	0.51	0.55	0.53	0.52	0.20	0.51	0.55	0.53	0.52	0.21
	GTH-Dataset	0.51	0.58	0.54	0.52	0.28	0.51	0.57	0.54	0.51	0.27
Song et al.	D4J-Dataset	0.65	0.92	0.76	0.71	0.70	0.64	0.89	0.74	0.69	0.74
	IJA-Dataset	0.51	0.56	0.54	0.52	2.25	0.51	0.56	0.54	0.52	2.35
	GTH-Dataset	0.52	0.59	0.55	0.52	0.61	0.51	0.57	0.54	0.51	0.58
Yang et al.	D4J-Dataset	0.43	0.57	0.49	0.41	1.14	0.44	0.58	0.50	0.41	1.12
	IJA-Dataset	0.48	0.61	0.54	0.47	13.73	0.46	0.57	0.51	0.46	5.22
	GTH-Dataset	0.48	0.66	0.55	0.47	0.83	0.47	0.63	0.54	0.46	0.80
VCIPR	D4J-Dataset	0.57	0.04	0.07	0.51	0.64	0.57	0.04	0.07	0.51	0.61
	IJA-Dataset	0.33	0.04	0.07	0.48	0.37	0.43	0.06	0.11	0.49	0.45
	GTH-Dataset	0.80	0.04	0.08	0.52	0.50	0.75	0.03	0.06	0.51	0.49
ICDT	D4J-Dataset	0.53	0.56	0.54	0.53	0.27	0.54	0.59	0.56	0.55	0.21
	IJA-Dataset	0.52	0.48	0.50	0.52	0.24	0.52	0.48	0.50	0.52	0.20
	GTH-Dataset	0.53	0.47	0.50	0.53	0.19	0.53	0.46	0.49	0.53	0.18
AYAT	D4J-Dataset	0.83	0.86	0.84	0.82	3.82	0.83	0.89	0.86	0.82	8.94
	IJA-Dataset	0.86	0.99	0.92	0.92	61.03	0.86	0.99	0.92	0.92	62.02
	GTH-Dataset	0.81	0.95	0.87	0.86	1.47	0.81	0.95	0.87	0.86	1.52
DroidMD	D4J-Dataset	0.62	0.08	0.14	0.52	0.43	0.64	0.09	0.16	0.52	0.37
	IJA-Dataset	0.60	0.06	0.11	0.51	1.58	0.69	0.09	0.16	0.53	1.62
	GTH-Dataset	0.85	0.23	0.36	0.59	0.29	0.85	0.22	0.35	0.59	0.29

(0.927 vs. 0.943) and significantly outperforming VCIPR and DroidMD by approximately 23 times and 7.2 times, respectively (0.93 vs. 0.04 and 0.13). These results emphasize *CodeGuard*'s unmatched proficiency in precise clone identification.

F1-score

The F1-score is an essential metric for assessing the performance of code clone detection techniques, harmonizing precision and recall to evaluate overall efficacy. It adeptly balances accurately identifying true clones with ensuring the complete detection of clones within a dataset. Defined by the harmonic mean of precision and recall, the F1-score is calculated using following equation:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4)$$

The F1 Score combines precision and recall into a singular metric, offering an all-encompassing evaluation of a technique's performance. Thus, the F1 Score stands as a pivotal indicator of the overall effectiveness and reliability of a clone detection technique. In assessing F1-scores, *CodeGuard* outperforms across all clone types, demonstrating its exceptional capability in balanced clone detection. Averaging results from the IJA-Dataset, D4J-Dataset, and GTH-Dataset, for Type-I clones, *CodeGuard* achieved an F1-score of 0.976, surpassing AYAT by 1.10 times (0.976 vs. 0.89) and vastly outperforming the lowest F1-scores of VCIPR and DroidMD by approximately 4.3 times and 3.1 times, respectively (0.98 vs. 0.23 and 0.32). In Type-II clones, *CodeGuard* achieved an F1-score of 0.98, significantly exceeding the lowest F1-scores of VCIPR and DroidMD by approximately 4.1 times and 4.7 times, respectively (0.98 vs. 0.24 and 0.21), and surpassing AYAT by 1.3 times (0.98 vs. 0.75). Within Type-III ST clones, *CodeGuard* achieved a 0.96 F1-score, outperforming the highest F1-score of AYAT by 1.10 times (0.96 vs. 0.877), and exceeding the lowest F1-scores of VCIPR and DroidMD by approximately 13.7 times and 4.8 times, respectively (0.96 vs. 0.07 and 0.20). For Type-III MT clones, *CodeGuard* consistently achieved an F1-score of 0.947, surpassing AYAT by 1.07 times (0.947 vs. 0.883) and

