Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Finding Open Source Code In Obfuscated Android Applications

Author

Keira Marie Gatt

Supervisor

Prof. Douglas Leith

April 2024

A Dissertation submitted in partial fulfilment

of the requirements for the degree of

Master in Computer Science (MCS)

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

_____

Keira Marie Gatt

15.04.2024

Abstract

In the domain of software engineering, detecting the use of open source code in obfuscated Android applications holds significant importance across various disciplines. The ability to identify syntactically or semantically similar code fragments, commonly referred to as clones, is essential in tasks such as code versioning, plagiarism detection and ethical considerations surrounding open source code usage.

This dissertation presents the concepts, artefacts and processes involved in the development of an APK Code Matching (ACM) application, designed to identify and match similar methods between open source and obfuscated variants of Java Android applications. The ACM employs a three-way approach, leveraging static code analysis, program call hierarchy, and the k-Nearest Neighbours (k-NN) algorithm for match detection of class methods.

The extensive evaluation carried out as part of the development process underscores the effectiveness and efficiency of the ACM application in terms of its ability to accurately map methods between APKs, even in scenarios involving complex and dense obfuscation.

Acknowledgements

I would like to express my sincere gratitude to Professor Douglas Leith for his invaluable guidance throughout the course of this dissertation. His expertise and encouragement have been instrumental in navigating this project.

I am also profoundly grateful to my parents for their unwavering support and encouragement. Their belief in my abilities has been a constant source of motivation and strength.

Contents

1 Introduction 9

1.1 Motivation 9

1.2 Research Objectives 10

1.3 Dissertation Overview 10

2 Background 11

2.1 Obfuscation & Code similarity 11

2.2 Code Similarity Measures 13

Static Code Analysis 13

Dynamic Code Analysis 19

2.3 Tools 20

2.3.1 Android APK 20

2.3.2 JADX 20

2.3.3 F-Droid 20

2.3.4 Androguard 21

2.4 k-Nearest Neighbours 21

3 Design 23

3.1 Preliminary Stage 24

3.1.1 Overview 24

3.1.2 Environment Setup 24

3.1.3 Obfuscation Removal 24

3.2 Stage One 26

3.2.1 Code Filtering 26

3.2.2 Unique Composite Strings 27

3.2.3 Code Signatures 28

3.2.4 Feature Selection 30

3.2.5 List of Callees 31

3.3 Stage Two 33

3.3.1 String Matching 33

3.3.2 k-NN Analysis 33

3.3.3 Filtering 37

3.4 Stage Three 39

3.4.1 Ablation Study 39

3.4.2 Implementation 40
4 Evaluation 41
4.1 Setup 41
4.2 Evaluation Metrics 42
4.3 Determining the Ground Truth 43
4.4 Accuracy & Coverage 43
4.4.1 Precision 44
4.4.2 Recall 45
4.4.3 F-Measure 46
4.5 Runtime Efficiency 47
4.6 Additional Testing 48
4.6.1 Test Case 1 48
4.6.2 Test Case 2 48
5 Conclusion 49
5.1 Limitations 49
5.1.1 Scope Limitations 49
5.1.2 Design Limitations 49
5.2 Future Work 50
5.2.1 Support for Multiple Programming Languages 50
5.2.2 Comparison of Multiple APKs 50
5.2.3 Expand Input File Support 50
5.2.4 Improvement of User Interface 50
5.3 Closing Remarks 51
Bibliography 52

List of Figures

Figure 1: Examples of Clone Types 12
Figure 2: k-NN in two dimensional space 22
Figure 3: Process Flow Architecture Diagram 23
Figure 4: MultiLabelBinarizer 29
Figure 5: k Neighbours Selection 34
Figure 6: Example of k-NN Analysis I Error 44

Abbreviations

ACM	APK Code Matching
APK	Android Package
FN	False Negative
FP	False Positive
k-NN	k Nearest Neighbours
TP	True Positive

1 Introduction

In a digital age characterised by the proliferation of smartphone applications, analysing the integrity and security posture of these digital assets has become increasingly important. The goal of this dissertation is to present an effective method for comparing two Java-based Android applications, packaged in the standard Android Package (APK) format. Specifically, the aim is to analyse the similarities between two versions of APKs - one that includes obfuscated code and the other containing the target open source code. By parsing the decompiled bytecode included in these apps, the project attempts to identify and match class methods between the two APKs, providing important insights to those involved in software development, cyber security, and ethical oversight.

1.1 Motivation

The project is motivated by several compelling factors. Firstly, in the fast-paced realm of software development, where new iterations of applications are released frequently, the ability to swiftly and accurately compare different versions of an application is invaluable. For software maintenance and version control, identifying bugs that persist across multiple versions of an application is of significant importance; enabling developers to quickly resolve duplicated bugs. This dissertation aims to provide a tool to aid in the efficient and accurate comparison between different versions of an application by identifying similarities between the codebases.

Furthermore, the work carried out in the dissertation holds significant implications in the domain of corporate due diligence, facilitating thorough code reviews during mergers and acquisitions. By comparing the acquired codebase to open source code projects, companies can mitigate the risk of unintended commercial use of open source code without the proper permissions.

Additionally, the project addresses concerns regarding plagiarism in machine learning-generated code. By detecting similarities between code produced by machine learning models, such as GitHub's and OpenAI's Copilot [1] and existing open source repositories, the project aims to enhance the integrity of AI-generated code and highlight instances of inadvertent plagiarism from open source repositories.

Moreover, this work aims to identify instances of intentional code copying within software projects, even in cases of deliberate attempts to conceal such practices. Given two applications as input, the project can detect code snippets that are identical or similar to each other, known as code clones [2], thereby contributing to intellectual property protection.

Furthermore, as some developers incorporate malicious code into software packages, the project can be used to detect known malicious code snippets within software applications, contributing to cyber security efforts and aiding in the mitigation of malware threats.

With regards to the last two fields of application, it is important to acknowledge the limitations of the project. The techniques employed to conceal code plagiarism and malware within applications are often adversarial and can easily bypass the methods used in the project approach.

1.2 Research Objectives

In order to attain the goals of this dissertation, that is to map similar class methods between APKs, it was necessary to define the following project objectives:

(i) Acquire a set of APKs for development and testing purposes

(ii) Design a methodology to match open source code to its obfuscated counterpart

(iii) Determine suitable criteria to evaluate the matching confidence

(iv) Evaluate the performance of the resultant matches

1.3 Dissertation Overview

The structure of the dissertation is as follows:

Section 2 provides an overview of obfuscation and its various types, a literature review of prior research in the field of code similarity, a description of the various tools and environments utilised, and background information on the k-Nearest Neighbours algorithm.

Section 3 focuses on the design and implementation of the ACM application.

Section 4 presents the findings of the several tests carried out to assess the application's performance.

Section 5 discusses the limitations of the ACM application and the potential avenues for future work.

2 Background

2.1 Obfuscation & Code similarity

Obfuscation is a technique commonly employed to enhance software security, protect intellectual property and is an inherent aspect of code release management across different iterations of a codebase. However, obfuscation poses a challenge to code similarity analysis. So while two code segments, in this particular case two Java-based class methods, might be functionally identical, they might be dissimilar in their appearance due to obfuscation. The degree of variation between the two can differ, with code segments becoming increasingly more obfuscated when more aggressive techniques are employed. These levels of obfuscation can be classified on how closely they preserve the original code, essentially categorising them based on how similar the obfuscated code is to the original.

In literature related to code similarity, two code fragments form a code pair if they are sufficiently similar, according to a given definition of similarity [3]. Therefore, the concept of clone pairs can be leveraged to establish a framework for categorising different levels of obfuscation based on how closely the obfuscated code resembles the original. This approach draws upon widely-accepted definitions of clone types outlined in existing research [3, 4, 5].

While there are several types of clones, the following table lists only those types that are specific to this project:

Table 1: Clone types

Type	Description
I	Code remains largely identical, except for changes in whitespaces and comments. Such scenarios often arise when developers selectively choose not to obfuscate certain classes or methods.
II	Similar to Type I, differences in whitespaces and comments may occur, but there are also changes in identifiers. This form of obfuscation, often employed for code optimisation purposes, involves renaming classes, methods, and fields to shorter, meaningless names. The dual purpose of this is to conserve space by utilising shorter names and to safeguard intellectual property by complicating the comprehension of the code.
III	This type involves more substantial changes, including modifications to code structure. Portions of code may be deleted, updated, or supplemented with new segments. Such modifications are central to code shrinking, a process aimed at identifying and removing unused code segments to reduce application size and enhance performance. This can also be caused by method inlining, which involves the replacement of frequently called method invocations with their bodies, enhancing application speed by eliminating the overhead associated with function calls.

In addition, the following diagram illustrates examples of these clone types, written in Java.

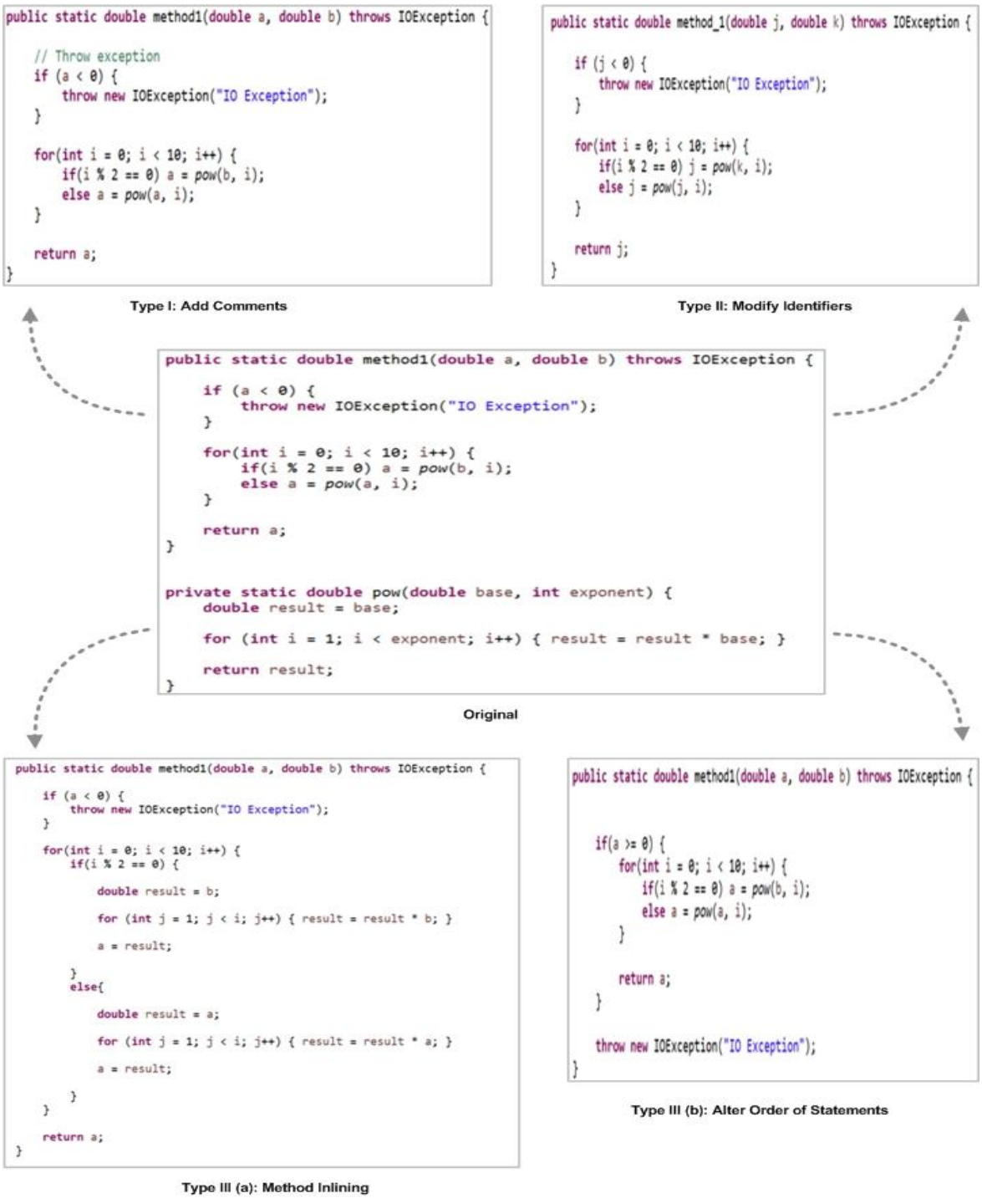

Figure 1: Examples of Clone Types

2.2 Code Similarity Measures

This section explores the methodologies employed in previous research for measuring the similarity between code snippets. The techniques used in the field relating to measuring code similarity can be broadly categorised as static code analysis and dynamic code analysis. Since this project deals only with static code analysis, much of the ensuing discussion will focus on this particular analysis technique as a field of research.

