Received 9 October 2023, accepted 1 January 2024, date of publication 9 January 2024, date of current version 17 January 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3351945

RESEARCH ARTICLE

Android Authorship Attribution Using Source Code-Based Features

EMRE AYDOGAN AND SEVIL SEN
Wireless Networks and Intelligent Secure Systems (WISE) Laboratory, Department of Computer Engineering, Hacettepe University, 06800 Ankara, Turkey
Corresponding author: Emre Aydogan (emreaydogan@cs.hacettepe.edu.tr)

ABSTRACT With the widespread use of mobile devices, Android has become the most popular operating system, and new applications being uploaded to the Android market every day. However, due to the ease of modifying and repackaging Android binaries, Android applications can easily be modified and imitated by other developers and released in third-party Android markets. Therefore, determining the original developers of Android applications is a challenging problem known as authorship attribution. This study explores the distinctive features of Android applications to identify their authors. Software developers generally leave a footprint that reflects their writing styles in their applications. Therefore, this footprint, which can be extracted from either the source code or the binary code, can help identify the authors of software applications. Since obtaining the source code of applications in the wild can be impractical, especially when dealing with malware, researchers prefer to focus on the binaries of applications. Therefore, this study proposes an approach that identifies Android developers by deriving a wide range of features from different parts of Android applications, such as smali files, libraries, manifest files, and metadata information. Moreover, other features such as configuration, dex code, resource-based, and string-related features are inherited from other studies in Android authorship attribution and fused with the proposed feature set. The proposed approach was evaluated on benign and malware datasets and compared with those of other studies. The results show that the proposed features increase the accuracy by showing 82.5% and 95.6% in the market and malware datasets, respectively. The results demonstrate the positive impact of the proposed features on Android authorship attribution.

INDEX TERMS Android, authorship attribution, mobile malware, metadata, obfuscation, source code-based.

I. INTRODUCTION

Authorship attribution (AA) aims to identify the author of a computer program. Although it is primarily used to detect software theft and solve copyright issues, it is also used in digital forensics and malware analysis. Software developers generally leave footprints on their applications that describe their writing style. Consequently, there is a strong correlation between applications developed by the same developer, and the programmer style is preserved in program binaries [1].

In the literature, these footprints were first used to identify the software application authors. Owing to the emergence of software variants developed by making changes to the original software, researchers have shifted their focus to automatically identifying software theft and copyright issues. One of AA's promising applications is identifying malware authors, as there has been a significant increase in the number of new malware and new variations of existing malware [2]. AA can be used to identify the developers of new malware; therefore, it can help observe the evolution process of malware in the wild by monitoring the developers. Similarly, it can also be used to examine new programs developed by known attackers; hence, these programs can be analyzed further. On Android, developers could release their applications in the official and/or third-party markets. Since a developer might use different names to distribute their applications across various markets, AA becomes particularly important in such cases.

There are two main approaches to AA: source code AA and binary code AA. Most studies in the literature are

The associate editor coordinating the review of this manuscript and approving it for publication was Hui Liu.

based on the source code AA because some salient features extracted from the source code are lost when compilation is completed [2]. However, we may not always obtain the source codes of applications, especially malicious ones. Therefore, in such cases, an application's binary code is the only source that can be used to identify the application's author. Nevertheless, the binary code AA is much more challenging than identifying authors based on the source code. Compilers can change the structure of binary code using optimization techniques. For example, they could remove some valuable features from the perspective of AA, such as linguistics and formatting. Furthermore, because various compilers are used, most studies depend on specific compilers and compilation settings [3]. Despite these challenges, we must rely on binary data for the AA of malware due to the unavailability of the source code. Moreover, the performance of the popular techniques in author attribution, namely machine learning and deep learning, is mainly depend on the size of the training data. Since binaries of applications are readily available, binary AA is a study area worth investigating.