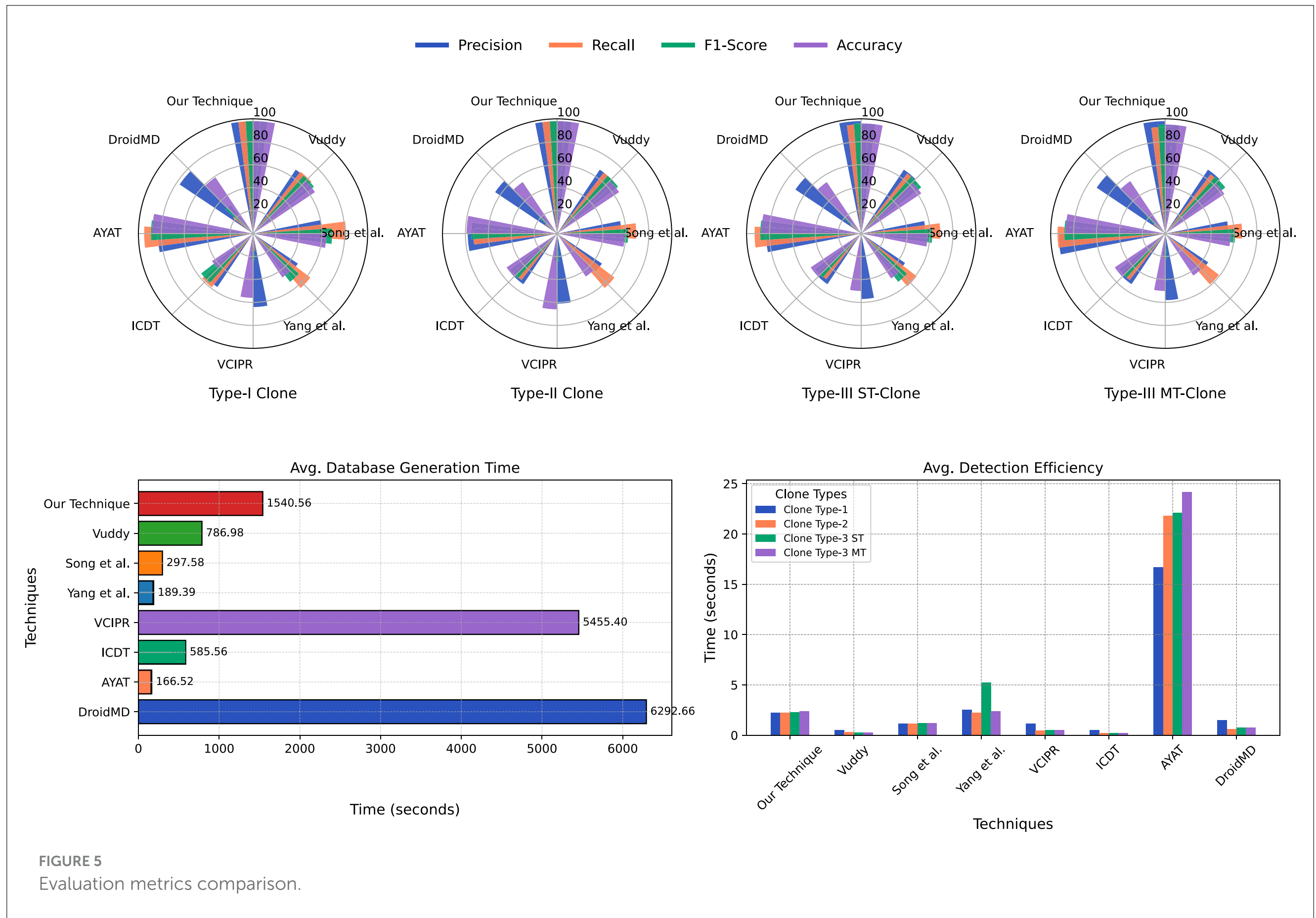


FIGURE 5
Evaluation metrics comparison.