Static Code Analysis

Methods based on static code analysis do not rely on the execution of the programs to measure similarity. Rather, they are data-centric, relying on information extracted from parsing the code and finding patterns within the source code. The many techniques that have been proposed in literature to identify similarities between code fragments will be discussed next.

2.2.1 Token-based

In token-based algorithms, code is processed into a sequence of tokens and then these sequences are compared to find common subsequences. Token-based approaches are widely used and are also very efficient when scaling to large source line counts. Still, as they capture only an abstracted representation, they do not consider the order of the code, and ignore structural information [6].

JPlag

JPlag, developed by Prechelt et al. [7], stands out as one of the most popular and well-known code plagiarism detection tools, boasting a powerful user interface. JPlag tokenises code programmes into a sequence of token strings and employs the Greedy String Tiling algorithm [8] to identify the largest set of contiguous substrings.

The tool has shown promising results in detecting plagiarisms among student submissions of Java exercises. In tests involving small sets of student submissions, typically around 28 programs, JPlag achieved remarkable precision and recall rates of 100%. However, its efficacy on large code repositories or applications remains uncertain, as it has primarily been tested on coding exercises from informatics courses and modules introducing Java and AWT to experienced students. While support for C, C++ and Scheme is also available, its performance across programming languages other than Java has not yet been evaluated.

CP-Miner

CP-Miner [9], another token-based tool, was developed to identify copy-and-paste bugs in large software suites. CP-Miner is based on frequent subsequence mining, which is an association analysis technique to discover frequent subsequences in a sequence database [10]. The tool employs CloSpan (Closed Sequetial Pattern Mining), a proposed frequent subsequence mining algorithm, which uses depth-first search and pruning to obtain a set of subsequences [11].

Despite its success in detecting bugs in large codebases, it has been reported to yield a high rate of false positives in some cases. For instance, when identifying copy-and-paste code segments in the Free-BSD operating system [12], 85% of the 443 bugs reported by CP-Miner were false positives. The authors suggest that to reduce the number of false positives, it is necessary to extract of more

semantic information from the source code. The prevalence of false positives could pose a significant challenge if CP-Miner were to be deployed in a production environment. Not only could it impede productivity, but it could also lead to frustration for engineers who must sift through numerous reported bugs, only to discover that the majority are false alarms.

Deimos

Deimos [13], a source code plagiarism detector with main purpose of detecting plagiarism in academic settings, operates in two steps:

(a) Parsing source code and transforming it into tokens

(b) Comparing each pair of token strings using Running Karp-Rabin algorithm [14]

As Deimos is tailored for detecting similarity in coding assignments and provides users with a similarity value distribution chart and a matrix of pairwise similarity scores, its application beyond academic purposes, such as detecting plagiarism in large codebases or applications, is currently limited.

2.2.2 Text-based

Text-based algorithms are one of the simplest methods of code similarity measure, where source code is treated as plain text in order to find the longest common sequence of strings.

CDSW

Hiroaki Murakami et al. [15] employ the Smith-Waterman algorithm within their tool called CDSW to identify code clones in large code repositories. By treating text as sequence of strings and using the Smith-Waterman algorithm, the tool aims to identify similar code. The Smith-Waterman algorithm is an approximate string matching technique for identifying similar alignments between two sequences even if they contain gaps.

The evaluation was conducted on freely available code clone data, which comprised large code repositories of eight software systems, including NetBeans and PostgreSQL. Despite being an efficient algorithm, with execution time for detecting code clones across all target software systems taking a maximum of 30 seconds, the median precision and recall reveal shortcomings. A median precision of 0.2 indicates that a significant proportion of clones detected where in fact false positives, undermining the algorithm's ability to accurately identify code clones. Moreover, the median recall of 0.46, suggests that the tool misses a considerable amount of clones within the repositories. These limitations underscore the need for further refinement of the CDSW tool before its deployment in production environments.

PlaGate

PlaGate, a tool developed by Georgina Cosma and Mike Joy [16], integrates with existing plagiarism detection tools to enhance performance. PlaGate employs an information retrieval approach called Latent Semantic Analysis [17] on pre-processed source code files, represented as a numeric matrix. Similar files are identified through pairwise comparisons using cosine similarity.

When combined with tools such as JPlag, an increase recall and decrease in precision were observed compared to using JPlag alone. In one of the datasets tested, when JPlag was used alone, a recall of 0.5 and precision of 0.75 was achieved. However, when integrated with PlaGate, recall increased to 0.67 and precision dropped to 0.67. The rise in recall is attributed to the fact that PlaGate and external tools complement each other by detecting different kinds of plagiarism attacks. However, the decrease in precision indicates that PlaGate may introduce more false positives.

Therefore, the decision to integrate PlaGate would depend on the specific requirements of the use-case. In situations where the primary goal is to identify as many clones as possible, and false positives are tolerable, PlaGate proves beneficial. However, in contexts where precision is important to minimise false alarms, the integration of PlaGate may not be the best option.

2.2.3 Tree-based

Utilising tree-based algorithms for static code analysis necessitates the conversion of the source code into either a parse tree or an Abstract Syntax Tree (AST), using a programming language parser.

A parse tree is a graphical representation of the derivation or parse that corresponds to the input program. It represents the complete derivation with a node for each grammar symbol in the derivation. In contrast, the AST is a contraction of the parse tree that omits most nodes for non-terminal symbols while preserving the fundamental structure [18].

By comparing subtrees within the generated parse tree or AST, or employing various metrics to fingerprint them, exact or similar code segments can be identified.

FAMIX Tree Representations

Sager et al. [19], present an approach to discern similarities between Java classes by leveraging ASTs. Firstly, they obtain the AST representation of each class using Eclipse's ASTParser [20]. Subsequently, the AST is converted into an intermediate model known as FAMIX (FAMOOS Information Exchange Model), a programming language-independent model for representing object-oriented source code [21]. Subsequently, by comparing the FAMIX tree representations of two classes using tree comparison algorithms, the similarity between two classes is discerned.

Their study evaluated three distinct methodologies for calculating similarity between trees:

(a) Bottom-up maximum common subtree isomorphism

(b) Top-down maximum common subtree isomorphism

(c) Tree edit distance

To assess the effectiveness of the proposed approach and the implemented similarity measures, the authors selected the widely-used Java plug-in, *org.eclipse.compare*, as their dataset. They measured the similarity of the classes within the same version as well as between different versions of the plug-in using each of the three similarity measures.

The findings indicated that the tree edit distance produced the best results. It successfully identified structural similarities between classes within the same version and different versions of the project. However, it is important to note that the approach is only capable of matching classes, not the

methods in classes. This leaves the comparison of methods within the classes to the user. Additionally, it is worth mentioning that the algorithm employing the best performing similarity measure, tree edit distance, required over an hour to analyse the 114 classes of the *org.eclipse.compare* plug-in, indicating considerable processing time.

DECKARD

Jiang et al. [22] have developed a clone detection tool called DECKARD using a novel tree similarity algorithm. The main idea of the algorithm is to capture structural information of ASTs as vectors and utilise an adaptation of Locality Sensitive Hashing [23] to cluster similar vectors. This clustering process effectively groups together similar code fragments, thereby identifying code clones.

DECKARD's effectiveness has been evaluated on large code bases written in Java and C, including JDK and the Linux Kernel. The evaluation encompassed various metrics such as the number of detected clones and the number of false positives. When evaluated on the JDK, DECKARD's performance was compared against CloneDR [24], a well-known AST-based clone detection tool for Java. Similarly, in the evaluation on the Linux Kernel, DECKARD's performance was compared against CP-Miner [9], a token-based tool for the C programming language. Evaluation revealed that DECKARD significantly outperformed both CloneDR and CP-Miner.

However, the performance of DECKARD is highly sensitive to the threshold used to determine similarity. Thus, parameter tuning is essential from the user's perspective to achieve optimal results.

2.2.4 Graph-based

Graph-based methods typically utilise the Program Dependence Graph (PDG) to analyse code fragments and identify similarities [25]. The PDG contains both data and control dependencies between program statements, and hence conveys both the semantics of the program in addition to syntax. Program statements are represented as nodes, while data and control dependencies are depicted as edges.

PDG Subgraphs

Krinke [26] has developed an approach to detect semantically similar code fragments within a program by using specialised PDGs. To detect duplicated code, similar subgraphs within the PDGs are identified. The detected similar subgraphs can then be directly mapped back onto the program code and presented to the user.

In the implementation, a specialised version of PDGs is employed. Krinke uses specialised edges for control and data dependencies, along with special nodes for variable and procedure definitions. This augmentation enhances the verbosity of the graphs, aiding the identification of similar or identical nodes and edges.

However, the time complexity of the proposed algorithm is $O(|N|^2)$, where N denotes the number of PDG nodes. Therefore, for code fragments with a large PDG, the algorithm is inefficient. Evaluation revealed prolonged execution times, with some runs taking over an hour to complete. For instance, in the case of the *agrep* project, the execution time exceeded 10 hours.

2.2.5 Metric-based

Metric-based methods represent code segments as vectors of various code metrics. This allows the use of vector similarity to assess the degree of similarity between code segments based on their metric values. By measuring the similarity using these metrics, metric-based methods effectively identify code segments with similar characteristics.

Metric-Axes Clustering

Kontogiannis et al. [27] proposed a metric-based algorithm for detecting instances of code cloning in moderately-sized production systems. To facilitate the analysis and numerical comparison of code segments, the authors utilised modifications of five metrics, specifically chosen for their low correlation, as determined by the Spearman-Pierson correlation test. These metrics were employed to characterise and classify code segments.

For a given code segment s, the following metrics were used for the analysis:

(a) $S\_COMPLEXITY(s) = FAN\_OUT(s)^2$, where $FAN\_OUT(s)$ is the number of individual function calls made within s.

(b) $D\_COMPLEXITY(s) = GLOBALS(s) \,/\, (FAN\_OUT(s) + 1)$,where $GLOBALS(s)$ is the number of global variable used within s.

(c) $MCCABE(s) = e - n + 2$,where e is the number of edges in the control flow graph, n denotes the number of nodes in the graph.

Essentially, it is a quantitative measure of independent paths in the source code. Alternatively, the McCabe metric can be calculated as $1 + d$,where d is the number of control decision predicates in the code segment s.

(d) $ALBRECHT(s) = p_1 * v + p_2 * g + p_3 * u + p_4 * f$, where v represents the number of data elements set and used, g denotes the number of global data elements set, u is the number of read operations, f stands for the number of files accessed for reading. The $p_1, \dots, p_4$ values are weight factors, with $p_1 = 5$, $p_2 = 4$, $p_3 = 4$ and, $p_4 = 7$

(e) $KAFURA(s) = \left(KAFURA\_IN(s) * KAFURA\_OUT(s)\right)^2$, where $KAFURA\_IN(s)$ is the sum of local and global incoming dataflow, and $KAFURA\_OUT(s)$ represents the sum of local and global outgoing dataflow

Once the metrics are computed for each code segment, the comparison process begins. The technique starts by creating clusters of potential clones for every metric axis $M_i (i = 1..5)$ based on a user-defined Euclidean distance threshold, d_i. Intersections of clusters across different axes yield intermediate results, and once all metric axes have been considered, the final clusters contain potential code clone fragments.

The metric-based clone detection analysis was applied to several C programs, including *tsch*, *bash*, and *CLIPS*. For the CLIPS system, consisting of 30,000 lines of code, and with approximately 500,000 pairs of functions for clone candidate selection, the clone detection system exhibited exceptional speed, taking less than 90 seconds to run.

However, the study revealed varying levels of false positives across different programs. Notably, Euclidean distance thresholds close to 0.0 yielding more accurate results. On average, across the three systems, 39% of the reported matches were false positives. The authors observed that integrating dynamic code analysis into the detection system significantly reduced the false positive rate to 10%, albeit at the expense of increased runtime, which increased to 3.9 minutes. Additionally, it is suggested that addressing this issue may involve the incorporation of new metrics such as Halstead's metric.

However, even 10% false positives rate remains an issue, particularly given the large size of the systems under analysis. The manual effort required to filter through these false matches could be substantial, potentially outweighing the benefits of the approach's speed.

2.2.6 Machine Learning-based

Machine learning-based algorithms have emerged as powerful tools in the domain of code similarity detection. These approaches leverage patterns and characteristics inherent in source code, learning from labelled datasets of known similar and dissimilar codes. From traditional classifiers to sophisticated deep learning architectures, a diverse array of learning-based approaches are employed to effectively measure code similarities and identify code clones.