This study investigated the distinguishing features of Android applications for AA. New features were extracted from the applications (smali files, libraries and manifest files) as well as their metadata information in the market. Here, these new features and the existing features used in previous studies [4], [5], [6] are analyzed. Due to the availability of smali files, which are easily extracted from Android binaries, Android allows us to apply some source code-based features used in the literature to Android binary applications. Smali is an intermediate language such as Assembly, but it preserves some insightful source code features such as variable and function names and code organization. Moreover, it is more human-readable than assembly code. This study also investigated the impact of different feature groups on AA. Since the features proposed in the literature are analyzed with different datasets, a common experimental setting was first established for a fair comparison. Our dataset contains benign, malicious, and obfuscated applications from various Android markets and studies, as presented in Section VI. The results indicate that the source code-based features proposed in this study increase the accuracy of AA for both benign and malware datasets.

The contributions of this study are summarized as follows.

- New features extracted from smali files, third-party libraries (TPL) used in applications, and metadata of applications have been introduced for Android authorship attribution (AAA). To the best of our knowledge, the features inherited from the source code AA in smali codes were first used in Android in this study. The results showed that, source code-based features increase the accuracy of attributing applications to their corresponding authors.
- Unlike traditional application distribution mechanisms, Android applications are centrally distributed in mobile markets. Therefore, in addition to the application code, metadata about Android applications, such as application descriptions and user reviews, can also be obtained. Consequently, this study explored the impact of textual data on AAA. A positive effect of metadata-based features was observed, especially when combined with the source code-based features employed in this study.
- The performance of the proposed approach was extensively analyzed. In addition to analyzing the effect of each feature set on AAA, the studies proposed in the literature were compared. The experimental results demonstrate the positive impact of the features proposed in this study on the accuracy.
- Different application characteristics, including code similarity in new versions, obfuscation, and third-party libraries, have also been explored from an AA viewpoint.
- We shared the source code of our study with the research community at:
  https://github.com/emreaydogan/
  SourceCodeBasedAAA

The rest of this paper is organized as follows: Related work is discussed in Section II. Background information regarding the Android environment is provided in Section III. The proposed model is presented in Section IV. Section V discusses the feature sets used in our study and the literature. The datasets used in this study are described in Section VI. Experimental results and their discussion are presented in Section VII. Threats to the validity of the proposed approach are discussed in Section VIII. Finally, Section IX concludes the paper.

II. RELATED WORK

Authorship attribution is related to many different areas, such as literary work analysis, code plagiarism detection, biometric research, code authorship attribution, and malware analysis. It has different objectives such as authorship identification, clustering, evolution, profiling, and verification [7]. However, researchers have begun to find application authors in recent years. While studies based on the source code of applications have previously appeared in the literature, binary authorship attribution started to emerge due to the lack of and difficulty in gathering source codes. One of the first studies on binary authorship attribution was proposed by Rosenblum et al. [1], a novel program representation and technique that automatically detects the stylistic features of binary code written in C or C++ and can be compiled using GCC 4.5. They extracted syntax and semantic-based features such as idioms, n-grams, and graphlets. They claimed that the top-ranked features are unique to each author and reflect how the authors wrote the code. Alrabaee et al. [8] stated that the unique features of each author are unrelated to their programming style, in contrast to Rosenblum's [1] findings. They removed unrelated codes in their work and compared their results with Rosenblum's in terms of accuracy and false positive rate. They stated that their results were more accurate than Rosenblum's. Caliskan et al. [9] extracted syntactical features located in the source code of decompiled binary code, rather than the pure source code of desktop executable software, similar to our work on Android. They claimed that their approach was robust against basic obfuscation techniques and compared their results with those reported by Rosenblum et al.

Most studies, such as those mentioned above, have mainly used desktop and benign applications. However, malware analysts are also keen to find the original author of malicious applications. Kalgutkar et al. [7] summarized various methods for code authorship attribution, mainly from the perspective of malware domains. Attackers often employ evasion techniques such as encryption and obfuscation to hide their malicious intentions, which introduces new challenges to the AA problem. Only a few studies have been published on the identification of malicious authors. Layton and Azab [10] analyzed the source code of the Zeus botnet to determine the number of authors involved in developing malware and their roles. The evolution of the Zeus botnet over time was also analyzed in this study. In [11], Internet Relay Chat (IRC) messages were analyzed to match them with known aliases, which are real users hiding behind aliases. The motivation for this study is that IRC messages are often used for cybercrime, including online rooms selling stolen credit card details, botnet access, and malware.