outperforming the lowest F1-scores of VCIPR and DroidMD by approximately 11.9 times and 4.3 times, respectively (0.95 vs. 0.08 and 0.22), showcasing its superior performance in complex clone detection scenarios.

Accuracy

Accuracy plays a key role in clone detection, illustrating a technique's ability to correctly classify both cloned and non-cloned code segments. It encompasses true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN), offering a comprehensive measure of detection performance. Accuracy is computed using the following equation:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (5)$$

A high accuracy level demonstrates a technique's reliability in clone detection, which is essential for minimizing misclassifications and ensuring the integrity of the technique. In our accuracy assessment, *CodeGuard* leads in clone detection accuracy across all clone types. Averaging results from the IJA-Dataset, D4J-Dataset, and GTH-Dataset, for Type-I clones, *CodeGuard* achieved a notable accuracy of 0.977, surpassing AYAT by 1.11 times (0.977 vs. 0.883) and far exceeding the lowest accuracy of ICDT by 2.27 times (0.977 vs. 0.43). For Type-II clones, it achieved a high accuracy of 0.977, outperforming AYAT by 1.24 times (0.977 vs. 0.787) and the lowest accuracy of ICDT by 1.86 times (0.977 vs. 0.527). In Type-III ST clones, *CodeGuard* achieved an accuracy of 0.963, vastly outperforming Vuddy's 0.633 by 1.52 times (0.963 vs. 0.633) and exceeding AYAT by 1.11 times (0.963 vs. 0.867). This demonstrates its precision in complex clone detection. For Type-III MT clones, *CodeGuard* achieved an accuracy of 0.95, surpassing AYAT by 1.10 times (0.95 vs. 0.867) and the lowest accuracy of Yang et al. by 2.14 times (0.95 vs. 0.443). These results highlight *CodeGuard*'s superior capability in detecting clones, showcasing its effectiveness in clone detection.

Efficiency

An effective clone detection technique is characterized by near-perfect precision, recall, F1 score, accuracy, and efficiency. Averaging results from the IJA-Dataset, D4J-Dataset, and GTH-Dataset, *CodeGuard* demonstrated notable efficiency across all clone types, with detection times averaging 2.23 seconds for Type-I, 2.25 seconds for Type-II, 2.28 seconds for Type-III ST, and 2.39 seconds for Type-III MT clones. In contrast, AYAT's times ranged from 16.71 to 24.16 seconds, indicating significantly less efficiency in clone detection. Yang et al.'s average times varied from 2.22 to 5.23 seconds, slightly exceeding *CodeGuard*'s, yet

demonstrating commendable efficiency. Other methods like Vuddy (0.26 to 0.54 seconds), Song et al. (1.17 to 1.22 seconds), and DroidMD (0.61 to 1.49 seconds) were more efficient, benefiting from a function-based approach that significantly reduced the volume of code for comparison and exact chunks and function signatures matching. In our analysis, the AYAT technique had the longest detection times among all techniques, ranging from 16.71 seconds to 24.16 seconds. However, despite their rapid efficiency of techniques, they sacrificed evaluation metrics regarding precision, recall, F1-score, and accuracy due to variations in the clone type. Although *CodeGuard* exhibits slightly longer detection times than the fastest methods, it effectively balances accuracy and detection efficiency. Its exceptional precision, recall, F1-score, and accuracy ensure reliable clone detection, essential for complex software development and maintenance.