Clonewise

Cesare et al. [28] utilised conventional classification models to detect package clones. The authors define a package clone as the duplication of one package's code within another package. Their system, Clonewise, serves as a tool for security teams to conduct audits and maintain operating system distributions. It takes two packages as input and extracts features from them based on filenames and file content. Among the 26 features used are:

(a) Number of filenames

(b) Number of common filenames between the packages

(c) Number of similar filenames between the packages (calculated using a fuzzy string similarity function based on the filenames' edit distance)

(d) Number of files with identical content, identified by hashing the file content to identify matching files

The authors experimented with various classifiers, including Naïve Bayes [29], Multilayer Perceptron, C4.5 [30], and Random Forest [31]. Evaluation was conducted on Fedora and Debian Linux distributions, focusing on identifying shared code among packages such as lib3ds, PostgreSQL, and libwmf.

The Random Forest classification algorithm yielded the best results, achieving a true positive rate of 70.04% and a false positive rate of 0.11%. In an attempt to reduce false positives, the decision threshold of the Random Forest algorithm was adjusted to 0.8, prioritising false positives over false negatives. This adjustment lowered the false positive rate to 0.03%, but also decreased the true positive rate to 58.61%.

Given that the purpose of Clonewise is to conduct security audits, the relatively low true positive rate raises concerns about its effectiveness in accurately identifying potential threats. Having to sift through a significant number of false alarms (41.39%) can consume valuable time and resources for security teams. Therefore, further work may be required to improve this.

Despite its limitations, Clonewise's package-level approach provides security teams with a convenient way of identifying potential security threats. However, if this approach were to be applied to other applications like plagiarism detection, a more granular identification of duplicated code segments, such as identifying exact replicated methods across packages, may be necessary.

Siamese Neural Network

Patel and Sinha [32] used a Siamese Neural Network (SNN) [33] to address code clone detection. Firstly, they extracted ASTs and CFGs for each method in the source code. To use these as input to the network, these structures were converted into vectors using Word2vec with skip-o-gram [34] and Graph2vec [35] techniques, respectively. The resulting vectors were combined to generate an embedding to be fed into the network.

The chosen architecture was an intermediate-merge SNN, where two embeddings are fed separately into identical arms of the SNN. Each arm of the SNN shares the same architectures and weights, and consists of two linked networks: ConvNet and Dense. Each arm computes the features of one input independently. Subsequently, the similarity between the computed features is calculated using their difference, which is then passed to the final output layer. The output layer performs a binary classification, where an output of 1 denotes clones and 0 denotes non-clones. Because of the binary nature of the classification, cross-entropy loss was used for training. Upon comparing Leaky-ReLU, ReLU and Tanh activation functions, the authors found that ReLU produced the best results.

The solution was evaluated on the OJClone dataset, containing C++ solutions to programming problems from various students. The set of answers to a problem were considered clones of each other as they were expected to exhibit the same logic. The model performed well on this dataset, demonstrating high performance with precision, recall, and F1-score values of 0.902, 0.933, and 0.917 respectively. However, evaluation was limited to 15 problem sets, each containing 100 C++ source code files, without extension to other programming languages or real-world applications.

Dynamic Code Analysis

In contrast to static code analysis, dynamic code analysis involves examining data collected from a process, i.e. a running program. While static code analysis focuses on deriving properties from a program’s text and is effective at detecting lexically similar codes; dynamic code analysis is better suited for detecting semantic code clones, which are similar in functionality but different in implementation.

This type of analysis typically involves analysing a system’s execution under test [36], requiring an execution environment like the Java Virtual Machine (JVM) or a more sophisticated setup such as a program analysis framework like JNuke [37]. Through this analysis, a comprehensive understanding of a software system's behaviour is gained, revealing its actual execution paths and interactions with its environment. The data collected from these executions serves as basis for evaluating code similarity, offering insights that range from detailed class-level information to broader, high-level architectural views [38].

2.3 Tools

2.3.1 Android APK

The Android Package (APK), also known as Android Package Kit or Android Application Package, serves as the standard file format for distributing and installing apps on Android devices. When an Android developer is ready to distribute their app, they compile the code to Dalvik bytecode. The resulting *.dex* files, along with other resources, are bundled together into a single APK container that is organised as follows:

classes.dex	The Dalvik Executable file containing the code executed by Android Runtime.
lib/.	Folder that contains platform-dependent compiled code and native libraries for device-specific architectures, such as x86 or x86_64.
resources.arsc	Pre-compiled resources used by the application, such as Binary XML files.
res/.	Folder containing resources such as images, that were not compiled into the resources.arsc.
AndroidManifest.xml	Information about the application, including its name, version number and permission rights.
META-INF/.	Folder containing resources like the manifest file, app certificate and list of all resources within the APK.
assets/.	Contains assets and resources for the application, such as code and data files.

2.3.2 JADX

JADX [39] is a decompiler for Java code that provides a user-friendly graphical interface for decompiling Dalvik bytecode to Java classes from various file formats including APK, DEX, AAR, AAB and ZIP files. It includes a powerful search tool to locate code segments, method names and classes across the entire code base. Additionally, JADX offers syntax highlighting and tabbed views, along with a '*jump to declaration*' feature for easy code navigation.

2.3.3 F-Droid

F-Droid [40] is an app store specialising in free and open source software for Android devices, with a strong emphasis on promoting privacy, security and transparency for end-users. It fosters a culture of user freedom by allowing individuals to inspect and modify the code of the apps they use, empowering them to tailor their experience to their preferences.

The platform provides a diverse range of applications, covering categories such as security utilities, games, and messaging tools. Similar to other mainstream app stores, F-Droid allows users to easily install, update and review applications. Notably, as the applications featured on F-Droid are open source, their source code is publicly available on code repository platforms such as GitHub.

2.3.4 Androguard

Androguard [41] is a powerful Python tool for reverse engineering Android applications. It allows users to decompile APK and DEX files into their original source code. With its extensive APIs, Androguard enables thorough analysis of source code, allowing users to examine the logic, flow and implementation details of applications. Additionally, it offers functionality to generate graphical representations such as control flow graphs and call graphs.

2.4 k-Nearest Neighbours

The aim of the k-Nearest Neighbours (k-NN) classification problem is to find the k nearest data points in a dataset to a given query data point [42]. Let the query dataset be denoted as Q and the training dataset as T. The dataset T comprises of data that has already been classified, in the context of this project, T represents the set of open source class methods. Whilst Q denotes the dataset containing the obfuscated class methods that are awaiting classification based on the classification of the training dataset T. Each method in both datasets represents an n-dimensional object, where n is the number of features used for classification. A detailed discussion of the specific features used by the k-NN implementation in this project can be found in the *Section 3.3.2*.

The fundamental principle of k-NN is to classify these objects (i.e. the class methods) based on their proximity to other objects in the n-dimensional feature space. Objects that are similar to each other tend to be closer to one another in the feature space, whilst dissimilar objects are farther apart. Consequently, the distance between two objects is a measure of their dissimilarity. When presented with a new object from the query dataset Q, the k-NN algorithm identifies the k objects from the training dataset T that are nearest to the query object. Subsequently, it assigns the most common label among these k training objects to the query object. In this way, a method is classified by a majority vote of its neighbours.

In this project, k is set to 1, and thus each obfuscated class method in Q is simply assigned to the label of its nearest neighbour in T. In the context of the project, what this essentially means is that each obfuscated class method in Q is matched to the open source class method most similar to it in T.

Below is a simplified visual representation of the k-NN algorithm as implemented in this project. The data points, representing class methods, are plotted in two-dimensional space. The dimensions of the space represent the number of features utilised for classification. So in this case, the two axes labelled X1 and X2 correspond to the two features used to characterise the class methods. The three red data points denote the training dataset T, comprising of the open source class methods, while the blue data point represents the obfuscated class method awaiting classification.

As the value of k is set to one, the algorithm only considers the nearest neighbour from the training dataset closest to the obfuscated class method. In the plot, the nearest neighbour is identified by the connection point between the red data point and the blue data point. Consequently, the obfuscated class method is classified as or matched with this open source class method.

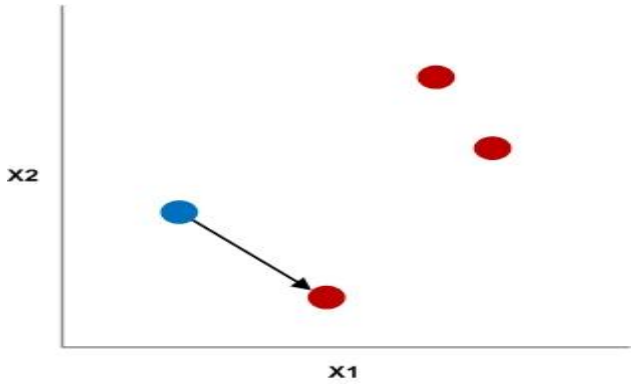

Figure 2: k-NN in two dimensional space

To conclude, k-NN is an intuitive, non-parametric classification algorithm that operates by calculating the most similar objects in an n-dimensional feature space. Within the context of this project, k-NN is used to effectively match class methods between open source and obfuscated APKs. As will be discussed shortly in the *Design* section, its efficacy is further improved with the support of other features incorporated in the design of the ACM application. All these factors have played a crucial role in the implementation of a code mapping infrastructure based on efficient and accurate match detection techniques.

3 Design

The following process flow architecture diagram provides a visual representation of the internal structure of the ACM application in terms of its subsystems, inputs, outputs and workflows. Designed to meet the requirements of the second research objective, the end-to-end process consists of a four-stage pipeline that receives two artefacts as input and delivers the results as a single output file. As will be discussed shortly, each subsystem is an independent unit, designed to carry out a discrete set of tasks and to prepare the input required by the next subsystem in the workflow.

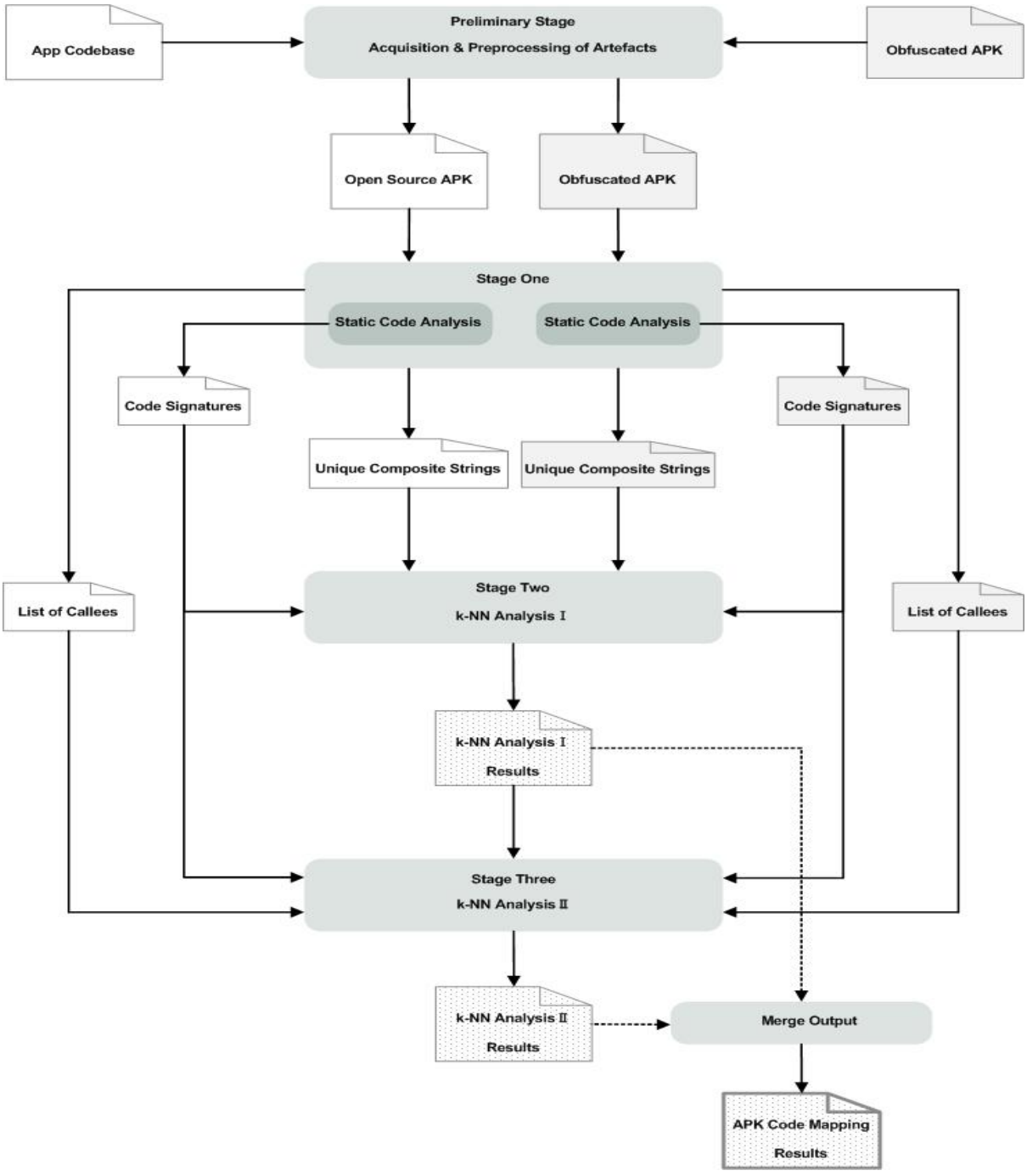

Figure 3: Process Flow Architecture Diagram

3.1 Preliminary Stage

3.1.1 Overview

Besides meeting the requirements of the first research objective, namely '*to acquire a set of APKs for development and testing purposes*', the overall aim of the *Preliminary Stage* is to ensure that all prerequisites are in place before the mapping process can commence. As will be discussed further on in this section, this is the stage where the required artefacts are downloaded, assembled, and verified before they can be used by the data preparation functions included in *Stage One*. In effect, the *Preliminary Stage* is not a component of the ACM application but a series of actions that are more or less relevant only to the development and testing process. This is because when the application is used '*live*' in the field, the APKs are typically precompiled and packaged, ready to be used by the application without the need for any preprocessing.