CodeGuard efficiently processed a comprehensive database in just 1,540.56 seconds on average, emphasizing its effectiveness. *CodeGuard* completed the database generation slightly slower than Vuddy (786.98 seconds) and Song et al. (297.58.4 seconds). This is due to the comprehensive preprocessing, including abstraction, which enables our technique to detect clones, especially Type-II clones, effectively. Although AYAT and Yang et al. reported more rapid generation times of 166.52 and 189.39 seconds, *CodeGuard*'s performance remains competitive. Even though our technique takes slightly longer to generate the database, it performs well in evaluation metrics such as precision, recall, F1-score, and accuracy and is more efficient than DroidMD (6,292.66 seconds) and VCIPR (5,455.4). Since database generation is a one-time process, the slightly longer generation time can be overlooked.

7 Discussion and limitations

Code cloning techniques should be implemented in software development only when they demonstrably enhance key performance indicators such as precision, recall, F1-score, and accuracy. These criteria are essential for reducing false identifications while accurately detecting both clone and non-clone segments.

CodeGuard distinguishes itself with an advanced preprocessing phase, meticulously removing whitespaces and comments to enhance Type-I clone detection, and applies a comprehensive level-by-level abstraction, covering variables, data types, literals, methods, and classes for precise Type-II clone identification. To reduce false positives through comprehensive level-by-level abstraction, semi-colons are utilized for tokenizing preprocessed code, replacing the existed methods of splitting code by newline. *CodeGuard* excels in its processing phase by employing an advanced matching algorithm targeting a 50% to 100% similarity range for Type-III clones (ST and MT). It uses an efficient index-based chunk-matching algorithm that begins by comparing the first signature of each subject chunk against the database's first signatures. When a match is found, it then ensures that corresponding chunks share a minimum of 50% similarity across at least two chunks from the same file, irrespective of their sequence. Beyond mere detection, *CodeGuard* utilizes the Diff Algorithm to pinpoint the changes made in clone code, offering invaluable insights for maintenance and debugging in software maintenance.

*CodeGuard*effectively detects code clones by achieving high precision, recall, F1-score, and accuracy. However, this comes at the cost of efficiency. Compared to other tools like VUDDY, ICDT, DroidMD, and VCIPR, *CodeGuard* takes longer time to detect clones. While these tools are efficient at clone detection, they fall short of delivering the same level of accuracy and overall performance in evaluation metrics. Another limitation is CodeGuard's language dependency. It is specifically designed for Java, which may restrict its adaptability to other programming languages. Different languages, with varying syntax and structure, may require significant changes to the preprocessing and chunk-matching algorithms to maintain the same level of clone detection accuracy.

Whereas Vuddy's technique (Kim et al., 2017) for clone detection initially involves extracting code functions and applying limited abstraction to the preprocessed code. This selective abstraction may lead to false positives, mistakenly identifying small non-clone functions as clones. Vuddy uses MD5 hashes for function signature generation, which pose challenges in handling code modifications like insertions or deletions, prevalent in Type-III (ST and MT) clones, due to resulting hash value changes. The ICDT technique (see foot note 2) demonstrated strong performance in clone detection due to its use of smaller chunk sizes, which significantly influence precision, recall, F1-score, and accuracy. Appropriate chunk size is essential; if it is too small, it may incorrectly flag non-clones as clones; if it is too large, true clones might be missed. For instance, the recently proposed VCIPR technique underperformed across all datasets. This poor performance was primarily due to its reliance on exact chunk signature matches and its use of larger chunk sizes, such as a window size of 15.

AYAT (Giani et al., 2022) shows robust performance in detecting Type-I and Type-III (ST and MT) clones. It attributes its success to adopting smaller chunk sizes, significantly achieving good precision, recall, F1 score, and accuracy. Appropriate chunk size is essential; overly small chunks can lead to false positives by flagging non-clones as clones, whereas excessively large ones may miss true clones. Despite its commendable metrics due to small chunk adoption, AYAT's absence of abstraction affected its ability to detect Type-II clones effectively. Whereas (Song et al., 2020) process source code structural patterns by extracting functions, segmenting them into slices, and transforming these slices to the FNV-1a hash signature. It utilizes a binary bit-vector for function comparison, identifying clones by matching these vectors. However, lacking flexibility leads to potential false positives and negatives in large databases. This inflexibility may result in mismatches due to code variations in Type-III (ST and MT), adversely affecting the technique's precision, recall, f1-score, and accuracy.