3.1.2 Environment Setup

Once the toolchain covered earlier in *Section 2.3* is installed, it is time to initialise the environment so the required artefacts can be prepared for use by the ACM application. As a first step, it is necessary to obtain the F-Droid APK from the official site [40] and load it in Android Studio as a newly configured project, complete with an Android Virtual Device (AVD) and an Android Emulator, to simulate a virtual phone, this being Google Pixel 3A for this specific project.

With the F-Droid utility app running on the virtual phone, a Java-based Android application is downloaded from the F-Droid app store, installed locally in the virtual environment and locating the app's *base.APK* [43] file within the data folder of the emulated device. This application package represents the *Obfuscated APK* depicted in the process flow diagram as one of the *Preliminary Stage* inputs.

The item *App Codebase*, the second input to the *Preliminary Stage*, is a reference to the open source version of the obfuscated app, which is sourced from *GitHub* by locating the project repository. Interestingly, most developers do not provide pre-built APKs on *GitHub* or else, when these are made available, they are often built using obfuscated code, rendering them unsuitable for recovering the open source APK. This means that it is necessary to manually build the APK by cloning the *GitHub* source code repository locally and setting up the project in Android Studio in accordance with the developer's instructions.

3.1.3 Obfuscation Removal

It is important that the open source *App Codebase* is checked for obfuscation settings which are normally found in the *build.gradle* file or its equivalent as it is common for developers to enable obfuscation using tools such as *ProGuard* [44]. Some common types of code optimisation techniques that lead to code obfuscation include:

(a)	Code Shrinking	This process entails identifying and removing unused code segments, reducing the application's size and improving its performance.
(b)	Identifier Name Obfuscation	A technique that involves the renaming of classes, methods and fields to shorter, meaningless names. The dual purpose

		of this is to conserve space by utilising shorter names and to safeguard intellectual property by complicating the comprehension of the code.
(c)	Method Inlining	By replacing invocations of frequently called methods with their bodies, method inlining improves the application's speed by eliminating the overhead associated with function calls.

If the APK build configuration is found to have settings that would directly or indirectly lead to obfuscation, it is important that they are disabled before re-building the package to ensure that the *Open Source APK* is free of obfuscated code.

3.2 Stage One

The first task of the application is to import two APKs, analyse the code and extract the data required for the matching process. Shown in the process flow diagram as *Stage One*, the ACM application is launched with two complete APK archives as input that would include all the compiled code, resources and assets. The diagram depicts this with an instance of an APK that has been compiled without obfuscation and packaged directly from the open source app codebase (*Open Source APK*) and an obfuscated Android app downloaded as an APK from an Android app repository (*Obfuscated APK*). The two archives are processed separately and in exactly the same way, with each APK going through a multi-stage pipeline that includes filtering, code analysis, fingerprinting and encoding.

The output from *Stage One*, which consists of several data sets for each APK, is extensive and conveys enough information about the organisation, structure and features of the code that the ACM application can perform the mapping between the *Obfuscated APK* and the *Open Source APK* without further need to access the APK archives.

A detailed discussion of the key aspects of *Stage One*, their implementation in the ACM application and the concepts behind the design will now follow.

3.2.1 Code Filtering

Classes

One particularly important component of *Stage One* is Androguard's *AnalyzeAPK()* module that returns an object of type *Analysis*, configured with the required features to explore the *.dex* files within the APK. One of the first tasks using the *Analysis* object is to distinguish between the classes that are native to the application and those which belong to 3rd party libraries that have been bundled with the *.dex* files.

This is achieved with the *find_classes()* module from the *Analysis* object which retrieves all classes contained in the *.dex* files. Importantly, these classes are listed using their fully qualified names, i.e. the package name combined with the class name, using a dot delimiter. The format streamlines the process of distinguishing between classes that belong to the ACM application and those of its dependencies, by simply using a regex rule to filter out unwanted classes. For instance, if the package name of one of the tested applications is *eu.faircode.email*, a regex expression of "eu.faircode.email*" would effectively capture only those classes contained in the package *eu.faircode.email*.

Methods

Once the applicable classes are shortlisted, the next step involves static code analysis where each class method is examined for its attributes, features and string literals. This is accomplished with Androguard's *find_methods()* routine which facilitates an iterative process that allows access to method properties within the boundaries of a class object identified by its class name.

However, before running the static code analysis, it is first necessary to filter out those methods that are not in scope for this process. Synthetic methods are automatically inserted by the Java compiler, meaning that they are not authored by the developers and as such are not included with the app

codebase. For this reason, synthetic methods are not eligible for the matching process and will therefore need to be removed from further analysis. All such methods have their names prefixed with the token “access$”, which makes it easy to identify and hence exclude.

Identifiers

It was necessary to devise a standard naming system for class methods that could be used to uniquely identify code artefacts throughout the analysis and matching process. The naming convention derives the *Method ID* from features and attributes that describe the class method under consideration which include the name of the class and method as well as the method’s parameter types, return type and access flags. All this information is obtained using the Androguard *get_method()* routine, which very much simplifies the process of determining the *Method ID*.

3.2.2 Unique Composite Strings

Rationale

The first output artefact of *Stage One* is essentially a collection of encoded strings. The source of these strings and the encoding method are discussed in the *Implementation* section below but it will suffice to say here that each string is a concatenated form of all the string literals defined within the boundaries of a class method. Furthermore, a concatenated string is only considered suitable for further processing if it can uniquely identify the class method within the scope of the APK. In other words, if it results that an APK has more than one copy of the same concatenated string; all instances of that string will be discarded. For this reason, and as illustrated in the process flow diagram, these string formations are referred to as *Unique Composite Strings*.

The significance of the composite strings and why they need to be unique becomes more evident when in *Stage Two*, these strings are used to match class methods across the two APK archives supplied as input to *Stage One*. Underlining this concept is the suggestion that if a unique composite string in the first APK is an exact match to a unique composite string in the second APK, then the corresponding methods must be logically identical. The idea has been successfully tested with the *FairEmail* application [45] and ground truth data, gathering enough evidence to support the hypothesis. The test case involved using unique composite strings to map class methods between *FairEmail*’s open source and obfuscated APKs and then comparing the outcome against the ground truth dataset. The outcome reported a total of 1,578 matches, all of which were confirmed to be correct, with zero false positives. *Section 4* provides more details on the *FairEmail* application and how ground truth was established.

Implementation

Individual strings are identified in class methods with the use of Androguard’s *get_instructions()* function, that provides access to the disassembled bytecode. Since in Dalvik bytecode, strings are declared using the “const-string” opcode, the strings contained in a method can be extracted simply by searching the opcode in the list of instructions obtained with *get_instructions()*.

The method strings are then sorted alphabetically to ensure that they are ordered, regardless of the sequence they have been declared. This measure is necessary before the strings can be

concatenated so that class methods which declare the same set of strings but in a different order would not yield a different unique composite string.

The idea of concatenating all the strings contained in a method to form a composite string rests on the premise that a composite string can be used as a fingerprint to uniquely identify a class method. Although individual strings can also make for good fingerprints, it is often the combination of multiple unique characteristics that provides a more reliable means of identification. This aspect is supported with empirical evidence gathered during the development of the ACM application, using both the open source and obfuscated variants of the *FairEmail* application. For instance, analysis of the open source APK found that class methods declared a total of 1,089 individual unique strings. In contrast, the same study reported 1,578 instances when the same APK analysis involved unique composite strings. This 45% increase in unique "fingerprints" is significant, not just because it confirms the direct correlation between the length and complexity of a string with its distinguishing unique qualities, but also because of the potential increase in class methods matched between the two APKs on the basis of unique composite strings.

Once the individual strings are ordered and concatenated, the composite form of the method strings are used to obtain a UTF-8 encoded cryptographic hash based on the SHA-256 one-way function. The reason for the transformation is not related to any security requirement but merely to enforce a standard length on all unique composite strings with the intent of facilitating easier handling of the string data at a later stage. From this point onwards, since the original strings are no longer used, any mention of unique composite strings is a reference to the encoded hash version of the cleartext concatenated strings.

With a complete list of unique composite strings paired with the *Method IDs* that identify the corresponding class methods, the final step is to remove any string entries that appear multiple times. This will ensure that only composite strings that are truly unique across the entire APK will be considered for *Stage Two*.

3.2.3 Code Signatures

Rationale

Once the matching methods across the two APKs are identified, *Stage Two* also contends that if a class method in the *Open Source APK* is logically identical to another class method in the *Obfuscated APK*, then the similarities are likely to extend to other members of the classes to which the two matching methods belong. The premise here is that once two class objects produce at least one pair of matching methods across the two APKs using unique composite strings, it is highly probable that the same class objects bear other similarities that cannot be identified using unique composite strings. Although efficient and highly accurate, the limitations inherent to using only unique composite strings are self-evident. For instance, it has already been mentioned that the ACM application was designed to drop multiple occurrences of the same composite string when present in a single APK. What this means is that all methods associated with these strings will not qualify for the matching procedure involving unique composite strings. Other shortcomings include the fact that it is not uncommon for class methods not to have any string literals or that there is always the possibility that a class method with a unique composite string does not produce a match.

It is clear therefore that in order to overcome these limitations, further drill down into the APKs will necessitate a different kind of match detection technique that does not rely on composite strings.

The key requirement here is that, except for class methods which have already been matched using unique composite strings, all methods pertaining to class objects that yielded at least one positive match using composite strings will need to be assessed. And this is where another type of input, derived from the static code analysis, comes into play. Identified in the process flow diagram as *Code Signatures*, these data sets describe the class methods in terms of their features and attributes such as dependencies, parameters and return types.

Code signatures are used by the k-NN analysis performed in *Stage Two* and *Stage Three* of the ACM application and are essentially the union of features serving as input to k-NN. It is important to note that the selection of these features was not arbitrary but rather, they were chosen based on their ability to enhance the matching performance of the ACM application. The process of how features were assessed is covered in *Section 3.2.4* so for now the discussion will limit itself to the six code features that made it to the final selection.

Parameters, Returns, and Access Flags

The first three features selected for inclusion in the code signature of a class method are the method's parameter types, return type and access flags. Their relevance in terms of a method's distinguishing attributes is mainly based on the fact that these features are always explicitly or implicitly defined in a class method.

The procedure to read the values assigned to these features from a method object is very much the same for all three and relies exclusively on the functionality provided by the Androguard API. In essence, when a class method is queried with the *get_descriptor()* function, the list of parameters types and the return type associated with the method are returned. Similarly, a call to Androguard's *get_access_flags_string()* function returns the method's access control flags such as public, private and protected.

There is also a fourth element included in the method's code signature. This is the parameter count, which unlike the other features, does not directly involve the Androguard API but is derived from the list of parameter types returned by the the *get_descriptor()* function.

Once obtained, the value of each feature is stored in a list together with the corresponding *Method_ID*. This procedure is carried out for each method object that has been shortlisted for mapping analysis which will commence in *Stage Two* (refer to *Code Filtering* in *Section 3.2.1*). This is then followed by transforming the list into binary representation using class *MultiLabelBinarizer* from sklearn [46]. The following example demonstrates how *MultiLabelBinarizer* transforms such a list into a Machine Learning (ML) readable format:

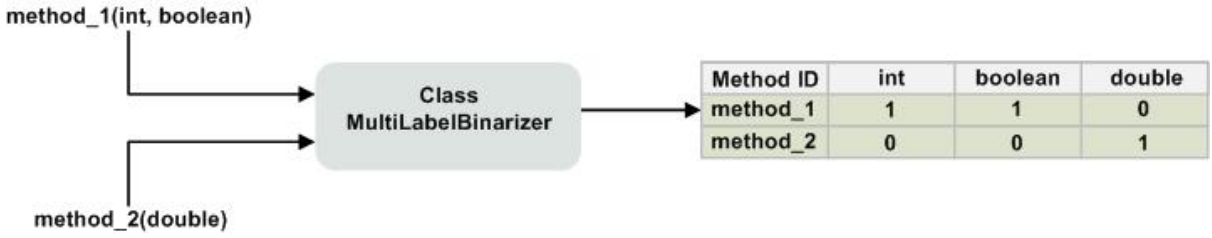

Figure 4: MultiLabelBinarizer

Method Invocations

Outbound method invocations represent the number of methods called by the method being analysed. This metric is derived by tallying the count of methods returned by the Androguard *get_xref_to()* function.

Conversely, inbound method invocations represent the number of times the method under analysis is invoked by other methods. This is calculated by counting the number of methods returned by the Androguard *get_xref_from()* function.

Code Signature

The code signature that describes a class method and which serves as the input for the k-NN analysis is effectively a construct made up of these six features as illustrated by the following example that shows the code signature for a single class method:

Table 2: Code Signature

	Parameter Types			Return Types		Access Flags		Miscellaneous		
Method ID	int	boolean	double	void	String	public	private	Inbound	Outbound	# Parameters
method_1	1	1	0	1	0	1	0	2	1	2

3.2.4 Feature Selection

The various code features considered for input to k-NN and the reasons behind the selection of the feature set that actually made it to production will now be discussed.

Central to the evaluation process is a performance baseline, representing the sum of all features under consideration. Using this baseline, it was possible to benchmark all the features by systematically disabling each feature and observing the deviations from the baseline and therefore the impact registered on the performance of the ACM application.

Besides the six features that were ultimately selected for implementation (*refer to Code Signature in Section 3.2.3*), the following is a list of code features that were also examined but which did not make the grade:

(a) McCabe Complexity

McCabe Complexity, drawn from the work of Kontogiannis et al. [27] as discussed in *Section 2.2.5* Metric-Axes Clustering, quantifies the number of independent paths in a method. It can be calculated as 1 + *d*, where *d* is the number of control decision predicates in the method.

(b) Number of Variables Declared

This metric measures the number of variables declared within a method. Variable declarations are identified using the disassembled Dalvik bytecode, with opcodes such as "const/4" and "const/16."

The evaluation of all features under consideration was carried out on the *FairEmail* application using the open source and the obfuscated APKs and utilising the same ground truth as in *Section 4.3*. The final results from the benchmarking exercise, that also include the set of implemented features, (marked with *****) are as follows:

Table 3: Feature Selection

Features Individually Removed	Metrics		
	Precision	Recall	F-measure
Baseline	**0.996874**	**0.844992**	**0. 914671**
McCabe Complexity	0.996874	0.844992	0. 914671
Number of Variables Declared	0.996874	0.844992	0. 914671
Outbound Method Invocations*	0.989365	0.814324	0.893351
Inbound Method Invocations*	0.993758	0.843220	0.912321
Number of Parameters*	0.996243	0.843361	0.913449
Parameter Types*	0.996787	0.821940	0.900958
Return Type*	0.996068	0.805938	0.890973
Access Flags*	0.996550	0.842036	0.912800

One can observe that the precision, recall and F-measure did not change with the removal of the *McCabe Complexity* and the *Number of Variables Declared* features. This is a clear indication that these two features do not provide the ACM application with any performance improvements. Consequently, it can be inferred that these features are non-informative and redundant for the k-NN analysis, as they do not contribute to the ACM application's ability to accurately match methods.

On the other hand, the removal of any other feature had a negative effect on the evaluation metrics, which suggests that the ACM application benefits from the use of these features, both in terms of better accuracy and improved matching capabilities.

3.2.5 List of Callees

Rationale

Recall that in *Stage Two*, all match detections are confined to classes that yield at least one positive match using the unique composite string method, even when the mapping involves code signatures and k-NN analysis. Although this approach is sufficient to cover the entire scope of this category of class objects, the method does not extend the entire footprint of the code contained in the APK. In fact, all those classes that for some reason do not register a single match in one of their methods when examined for unique composite strings would be completely ignored.

Stage Three builds on the logical deductions drawn from the evidence presented earlier for *Stage Two* (see *Section 3.2.3*) which suggest that if two different APKs contain the same unique composite string, then the corresponding class methods are logically identical and the class objects to which they belong are likely to contain other methods that would also yield a match. What *Stage Three* does differently is that instead of relying on unique composite strings, it uses the methods (the *callees*) invoked from within the class methods (the *callers*) that have already been matched in *Stage Two* to determine which methods, and hence classes, are in-scope for the k-NN analysis. The basic premise here is that if two caller methods were found to be logically identical in *Stage Two*, their respective callees should also exhibit close similarities. Furthermore, *Stage Three* asserts that once

two callees from different APKs are found to be logically identical, the other methods included in the class objects to which they belong should also demonstrate similar properties.

Implementation

Obtaining the list of callees defined in a class method requires the use of Androguard's *get_xref_to()* function, which returns a dictionary of all classes and their methods used by the caller, i.e. the method under review. Also, as per procedure highlighted in *Section 3.2.1*, any synthetic callee methods identified during this process are excluded from the list.

3.3 Stage Two

Once *Stage One* has prepared all the material required by the matching algorithms, the two APK archives are set aside and the mapping process can begin. In terms of output from *Stage One*, and as illustrated by the process flow diagram, the artefacts that are of interest to *Stage Two* are the collection of Unique Composite Strings and the Code Signatures.

The first task of the mapping process is to use the unique composite strings to locate class methods that are common to both the *Open Source APK* and the *Obfuscated APK*. The class objects which contain the methods that yielded a positive match using this form of mapping detection are then selected for k-NN analysis. The idea here is to inspect these classes in more detail in an attempt to match methods that, when using unique composite strings, either failed to qualify or did not yield a positive match.

For this part of the mapping process, the code signatures derived from the *Open Source APK* are used as the training data for the k-NN algorithm whereas the code signatures from the *Obfuscated APK* serve as the query dataset. The specifics of the k-NN model and the rationale behind certain decisions such as why the Euclidean distance was the metric of choice or the basis for setting the k value to 1, are all covered in *Section 3.3.2*. However, it will suffice to say here that the results from the unique composite strings and the k-NN analysis are merged and the data refined by removing low confidence matches, using one of the filtering techniques, also discussed separately in *Section 3.3.3*.

3.3.1 String Matching

Recall from *Section 3.2.2* that a unique composite string consists of a series of string literals that have been ordered, concatenated and transformed into a fixed-size 32-byte SHA-256 hash. Also, each hash is UTF-8 encoded to ensure safe handling of the hash value during the mapping process, especially in those instances when the hashed data might contain special characters or non-ASCII characters.

The actual comparison of two unique composite strings relies on Python's equality operator '==' which internally is implemented as a C language *memcmp()* function that tests if two string objects contain the same data.

3.3.2 k-NN Analysis

Section 2.4 has already covered the basic concepts of k-NN and its application in a general context. It is now time to take a closer look at this algorithm and examine the specifics within the scope of the match detection techniques implemented in the ACM application.

Implementation

The application code responsible for all aspects of the k-NN algorithm is based on the *scikit-learn* machine learning library. Specifically, it relies on class *sklearn.neighbors.KNeighborsClassifier* [47] to perform the k-NN functions, with *NumPy*, another Python library, providing the required support for large array operations.

k Neighbours

The choice of k, the number of neighbours in the k-NN algorithm, plays an important role in determining the accuracy of classification for a query point. In order to choose the most appropriate value for k, the performance of the ACM application was assessed across a number of values for k, using the open source and obfuscated variants of the FairEmail APK as input.

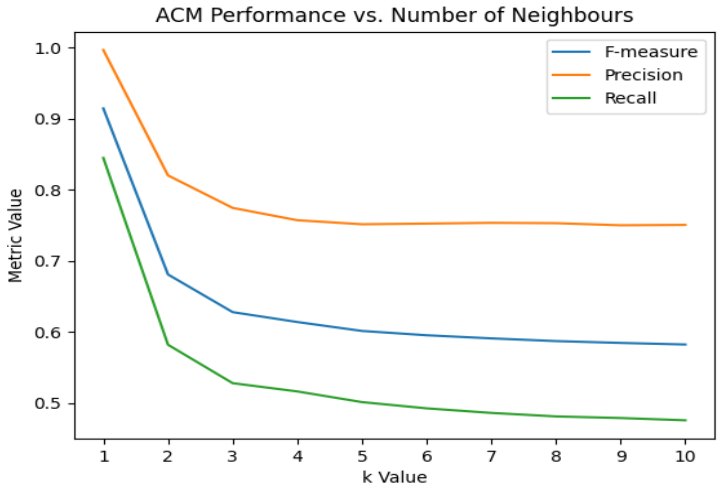

Figure 5: k Neighbours Selection

As can be observed from the graph, the analysis reveals that setting the value of k to 1 yields exceedingly better outcomes in terms of precision, which reflects the proportion of correctly identified matches among all predicted matches. Similarly, the recall metric, representing the proportion of correctly identified matches out of all possible matches, also demonstrates optimal performance when k is set to this value. As expected, the improved performance also has an influence on the F-measure, which takes into account both the precision and the recall values. A more detailed insight into these metrics is included in the *Evaluation* Section.

Weighting of Neighbour Points

In k-NN models, it is a requirement to specify the weighting of training data points. One approach that is frequently used is Gaussian weighting, which assigns less weight to training points that are farther away from the query point. Another method that is also commonly used is uniform weighting where each training point is treated equally, assigning the same weight to all points within the neighbourhood of the query point.

However, in scenarios like the one presented in this project, where k is set to 1, weightings schemes like the Gaussian and the uniform would essentially exhibit the same behaviour simply because only the nearest neighbour is considered, and there is no aggregation of multiple neighbours' outputs. Applying different weights to training points only becomes meaningful when there are multiple nearest neighbours (i.e. k > 1), whose outputs need to be aggregated. But since this is not the case with the k-NN model as implemented in *Stage Two* and *Stage Three* of the ACM application, the uniform weighting is applied.

Distance Metric

An appropriate distance metric for measuring the similarity between data points is also required. Three measures i.e. the Euclidean, Manhattan, and Cosine were considered [48]. Below are their mathematical definitions to measure the closeness between two vectors x and y, where:

$x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$

(a) Euclidean Distance (ED)

A special case of the Minkowski distance [49], with p = 2. It is an extension of the Pythagorean Theorem. This distance represents the root of the sum of the square of differences between the opposite values in vectors.

$$ED(x, y) = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2}$$

(b) Manhattan Distance (MD)

Also known as City block distance, it is another case of the Minkowski distance with p = 1. This distance represents the sum of the absolute differences between the opposite values in vectors.

$$MD(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

(c) Cosine Distance (CD)

The Cosine distance, also called angular distance, is derived from the cosine similarity that measures the angle between two vectors. The Cosine distance is obtained by subtracting the cosine similarity from 1.

$$CD(x, y) = 1 - \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \sqrt{\sum_{i=1}^{n} y_i^2}}$$

To assess the performance of each distance metric, the ACM application was run using the open source and obfuscated versions of the FairEmail APKs, producing the following outcome:

Table 4: Distance Metric Selection

Distance	Metrics		
	Precision	Recall	F-measure
Euclidean	0.996874	0.844992	0. 914671
Manhattan	0.996585	0.842230	0.912930
Cosine	0.997173	0.841282	0.912619

As is evident from the table, the Euclidean distance achieved the best recall and F-measure, with precision only slightly less than that of Cosine. Hence, Euclidean distance was selected for use in the k-NN model in this project.

Training & Query Data

The training and query datasets for the k-NN are the code signatures derived from the static code analysis performed in *Stage One* on the *Open Source APK* and the *Obfuscated APK* respectively. This means that the k-NN algorithm will *predict* the class method in the *Open Source APK* that the query data point i.e. the class method in the *Obfuscated APK*, is most similar to; in which case the two class methods will be considered a match.

As can be observed from the following table, analysis shows that recall goes up when using code signatures derived from the *Obfuscated APK* to fit the model. Unfortunately, these gains in the recall metric are cancelled out by a simultaneous drop in the precision metric. What this means for the matching algorithm is that while using data from the *Obfuscated APK* to fit the model increases the number of true positives, it also introduces more false positives. As will be discussed further in the *Evaluation* section, minimising false positives is key to the success of the algorithm, which is the reason behind the decision to fit the k-NN model with the code signatures derived from the *Open Source APK.*

Table 5: Training Data Selection

Training Dataset	Metrics		
	Precision	Recall	F-measure
Open Source APK Code Signatures	0.996874	0.844992	0. 914671
Obfuscated APK Code Signatures	0.903031	0.926197	0.914467

Ablation Study

To validate the concept behind the implementation of k-NN in *Stage Two*, where it was performed on the class object rather than the entire APK, an ablation study was carried out to determine the effectiveness of this approach when compared to applying k-NN to map all methods between the two APKs, without the class boundaries. The results of this study are presented in the table below:

Table 6: k-NN Scope Performance

k-NN Scope	Metrics		
	Precision	Recall	F-measure
Class	0.996874	0.844992	0. 914671
APK	0. 581253	0. 610814	0. 595667

The data leaves no doubt as to the benefits of using class-level k-NN in *Stage Two* of the ACM application. The improved outcomes across all metrics can be attributed to the fact that, since the data domain used for training and query in the k-NN algorithm is limited in scope, the mapping process is more focused, leading to better accuracy in match detection. In other words, given that the k-NN algorithm is only concerned with measuring the similarities of code signatures between two classes as opposed to involving the entire class inventory of the APKs, there is less opportunity for false positives due to the reduced number of data points.