The DroidMD (Akram et al., 2021) employs a unique preprocessing approach, labeling code identifiers with generic "id" tags and incrementally numbering each identifier, excluding Java keywords. Streamlining identifiers risks losing contextual information crucial for precise clone detection. Strict signature chunk matching and large chunk size limits its ability to

recognize clones with minor changes, consequently impacting its effectiveness. Although DroidMD demonstrates its effectiveness in identifying simpler Type-I clones, its performance markedly declines for more complex Type-III (ST and MT). This decline is attributed to its strict chunk size requirements and inadequacy to accommodate code variations, reducing its overall detection efficacy. Simultaneously, Yang et al. (2018) transform functions into abstract syntax trees with several node types, including class elements, code blocks, statements, expressions, operators, keywords, and literals. These nodes are converted into a sequence of characters, providing a compact code representation stored in a database for unique function patterns. However, due to the large number of functions in extensive codebases and the limited characters, the local sequence alignment algorithm can cause a high ratio of false positives and negatives, affecting accurate clone identification in large databases. This challenge leads to the average performance of Yang et al. in the clone detection testing environment.

CodeGuard stands out in comparative studies with unmatched precision, recall, F1-score, accuracy, and efficiency across clone types. Beyond mere detection, it also pinpoints changes made in clone code, providing deep insights for analysis. This emphasizes *CodeGuard*'s critical role in enhancing software quality and maintenance efficiency. Offering in-depth analytical insights empowers developers with a profound understanding of code quality, enabling the development of more robust and maintainable software systems.

System specification

The CloneVault and clone detection were performed on a Windows 11 (64-bit) system with an Intel i5 13500H v5 CPU @2.6GHz, 16GB LPDDR5 RAM. *CodeGuard* and comparative techniques were evaluated using batch processing within this setup.

8 Conclusion

This research presents *CodeGuard*, an advanced code cloning technique for identifying the entire range of code clones, including Type-I, Type-II, and Type-III (ST and MT). It's comprehensive preprocessing significantly enhances the detection of direct clones, especially for Type-II clones, while innovative indexing and matching techniques enhance the general clones, including all Type-III sub-types clones . It surpasses existing methods; *CodeGuard* achieves unparalleled average precision (0.98), recall (0.96), F1-score (0.96), and accuracy (0.96) across all clone types, achieving a significant advancement in clone detection. Its exceptional ability to accurately identify complex Type-III MT clones, achieving both F1-scores and accuracy of 0.95, sets a new performance standard. In evaluating average efficiency through ten iterative computations, *CodeGuard* outperforms comparative techniques. Beyond simple detection, *CodeGuard* precisely pinpoints modifications made within clone pairs, significantly improving software maintenance by enhancing code quality and facilitating maintenance processes. Comprehensive evaluation results using a diverse dataset confirm *CodeGuard* as an essential technique for detecting the complex challenges of clone detection in modern software engineering. This research study highlights *CodeGuard*'s role as a benchmark in clone detection, emphasizing its significance and importance to software quality and maintainability.

Data availability statement

Publicly available datasets were analyzed in this study. This data can be found here: https://figshare.com/s/95c248d22beab531f1ec.

Author contributions

YG: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. LP: Funding acquisition, Supervision, Writing – review & editing.

Funding

The author(s) declare financial support was received for the research, authorship, and/or publication of this article. This research was conducted under the supervision of Prof. Luo Ping and was financially supported by the Key Research Program of the Chinese Ministry of Science and Technology, grant number 2022YFB3103903.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

References

Akram, J., and Luo, P. (2021). Sqvdt: A scalable quantitative vulnerability detection technique for source code security assessment. *Software* 51, 294–318. doi: 10.1002/spe.2905

Akram, J., Mumtaz, M., Jabeen, G., and Luo, P. (2021). Droidmd: an efficient and scalable android malware detection approach at source code level. *Int. J. Inf. Comput. Secur*. 15, 299–321. doi: 10.1504/IJICS.2021.116310