3.3.3 Filtering

With the match detection out of the way, the outcomes from the unique composite strings and the k-NN analysis are merged together, in preparation for the next task aimed at ensuring the integrity of the results in terms of quality, accuracy, and reliability. The process, which satisfies the requirements of the third research objective, features the following filtering algorithms:

Table 7: Filtering Types

Filter Type	Algorithm	Description
I	Discard matches where the k-NN distance is greater than 0	This criterion is based on the rationale that a k-NN distance greater than 0 indicates dissimilarities between the matched methods, suggesting that the match may not be authentic.
II	Matches are removed where an open source method is matched to more than one obfuscated method	This filtering criterion was devised on the basis that if an open source method is matched to multiple obfuscated methods; it suggests that the methods lack distinguishing properties that may negatively impact match detections.

To measure the effectiveness of the two filtering algorithms, they were applied directly to the matching outputs from *Stage Two* and *Stage Three*, using various combinations in order to observe the effect that different filtering arrangements have on the final outcome of the ACM application. The test results are summarised in the table below:

Table 8: Filtering Performance

Filter Type		Metrics		
Stage Two	**Stage Three**	**Precision**	**Recall**	**F-measure**
None	None	0.886651	0.874901	0.880736
I	I	0.942331	0.839735	0.888080
II	II	0.999369	0.839958	0.912756
I	II	0.944198	0.847152	0.893047
II	I	0.996874	0.844992	0.914671
I & II	I & II	1.0	0.807203	0.893318

One of the most interesting aspects of the test results is that it is possible to obtain a reasonably good outcome without filtering any of the outputs. However, the fact that just close to 89% of the final matches were correct indicates that there is room for improvement. And yet, when filtering is applied, precision goes up while recall reports a somewhat downward trend, suggesting that while unreliable matches are removed, some valid entries are also discarded along with the incorrect ones. The last entry in the table is a perfect example of this behaviour, with both filters applied to each stage, culminating in a perfect precision score (i.e. all final matches are correct) and an extremely poor recall performance (i.e. highest amount of discarded valid matches).

It is clear from the test data that the use of filters is always going to be a balancing act and hence trade-offs must be made. While filtering increases the likelihood of the remaining matches being correct, the risk of discarding valid matches cannot be fully eliminated. Ultimately, the decision was made to apply filtering to both *Stage Two* and *Stage Three* as it was deemed that the associated benefits far outweigh the risks.

Using F-measure as a guiding criterion, one can conclude that the most ideal filtering configuration should be based on Type II for *Stage Two* and Type I for *Stage Three*. As the data suggests, this option strikes a balance between achieving accurate predictions and retaining the highest number of correct matches. However, this does not mean that other combinations cannot be considered as viable options. For instance, using Type II in both stages yields a very high precision score while still maintaining an acceptable recall. In fact, one could argue that this would have made a better filtering option due to its higher precision score. On those grounds alone, the argument could make sense. However, the option chosen for the ACM application exhibits a better balance between precision and recall while still managing to score high on both counts. Which is precisely why it made it a more suitable option for a utility application such as the ACM, where the effectiveness of its outcome is determined by the volume of match detections and the accuracy of those matches.

3.4 Stage Three

Building on the outcome from *Stage Two*, *Stage Three* widens the scope of the k-NN analysis by including methods that were up to this point, inaccessible for reasons that will be explained shortly. Recall that in *Stage Two*, all methods matched on the basis of unique composite strings or k-NN had one thing in common – the scope of the mapping was limited to the class objects that yielded at least one positive match using unique composite strings. This means that any class which for some reason or other was not successful when the APKs were first scanned for matching composite strings were simply regarded as unsuitable for further analysis. Considering that this took place right after the first attempt at finding matches, it would appear that dropping these classes so early in the process would be an illogical thing to do. After all, as has already been discussed in *Section 3.2.3*, the limitations inherent to using unique composite strings cannot be simply overlooked if the ACM application is expected to cover as much footprint of the APK code as possible.

The approach used in *Stage Three* uses callees as the main driver to drill further into the code structure of the APK. In other words, methods (the *callers*) that have been matched in *Stage Two* are re-examined and their call tree inspected for methods (the *callees*) that belong to classes which have not yet been considered in the match analysis. Hence, on the basis of following the call hierarchy of the methods matched in *Stage Two*, it is possible to branch out and reach other class objects that would have otherwise remained inaccessible.

Also included as a supplementary task in *Stage Three* is the final merge of the results from both *Stage Two* and *Stage Three*, referred to as *Merge Output*, the final step in the process flow diagram. Basically, on completion of *Stage Three*, k-NN results are combined with the k-NN output from *Stage Two*, and once the data is cleaned of any duplicate entries, the final outcome from the ACM application is delivered as a single CSV file, consisting of several fields that would include the names of the matched methods, their location in the APKs as well as their k-NN distance.

3.4.1 Ablation Study

To determine the effectiveness of including *Stage Three* as an additional and final step in the match detection pipeline of the ACM application, an ablation study was carried out to determine the extent of its contribution, if any, to the overall matching process. The investigation involved running two instances of the ACM application, with the first instance configured to execute all the three stages and the second instance configured to run only the first two, i.e. *Stage One* and *Stage Two*.

As with all studies included in the *Design* section, the evaluation was carried out using both the open source and the obfuscated versions of the *FairEmail* application. The final results of the ablation exercise are summarised in the table below:

Table 9: Efficacy of Stage Three

ACM Configuration	Metrics		
	Precision	Recall	F-measure
With Stage Three	0.996874	0.844992	0. 914671
Without Stage Three	0.999362	0.830419	0.907091

Looking at the results, several important observations emerge. For instance, the inclusion of *Stage Three* led to an increase in false positives, and hence a drop in precision. However, Stage Three also improves recall, an indication that more correct matches are detected at the expense of the increase in false positives. Therefore, there is a trade off between matching more methods as opposed to

maintaining a very slightly higher level of accuracy. Given that the drop in precision is minimal compared to the increase in recall and that the score for F-measure is higher when *Stage Three* is present, the decision was made to incorporate *Stage Three* as part of the ACM application.

3.4.2 Implementation

Bar for the fact that in *Stage Three*, the mapping process is driven by callees as opposed to unique composite strings as in *Stage Two*, there is a lot in common in the way the two stages implement match detection between the *Obfuscated APK* and the *Open Source APK*. Suffice to say that they share exactly the same k-NN model, based on the configuration covered already for *Stage Two* in *Section 3.3.2*. The similarities also extend to the filtering procedure of the k-NN results and the dependency of the *Stage Three* k-NN analysis on the outcome from *Stage Two*, as illustrated in the process flow diagram. In essence, the core of the matching algorithm in *Stage Three* is implemented as follows:

(a) The callees of each method matched in *Stage Two* are identified and checked if they belong to class objects that have already been mapped, using k-NN analysis in Stage Two. If it is the case, the callee methods and their corresponding classes will not be considered for further analysis. Otherwise, each pair of matching class methods from *Stage Two* could give rise to one of two possible scenarios:

(a-i) In scenarios where each method in a matched pair from *Stage Two* consists of only a single callee, the algorithm proceeds to match these callees without using k-NN. This approach follows the same line as the one used earlier where two class methods from different APKs are matched on the basis of two identical unique composite strings. Once again, as was indeed the case with unique composite strings, the algorithm employs k-NN analysis to match the remaining methods in the callee classes across the two APKs. This is shown as k-NN Analysis II in the process flow diagram.

(a-ii) In the event that a pair of matching methods from Stage Two include multiple callees in their code, the algorithm performs two cycles of k-NN analysis, also illustrated as k-NN Analysis II in the process flow diagram. The first k-NN involves the callees themselves, matching the open source callees with the obfuscated callees. The second instance of k-NN is carried out on the remaining methods within the classes of the callees.

(b) Similar to the data filtering procedure used in *Stage Two* and as explained in more detail in *Section 3.3.3*, the *Stage Three* results are finalised with the inspection of the k-NN outcome and the removal any low confidence and unreliable entries.

4 Evaluation

In line with the fourth research objective, this section undertakes the task of evaluating the performance and efficiency of the ACM application. To ensure a comprehensive assessment, several key areas will be considered:

(a) Accuracy

The focus here is to determine the application's accuracy in matching open source class methods to their obfuscated counterparts. The purpose of this is to ascertain the reliability of the application by evaluating that validity of the matches.

(b) Coverage

Assessing the number of class methods that have been matched successfully provides an insight into the thoroughness of the ACM application. This aspect of the evaluation helps understand the extent to which the ACM application covers the code footprint.

(c) Speed

Another important factor to consider is performance as in the time it takes for the application to complete a full run. In essence, this provides a measure of the application's utility and its applicability to real-world scenarios.

To summarise therefore, the evaluation aims to provide a 360-degree view of the ACM application by thoroughly examining each of these key performance indicators that, in more ways than one, determine the quality and completeness of the project deliverable.

4.1 Setup

The evaluation setup was installed with four different Android applications, provisioned in both open source and obfuscated version, as per procedure outlined in *Section 3.1*. A basic profile for each of these applications is provided by the table below:

Table 10: Test Applications

App Name	Version	Lines of Code	# Class Methods	Use Case
FairEmail [39]	1.2143	88,203	3,776	Fully-featured, privacy oriented email application for Android.
OpenKeyChain [52]	6.0.2	52,822	3,324	Stores and manages encryption keys, facilitates key exchange, and encrypts/decrypts messages for secure communication.
PCAPdroid [53]	1.7.0	18,144	1,162	Privacy-friendly app for tracking,

				analysing, and controlling app connections, with features for exporting and inspecting traffic.
TrackerControl [54]	2024.01.03	2,282	128	Allows monitoring and control of data collection by mobile apps, promoting user privacy and data transparency.

4.2 Evaluation Metrics

The following three widely adopted metrics [50, 28, 51] are used to assess the performance of the ACM application:

Table 11: Evaluation Metrics

Metric	Formula	Explanation
Precision	$\frac{TP}{TP+FP}$	Evaluates the fraction of correct matches out of all matches made.
Recall	$\frac{TP}{TP+FN}$	Quantifies the number of correct matches made out of all possible matches that could have been made.
F-measure	$\frac{2 \times Precision \times Recall}{Precision + Recall}$	A combined metric that takes both precision and recall into account using a single score. It is calculated as the harmonic mean of precision and recall, providing a balanced measure of the algorithm's performance.

In the context of this project, true positives (TP) refer to the number open source methods correctly matched to obfuscated class methods. On the other hand, false positives (FP) are *Type I* errors and represent the number of open source class methods incorrectly matched to class methods in the obfuscated code. Additionally, false negatives (FN), known as *Type II* errors, represent the number of class methods in the open source code that were not matched to any class method in the obfuscated code.

In assessing the performance of the ACM application, the priority is to ensure the accuracy of the matches rather than attempting to match every class method between the two APKs, which underscores the importance of prioritising precision over recall. The reason behind this is because the primary goal of the ACM application is to accurately determine the extent of similarity between the two APKs, which is best achieved by obtaining a reasonable number of highly confident matches rather than a high volume of matches that are deemed unreliable. In essence, quality takes precedence over quantity.

However, it is important to not disregard recall entirely. The metric remains relevant because determining if equivalence between APKs exists also depends on having a sufficient number of matches. In other words, relying solely on a small percentage of high confidence matches may lead to erroneous conclusions. The key here is the need to strike a balance between reliability and volume as both are crucial to achieve results that are dependable.

4.3 Determining the Ground Truth

In order to calculate the precision, recall, and F-measure metrics, it is essential to first establish a ground truth for each application. The process involved inspecting the *build.gradle* file of the obfuscated APK, which specifies the obfuscation rules set by the developer. The inspection revealed that the method and class names remained unchanged between the open source and obfuscated APKs, a characteristic that made it possible to establish a reliable ground truth.

The process of creating the application's ground truth involves obtaining the Method IDs of the class methods in the open source and obfuscated APKs. Because the method and class names were not affected by the obfuscation, they remained the same in both APKs, which made it easy to build the ground truth by simply matching their Method IDs. Subsequently, a CSV file was generated, containing the Method ID of each open source class method alongside the Method ID of its obfuscated equivalent.

It is important to clarify that the absence of obfuscation in method and class names does not undermine the efficacy and validity of the proposed approach. As is clearly demonstrated in *Section 3*, method and class names play no role in the analysis or matching of class methods. Therefore, whether the open source and obfuscated APKs have identical or different method and class names does not bias or influence the results presented in this section. It merely simplifies the creation of a ground truth for evaluation purposes.

Furthermore, in line with the code filtering procedures discussed in *Section 3.2.1*, synthetic methods are not considered for the matching process by the ACM application, meaning that they are also excluded from the ground truth. By doing so, the evaluation provides a more accurate representation of the application's performance, focusing solely on code written by the developers.

4.4 Accuracy & Coverage

Once the ground truth for each application is established, the assessment of the ACM application's ability to map methods between the open source and obfuscated APKs can begin. The results of this assessment are presented in the table below:

Table 12: ACM Performance

Applications	Metrics			
	# Matches Made	Precision	Recall	F-measure
FairEmail	3199	0.996874	0.844992	0. 914671
OpenKeyChain	1945	0.992288	0.881325	0.933521
PCAPdroid	934	0.994647	0.800172	0.886874
TrackerControl	121	1.0	0.945312	0.971888

4.4.1 Precision

Analysing the precision across all four applications reveals strong performance, with over 99% correct matches. Such high precision highlights the effectiveness of the ACM application in accurately mapping methods between the open source and obfuscated APKs.

Upon closer inspection of the false positives, two distinct scenarios that lead to errors emerge. One scenario takes place during k-NN Analysis I, while the other arises in k-NN Analysis II. The errors that originate from k-NN Analysis I account for 60% of all observed errors, while the remaining 40% stem from k-NN Analysis II.

k-NN Analysis I Errors

As k-NN Analysis I matches one class from the open source APK to precisely one class in the obfuscated APK, any errors encountered involve discrepancies when matching methods within the scope of a single class. An instance of such an error was identified during the evaluation of the *FairEmail* application.

In this case, the class method *onDateAfter()* from the open source APK was incorrectly matched with *onDateBefore()* in the obfuscated APK. The two methods belong to *FragmentRule*, a class that is shared by both APKs. A manual inspection of the source code revealed that the two methods are syntactically and semantically equivalent, bar for two lines of code where the methods are referencing different class objects, as can be verified from the following code excerpts (*this.tvDateAfter* vs *this.tvDateBefore*):

onDateAfter()

```
private void onDateAfter(Bundle args) {
    boolean z = args.getBoolean("reset");
    long j = args.getLong("time");
    if (z) {
        j = 0;
    }
    this.tvDateAfter.setTag(Long.valueOf(j));
    this.tvDateAfter.setText(j == 0 ? "-" : this.DF.format(Long.valueOf(j)));
}
```

onDateBefore()

```
private void onDateBefore(Bundle args) {
    boolean z = args.getBoolean("reset");
    long j = args.getLong("time");
    if (z) {
        j = 0;
    }
    this.tvDateBefore.setTag(Long.valueOf(j));
    this.tvDateBefore.setText(j == 0 ? "-" : this.DF.format(Long.valueOf(j)));
}
```

Figure 6: Example of k-NN Analysis I Error

However, due to the features employed in the k-NN analysis, this discrepancy was not captured. Consequently, the features extracted from both methods were identical, leading to the algorithm erroneously matching the two methods during k-NN Analysis I.

One way of addressing the shortcoming would necessitate the inclusion of additional features in the k-NN analysis to help it narrow down differences between class methods. For instance, incorporating the method names of the callees as a feature can possibly mitigate this type of error.

k-NN Analysis II Error

In k-NN Analysis II, there can be instances when multiple classes from both APKs are involved in the matching process. Consequently, methods originating from different classes may be matched incorrectly because they contain identical source code and hence, share the same code signatures. An example of this type of error was encountered during the evaluation of the *FairEmail* application.

In this case, class method *getList()* from class *SimpleTask* was incorrectly matched to *getList()* in class *TwoStateOwner*. Upon examining these methods, it was revealed that every line of code was the same, resulting in identical features being extracted. As a result, the algorithm erroneously matched these methods during k-NN Analysis II, despite them originating from different classes.

In contrast to the errors encountered in k-NN Analysis I, adding additional features to better compare and distinguish class methods would not have any effect in this particular case because the mismatched methods in k-NN Analysis II have identical source code. Instead, it might be more beneficial to include features that characterise the class to which the method belongs. While this concept remains untested in this project, such features could include metrics like the total number of methods in a class. By incorporating class-related features, it could be possible to mitigate errors like those observed here.

4.4.2 Recall

The recall metric also showed strong performance, with each application correctly predicting over 80% of all possible matches. An analysis of the false negatives, which represent the open source class methods that could have been matched to their obfuscated equivalent but failed, reveals additional insights. Taking the evaluation of the *FairEmail* application as an example, it had 18% of the open source class methods not matched at all. An inspection of these false negatives revealed that they were caused by one of these three conditions - 46% of the time it was due to the way the k-NN analysis was implemented, and the other 54% of the time was because matches were removed during filtering, or because the classes to which the false negative methods belong were never considered in the matching process.

k-NN Implementation

In the context of the k-NN model, the open source class methods serve as the training dataset, while the obfuscated class methods act as the query dataset. Hence, for an open source class method to be matched with an obfuscated class method, it needs to be the nearest neighbour to at least one method in the query dataset. This requires its code signature to be the most similar to that of at least one obfuscated method. However, in practice, there are instances where an open source method is not the nearest neighbour to any obfuscated method and thus the open source method is not matched at all.

These instances of missed matches stem from discrepancies in the code signatures, which are attributable to the obfuscation techniques employed. Specifically, the culprit is method inlining, which negatively impacts the number of identified outbound method invocations. With method inlining, method invocations are incorporated directly into the body of the method, thereby decreasing the number of outbound method invocations that can be detected. Consequently, the dissimilarity between the code signatures of the open source and the obfuscated code results in the open source method failing to be matched to any method, leading to the false negatives observed here.

Filtering

As discussed in *Section 3*, low-confidence, duplicate and incorrect matches are filtered out in both instances that involve k-NN analysis, the main aim being the mitigation of Type I errors. However, while the filtering successfully accomplishes this, it also inadvertently removes some correct matches too, contributing to some of the Type II errors being observed.

Matching Scope

The open source classes considered during k-NN Analysis I and k-NN Analysis II in *Stage Two* and *Stage Three* respectively were:

(a) k-NN Analysis I

Classes containing a method with a unique composite string that has an identical match in the obfuscated APK.

(b) k-NN Analysis II

Classes that contain the methods defined as callees in class methods matched during k-NN Analysis I.

Consequently, a class that does not fall into one of these two categories will not be included in either stage of the matching process. As a result, all member methods of these classes will remain inaccessible, giving rise to false negatives.

4.4.3 F-Measure

The F-measure serves as a summary metric, providing a consolidated assessment of the ACM application. Across all four test applications, the F-measure is consistently high, with even the lowest score exceeding 0.8. These scores affirm the ACM application's ability to maintain both false positives and false negatives to a minimum. In effect, this demonstrates that the ACM is capable of accurately matching a substantial number of methods between the APKs. Overall, the strong F-measure scores validate the algorithm's efficacy and its suitability for practical applications such as code versioning, cyber security and ethical oversight concerning the use of open source code.

4.5 Runtime Efficiency

The following graph illustrates the relationship between the processing time and the number of lines of code in each of the four test applications. The ACM application was benchmarked on a 12th Gen Intel Core i7 1.70GHz workstation, installed with 16GB of RAM.

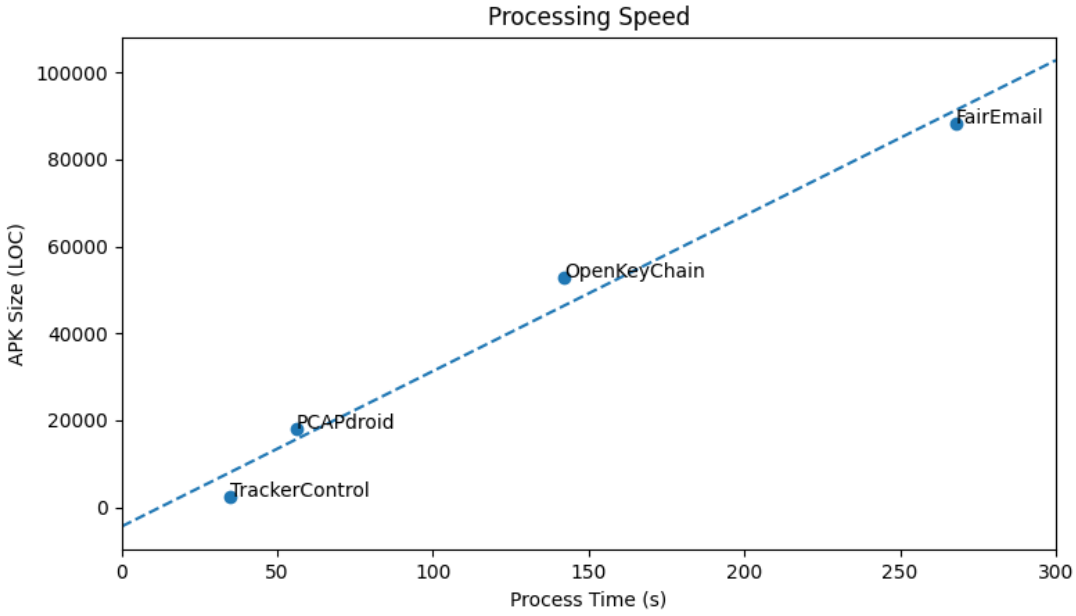

Figure 2: Processing Speed

The plot demonstrates a linear relationship between runtime and the number of lines of code in the APK. As a result, the runtimes remain relatively small, with the ACM application taking less than 5 minutes to process *FairEmail*, the largest test Android application. These efficient runtimes highlight the suitability of the ACM application for real-world scenarios.

A breakdown of the runtime performance is provided in *Table 13*, detailing the time spent in each stage of the ACM application. The breakdown reveals that the majority of time is dedicated to analysing the APKs in *Stage One*. In the case of *FairEmail*, *Stage One* constitutes 58% of the total processing time. This is to be expected, given that nearly 180,000 lines of code across both APKs had to be analysed.

Table 13: Processing Time

Applications	Time (s)			
	Stage One	Stage Two	Stage three	Total
FairEmail	155	72	41	268
OpenKeyChain	54	70	18	142
PCAPdroid	46	7	3	56
TrackerControl	33	1	1	35

4.6 Additional Testing

In addition to testing the performance of the ACM using the open source APK and its obfuscated variant for the four applications presented in *Section 4.1*, two additional tests were carried out to further evaluate the ACM application.

4.6.1 Test Case 1

This section introduces a more challenging scenario to evaluate the ACM application's performance in a real-world context. Unlike the previous test of *Section 4.4*, where the APKs supplied as input to the ACM application were practically identical except for the obfuscation; this test incorporates additional code embedded in the open source APK. The purpose of this is to determine whether the ACM application would mistakenly match these additional call methods to methods in the obfuscated variant.

The test was carried out in a similar manner to the test in *Section 4.4*, with the exception that the open source code was modified to include additional classes, in effect increasing the application's size. The inclusion of the extra code resulted in a 15% increase in the total number of methods.

The results were the identical to those registered in *Table 12*, indicating that the ACM did not match any of the additional methods. This demonstrates that augmenting the application with additional code did not affect the ACM's ability to accurately identify matches.

4.6.2 Test Case 2

The final assessment conducted a form of negative testing to confirm that the ACM would not identify any matches when provided with two entirely different APKs. To carry out this test, the ACM was supplied with two APKs - one from an application, such as the open source FairEmail APK, and the other from a different application, for example, the obfuscated variant of the PCAPdroid APK.

This process was repeated four times with the following combinations of APKs:

(a) FairEmail Open source & OpenKeyChain Obfuscated

(b) OpenKeyChain Open source & PCAPdroid Obfuscated

(c) PCAPdroid Open source & TrackerControl Obfuscated

(d) TrackerControl Open source & FairEmail Obfuscated

In all instances, the ACM application did not identify any matches. This result can be attributed to the methodology employed in k-NN Analysis I in that, when there are no unique composite strings that are common between the APKs, the matching process is not initiated. These results confirm that the ACM accurately distinguishes between APKs that are not similar in any way.

5 Conclusion

The development and testing of the ACM application have successfully met all four project objectives set out at the beginning of this dissertation.

To address the first objective, four applications, along with their obfuscated and open source APKs were acquired for development and testing purposes. As outlined in the *Preliminary Stage*, this involved obtaining the obfuscated APKs from F-Droid and building the open source APKs using the applications' respective GitHub code repositories.

The *Design section* provides a detailed discussion regarding the implementation of the ACM application, thus achieving the second objective.

Furthermore, within the *Design section*, experiments were conducted to test various combinations of filtering techniques, with the purpose of finding the most effective approach for identifying and eliminating low-confidence matches, thereby fulfilling the third objective.