Akram, J., Mumtaz, M., and Luo, P. (2020). Ibfet: Index-based features extraction technique for scalable code clone detection at file level granularity. *Software* 50, 22–46. doi: 10.1002/spe.2759

Akram, J., Qi, L., and Luo, P. (2019). "Vcipr: vulnerable code is identifiable when a patch is released (hacker's perspective)," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)* (IEEE), 402–413. doi: 10.1109/ICST.2019.00049

Allamanis, M., and Sutton, C. (2013). "Mining source code repositories at massive scale using language modeling," in *The 10th Working Conference on Mining Software Repositories* (IEEE), 207–216. doi: 10.1109/MSR.2013.6624029

Chau, N.-T., and Jung, S. (2020). Enhancing notation-based code cloning method with an external-based identifier model. *IEEE Access* 8, 162989–162998. doi: 10.1109/ACCESS.2020.3016943

Chen, J., Alalfi, M. H., Dean, T. R., and Zou, Y. (2015). Detecting android malware using clone detection. *J. Comput. Sci. Technol*. 30, 942–956. doi: 10.1007/s11390-015-1573-7

Chodarev, S., Pietrikova, E., and Kollar, J. (2015). "Haskell clone detection using pattern comparing algorithm," in *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)* (IEEE), 1–4. doi: 10.1109/EMES.2015.7158423

Ducasse, S., Rieger, M., and Demeyer, S. (1999). "A language independent approach for detecting duplicated code," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)* (IEEE), 109–118. doi: 10.1109/ICSM.1999.792593

Giani, Y., Ping, L., and Shah, S. A. (2022). "Ayat: a lightweight and efficient code clone detection technique," in *2022 3rd Asia Conference on Computers and Communications (ACCC)* (IEEE), 47–52. doi: 10.1109/ACCC58361.2022.00015

Glani, Y., Ping, L., Lin, K., and Shah, S. A. (2023). "Ayatdroid: a lightweight code cloning technique using different static features," in *2023 IEEE 3rd International Conference on Software Engineering and Artificial Intelligence (SEAI)* (IEEE), 17–21. doi: 10.1109/SEAI59139.2023.10217577

Glani, Y., Ping, L., and Shah, S. A. (2022). "Aash: a lightweight and efficient static iot malware detection technique at source code level," in *2022 3rd Asia Conference on Computers and Communications (ACCC)* (IEEE), 19–23. doi: 10.1109/ACCC58361.2022.00010

Higo, Y., Ueda, Y., Kamiya, T., Kusumoto, S., and Inoue, K. (2002). "On software maintenance process improvement based on code clone analysis," in *Product Focused Software Process Improvement: 4th International Conference, PROFES 2002 Rovaniemi, Finland* (Springer), 185–197. doi: 10.1007/3-540-36209-6_17

Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. (2009). "Do code clones matter?" in *2009 IEEE 31st International Conference on Software Engineering* (IEEE), 485–495. doi: 10.1109/ICSE.2009.5070547

Kapser, C. J., and Godfrey, M. W. (2006). Supporting the analysis of clones in software systems. *J. Softw. Mainten. Evol*. 18, 61–82. doi: 10.1002/smr.327

Kim, S., Woo, S., Lee, H., and Oh, H. (2017). "Vuddy: a scalable approach for vulnerable code clone discovery," in *2017 IEEE symposium on security and privacy (SP)* (IEEE), 595–614. doi: 10.1109/SP.2017.62

Li, D., Piao, M., Shon, H. S., Ryu, K. H., and Paik, I. (2014). "One pass preprocessing for token-based source code clone detection," in *2014 IEEE 6th International Conference on Awareness Science and Technology (iCAST)* (IEEE), 1–6. doi: 10.1109/ICAwST.2014.6981824

Lyu, F., Lin, Y., Yang, J., and Zhou, J. (2016). "Suidroid: an efficient hardening-resilient approach to android app clone detection," in *2016 IEEE Trustcom/BigDataSE/ISPA* (IEEE), 511–518. doi: 10.1109/TrustCom.2016.0104

Misu, M. R. H., and Sakib, K. (2017). "Interface driven code clone detection," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)* (IEEE), 747–748. doi: 10.1109/APSEC.2017.97

Myers, E. W. (1986). An o (nd) difference algorithm and its variations. *Algorithmica* 1, 251–266. doi: 10.1007/BF01840446

Nakamura, Y., Choi, E., Yoshida, N., Haruna, S., and Inoue, K. (2016). "Towards detection and analysis of interlanguage clones for multilingual web applications," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (IEEE Computer Society), 17–18. doi: 10.1109/SANER.2016.55

Pati, J., Kumar, B., Manjhi, D., and Shukla, K. K. (2017). A comparison among arima, bp-nn, and moga-nn for software clone evolution prediction. *IEEE Access* 5, 11841–11851. doi: 10.1109/ACCESS.2017.2707539

Ragkhitwetsagul, C., and Krinke, J. (2017). "Using compilation/decompilation to enhance clone detection," in *2017 IEEE 11th International Workshop on Software Clones (IWSC)* (IEEE), 1–7. doi: 10.1109/IWSC.2017.7880502

Saini, N., Singh, S., et al. (2018). Code clones: detection and management. *Procedia Comput. Sci.* 132, 718–727. doi: 10.1016/j.procs.2018.05.080

Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V. (2016). "Sourcerercc: scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, 1157–1168. doi: 10.1145/2884781.2884877

Singh, G. (2017). "To enhance the code clone detection algorithm by using hybrid approach for detection of code clones," in *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)* (IEEE), 192–198. doi: 10.1109/ICCONS.2017.8250708

Song, X., Yu, A., Yu, H., Liu, S., Bai, X., Cai, L., et al. (2020). "Program slice based vulnerable code clone detection," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)* (IEEE), 293–300. doi: 10.1109/TrustCom50675.2020.00049

Startin, R. (2019). *XXHash, designed by Yann Collet*. Richard Startin's Blog. Retrieved from: https://richardstartin.github.io/posts/xxhash#::text=XXHash%20is%20a%20fast%20the,than%20it%20can%20be%20copied (accessed June 1, 2024).

Svajlenko, J., and Roy, C. K. (2015). "Evaluating clone detection tools with bigclonebench," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (IEEE), 131–140. doi: 10.1109/ICSM.2015.7332459

Uemura, K., Mori, A., Fujiwara, K., Choi, E., and Iida, H. (2017). "Detecting and analyzing code clones in hdl," in *2017 IEEE 11th International Workshop on Software Clones (IWSC)* (IEEE), 1–7. doi: 10.1109/IWSC.2017.7880501

Wang, M., Wang, P., and Xu, Y. (2017). "Ccsharp: an efficient three-phase code clone detector using modified pdgs," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)* (IEEE), 100–109. doi: 10.1109/APSEC.2017.16

Wang, P., Svajlenko, J., Wu, Y., Xu, Y., and Roy, C. K. (2018). "Ccaligner: a token-based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering*, 1066–1077. doi: 10.1145/3180155.3180179

Yang, Y., Ren, Z., Chen, X., and Jiang, H. (2018). "Structural function based code clone detection using a new hybrid technique," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)* (IEEE), 286–291. doi: 10.1109/COMPSAC.2018.00045

Yu, D., Wang, J., Wu, Q., Yang, J., Wang, J., Yang, W., et al. (2017). "Detecting java code clones with multi-granularities based on bytecode," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)* (IEEE), 317–326. doi: 10.1109/COMPSAC.2017.104

Yuki, Y., Higo, Y., and Kusumoto, S. (2017). "A technique to detect multi-grained code clones," in *2017 IEEE 11th International Workshop on Software Clones (IWSC)* (IEEE), 1–7. doi: 10.1109/IWSC.2017.7880510

Zakeri-Nasrabadi, M., Parsa, S., Ramezani, M., Roy, C., and Ekhtiarzadeh, M. (2023). A systematic literature review on source code similarity measurement and clone detection: techniques, applications, and challenges. *J. Syst. Softw*. 204:111796. doi: 10.1016/j.jss.2023.111796