Finally, the fourth objective pertained to the assessment of the final matches generated by the ACM application. This was realised by establishing a ground truth and evaluating the accuracy, coverage, and processing time of the ACM application across the four applications.

5.1 Limitations

Recognising the limitations of the ACM is important for understanding its applicability and its potential constraints. These limitations can be broadly classified into scope limitations, which define the boundaries of what was explored, and design limitations, which concern the constraints inherent in the methodology adopted for the ACM application.

5.1.1 Scope Limitations

Obfuscation

The project considers types of obfuscation categorised by code clones Type I, II and III, as outlined in the Background section. However, another clone type, Type IV, characterised by differing code texts but identical functionality, was not addressed. Consequently, the ACM's effectiveness in identifying matches which employ this form of obfuscation remains unknown.

5.1.2 Design Limitations

Identifying Strings

As detailed in the Design Section, the ACM initiates its matching process by identifying unique composite strings within the APKs. While empirical evidence suggests that this approach is successful, the absence of initialised strings in an application would lead to failure in the matching process. In such cases, the ACM would be unable to initiate the matching process.

Matching Strings

The successful identification and matching of unique composite strings between APKs is crucial for the ACM's operation. However, the assumption that identical unique composite strings imply method correspondence, while supported by empirical evidence, carries a risk of failure. Although the likelihood of unrelated methods sharing the same unique composite string is low, acknowledging this possibility is important.

Class Membership

Following the unique composite string matching, the ACM employs k-NN analysis within the class boundaries. This is based on the assumption that class membership is not affected by obfuscation. While evaluation revealed no issues with this assumption, the potential for obfuscation to alter method class memberships cannot be discounted. Although, this phenomenon was not observed during evaluation, it is still worth acknowledging the possibility of it happening.

5.2 Future Work

The work undertaken in this dissertation can be extended in order to broaden its scope and utility.

5.2.1 Support for Multiple Programming Languages

As the development and testing of the project was carried out solely on Java applications, extending the project to support other programming languages commonly used in Android app development, such as Kotlin, could enhance its versatility and applicability in diverse development environments.

5.2.2 Comparison of Multiple APKs

Currently, the system allows for pairwise analysis and matching of two APKs. To enhance scalability, it would be beneficial to accommodate for the comparison of multiple APKs simultaneously. This feature would be particularly valuable in scenarios where it is necessary to compare one APK to several candidate APKs such as in areas of plagiarism detection, the identification of open source code and malware analysis.

5.2.3 Expand Input File Support

Presently, the ACM application exclusively supports APK archives as input. Expanding its capabilities to handle additional file types, such as iOs AppStore Package (IPA) files, equivalent to APKs for iOS applications, would broaden its utility and cater to a wider range of application development ecosystems.

5.2.4 Improvement of User Interface

The current implementation lacks a user-friendly interface, which could prove challenging for non-technical users. Investing resources into designing and implementing a more intuitive and accessible user interface would improve user-experience, making the project more widely accessible to users with varying levels of technical expertise.

5.3 Closing Remarks

This paper presents a novel approach for identifying similar methods between open source and obfuscated Java Android applications. Based on the review of existing literature, it becomes clear that employing unique strings for class method matching and using class boundaries and program call hierarchy to define the scope of the k-NN analysis constitutes an innovative approach.

The ACM application was successful in correctly matching class methods and covering a large footprint of the APKs. However, like any research endeavour, there are areas for improvement and inherent limitations. Nevertheless, the dissertation has demonstrated the ACM application's capability in identifying and matching class methods between APKs - a task that can be applied to code versioning, plagiarism detection, and addressing ethical considerations pertaining to open source code usage.

Bibliography

[1] Copilot. (2024). *Copilot* [online]. Available from: https://github.com/features/copilot [Viewed 29.03.2024].

[2] S. Carter, R.J. Frank, and D.S.W Tansley. (1993). Clone Detection in Telecommunications Software Systems: A Neural Net Approach. In *Proceedings of International Workshop on Application of Neural Networks to Telecommunications*, pp. 273–287.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9), pp. 577-591.

[4] N. Davey, P. Barson, S. Field, R.J. Frank, D.S. Tansley. (1995). In *Proceedings The development of a software clone detector*.

[5] Fabien Viertel, Wasja Brunotte, Daniel Strüber, Kurt Schneider. (2019). Detecting Security Vulnerabilities using Clone Detection and Community Knowledge. In *Proceedings International Conference on Software Engineering and Knowledge Engineering*.

[6] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. (2002). CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28, pp. 654-670.

[7] Lutz Prechelt, Guido Malpohl, and Michael Phlippsen. (2003). Jplag: Finding plagiarisms among a set of programs. *Journal of Universal Computer Science*, 8.

[8] Wise, M.J. (1993). String Similarity via Greedy String Tiling and Running Karp-Rabin Matching. *Unpublished Basser Department of Computer Science Report*.

[9] Z. Li, S. Lu, S. Myagmar and Y. Zhou. (2006). CP-Miner: finding copy-paste and related bugs in large-scale software code. IEEE *Transactions on Software Engineering*, 32(3), pp. 176-192.

[10] Rakesh Agrawal, Srikant Ramakrishnan. (1995). Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, pp. 3-14.

[11] Purushothama Raju v, Saradhi G.P. (2015). Mining Closed Sequential Patterns in Large Sequence Databases. *International Journal of Database Management Systems*, 7, pp. 29-39.

[12] FreeBSD. (2024). *The FreeBSD Project* [online]. Available from: https://www.freebsd.org/ [Viewed 29.03.2024].

[13] C. Kustanto, I. Liem. (2009). Automatic Source Code Plagiarism Detection. In *Proceedings of 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, pp. 481-486.

[14] Wise, M.J. (1993). String Similarity via Greedy String Tiling and Running Karp-Rabin Matching. *Unpublished Basser Department of Computer Science Report*.

[15] H. Murakami, K. Hotta, Y. Higo, H. Igaki and S. Kusumoto. (2013). Gapped code clone detection with lightweight source code analysis. *IEEE International Conference on Program Comprehension*, pp.93-102.

[16] G. Cosma and M. Joy. (2012*).* An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis. *IEEE Transactions on Computers*, 61(3), pp. 379-394.

[18] Keith D. Cooper, Linda Torczon. (2012). *Engineering a Compiler*, 2nd ed. Morgan Kaufmann, pp. 221-268.

[17] P. Nakov. (2000). Latent Semantic Analysis of Textual Data. In *Proceedings Conference Computer Systems and Technologies,* pp. 5031-5035.

[19] Tobias Sager, & Abraham Bernstein, Martin Pinzger, Christoph Kiefer. (2006). Detecting similar Java classes using tree algorithms. In *Proceedings of International Workshop on Mining Software Repositories*, pp. 65-71.

[20] Eclipse. (2024). *Eclipse ASTParser* [online]. Available from: https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTParser.html [Viewed 29.03.2024].

[21] Serge Demeyer, S. Tichelaar, Stéphane Ducasse. (2001). *FAMIX 2. 1-the FAMOOS information exchange model*.

[22] L. Jiang, G. Misherghi, Z. Su and S. Glondu. (2007). DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones*. 29th International Conference on Software Engineering*, pp. 96-105.

[23] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. (2004). Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry,* pp. 253–262.

[24] Ira Baxter, Andrew Yahin, Leonardo de Moura, Marcelo Sant'Anna, Lorraine Bier. (1998). Clone Detection Using Abstract Syntax Trees. In *Proceedings of International Conference on Software Maintenance*, pp. 368-377.

[25] Susan Horwitz, Thomas Reps. (1992). The use of program dependence graphs in software engineering. In *Proceedings of the 14th international conference on Software engineering*, pp. 392–411.

[26] Jens Krinke. (2001). Identifying Similar Code with Program Dependence Graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pp. 301-309.

[27] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, M. Bernstein. (1996). Pattern Matching for Clone and Concept Detection*. Automated Software Engineering*, 3, pp. 77-108.

[28] S. Cesare, Y. Xiang, J. Zhang. (2013). Clonewise – Detecting Package-Level Clones Using Machine Learning. In: T. Zia, A. Zomaya, V. Varadharajan, M. Mao, eds. *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer.

[29] John, G.H., P. Langley. (1995). Estimating continuous distributions in Bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pp. 338-345.

[30] S.L Salzberg. (1994). Book Review: C4.5: Programs for Machine Learning by J. Ross Quinlan. *Machine Learning*, 16, pp. 235–240.

[31] L. Breiman. (2001). Random Forests. *Machine Learning,* 45, pp. 5 – 32.

[32] Smit Patel, Roopak Sinha. (2022). Combining Holistic Source Code Representation with Siamese Neural Networks for Detecting Code Clones. In: Inmaculada Medina-Bulo, Mercedes G. Merayo, Robert Hierons, *Testing Software and Systems*. 30th IFIP WG 6.1 International Conference, pp. 148-159.

[33] Davide Chicco. (2020). Siamese Neural Networks: An Overview. *Methods in molecular biology*, 2190, pp. 73-94.

[34] S. Bezawada, Charanjitha, V. Reddy, Naveena, V. Gupta. (2016). Word2Vec. *Natural Language Engineering*, 23, pp. 155 - 162.

[35] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, S. Jaiswal. (2017) *graph2vec: Learning Distributed Representations of Graphs*.

[36] Cyrille Artho, Armin Biere. (2005). Combined Static and Dynamic Analysis*. Electronic Notes in Theoretical Computer Science*, 131, pp. 3-14.

[37] JNuke. (2024). *A program analysis framework for Java* [online]. Available from: https://fmv.jku.at/jnuke/#:~:text=written%20in%20C.,JNuke%20is%20a%20collection%20of%20program%20analysis%20tools%20for%20Java,and%20bytecode%20analysis%2Finstrumentation%20framework. [Viewed 29.03.2024].

[38] Bas Cornelissen, Andy Zaidman, Arie Deursen, Leon Moonen, Rainer Koschke. (2009). A Systematic Survey of Program Comprehension through Dynamic Analysis*. IEEE Transactions on Software Engineering,* 35, pp.684 - 702.

[39] Jadx. (2024). *Jadx* [online]. Available from: https://github.com/skylot/jadx. [Viewed 15.03.2024].

[40] F-Droid. 2024. *F-Droid Apps* [online]. Available from: https://f-droid.org/en/packages/org.fdroid.fdroid/ [Viewed 07.03.2024].

[41] Androguard. (2024). *Androguard Read the Docs* [online]. Available from: https://androguard.readthedocs.io/en/latest/. [Viewed 15.03.2024].

[42] Mahmoud Parsian. (2015). K-Nearest Neighbors. *Data Algorithms*. O'Reilly Media, Inc., pp.305-308.

[43] Ben Stegner. (2023). *What Is an APK File and What Does It Do? Explained* [online]. Available from: https://www.makeuseof.com/tag/what-is-apk-file/ [Viewed 07.03.2024].

[44] GuardSquare. (2024). *ProGuard* [online]. Available from: https://www.guardsquare.com/proguard. [Viewed 15.03.2024].

[45] Marcel Bokhorst. *FairEmail* [online]. Available from: https://github.com/M66B/FairEmail. [Viewed 30.03.24].

[46] scikit-learn. (2024). *MultiLabelBinarizer* [online]. Available from: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MultiLabelBinarizer.html. [Viewed 15.03.2024].

[47] scikit-learn. (2024). *KNeighborsClassifier* [online]. Available from: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html. [Viewed 15.03.2024].

[48] Alfeilat, Hassanat, Lasassmeh, Tarawneh, Alhasanat, Eyal-Salman, Prasath. (2019). Effects of Distance Measure Choice on K-Nearest Neighbor Classifier Performance: A Review. *Big Data*, 7.

[49] Biehl, Hammer, Villmann. (2014). Distance Measures for Prototype Based Classification. In *Proceedings International Workshop on Brain-Inspired Computing*, pp. 100-116.

[50] Steven Burrows, S.M.M. Tahaghoghi, Justin Zobel. (2004). Efficient and Effective Plagiarism Detection for Large Code Repositories. In *Proceedings of the Second Australian Undergraduate Students' Computing Conference*, pp. 9-16.

[51] Mariam Kouli, Abbas Rasoolzadegan. (2022). A Feature-Based Method for Detecting Design Patterns in Source Code. *Symmetry*, 14, pp. 1491.

[52] OpenKeyChain. *OpenKeyChain* [online]. Available from: https://github.com/open-keychain/open-keychain. [Viewed 29.03.2024].

[53] Emanuele Faranda. *PCAPdroid* [online]. Available from: https://github.com/emanuele-f/PCAPdroid. [Viewed 29.03.2024].

[54] Konrad Kollnig, Nigel Shadbolt. (2022). TrackerControl: Transparency and Choice around App Tracking. *Journal of Open Source Software*, 7(75), pp. 4270.