Alazab et al. [12] proposed a model based on the authorship clustering. Initially, a set of spam emails represented by n-grams was divided into clusters using the Unsupervised Automated Natural Cluster Ensemble (NUANCE) method. These clusters were then examined using linguistic, structural, and syntactic features. This information can be used to create new forms of profiles for spam detection or to help analysts in investigating the size and scope of the operations behind spam attacks.

Alrabaee et al. [2] compared existing studies [1], [8], [13] on benign binaries and conducted an experiment to investigate the effect of their models on real malware. The study by [13] achieved better result in terms of accuracy than the other two studies in a benign dataset. The standard k-means clustering algorithm was used to compare the three studies in a malware dataset. The study given in [1] performed the worst among them, while other studies achieved similar results.

With mobile devices starting to affect a large area of our lives in the late 2000s, the need to carefully examine Android applications arose. Because our study aims to find the authors of both malicious and benign applications within the Android environment, we also explain related studies on Android AA. Furthermore, we have conducted a comparison of our newly proposed feature sets derived from the source code of the smali files, metadata, and third-party libraries against the feature sets used in the studies given below. These studies are explained in detail below.

A study focusing on string analysis of apk files was proposed by Kalgutkar et al. [4]. Three types of strings were employed: the string identifier list in the DEX file, all string components present in the DEX file, and strings extracted from the strings.xml file. Subsequently, n-gram analysis was employed on these string lists to generate a feature vector representing an author. Finally, the authors were classified using a Support Vector Machine (SVM). This approach was evaluated in different datasets that consisted of benign, malicious, and obfuscated applications, and accuracies of 98%, 96%, and 71% were obtained, respectively.

Gonzalez et al. [5] proposed an AA method based on array-related, array-unrelated, and n-gram features by retrieving bytecodes from the .dex file and converting them into a smali representation. Their framework consisted of two phases: profile construction and incremental analysis. In the first phase, author profiles were constructed using the Random Forest algorithm. In the second phase, incremental analysis is applied to attribute new applications to existing profiles and to find new possible profiles for applications that have not yet been attributed to the existing authors. They achieved 97.5% accuracy in a dataset with 33 authors and 1428 applications. They also applied their approach to over 131,000 applications collected from various sources to find applications belonging to author profiles in the wild.

Xu et al. [6] proposed a novel approach, AppAuth, that detects the original author of a group of given repackaged Android applications with common properties such as label names, icons, similar package names, and even file sizes. Several coding style-related features were extracted from the apk files: i) binary features from the .dex file; ii) resource features from decompiled files; and iii) configuration features from the manifest file. They successfully identified 69 original authors of 75 clone pairs (92%) in the wild. Additionally, They analyzed the impact of third-party libraries on their results, and stated that removing code-level features introduced by third-party libraries slightly improved the performance of their framework. They also evaluated the prediction performance of independent developers and development teams. However, main drawback of their study was that they compared only two repackaged (clone) applications by two different authors. If these two applications are clones of the same original application, they find an author between two plagiarists, even though the original author is different.

Wang et al. [14] extracted three types of string-related features: dex-based, manifest-based, and lib-based. In the dex files, identifier names, instruction sequences, and the use of Android APIs are collected. They extract the names of activities, providers, services, and broadcast receivers and use features from the manifest file. Finally, they obtained the names of the third-party libraries used in the applications. They compared the CountVectorizer, TFIDFVectorizer, and word2vec models to convert features into author profile vectors. Because the word2vec model yielded the best result, it was chosen for further evaluation. Subsequently, three machine learning models (i.e., Linear SVM, Random Forest, and Logistic Regression) were applied to these vectors to classify authors on different datasets containing benign, malware, and obfuscated applications. The results showed that their approach achieved 92.5% accuracy on average for the entire dataset. Moreover, it outperformed AppAuth in 2,900 non-obfuscated applications, thereby improving authorship identification by a 3.4%.

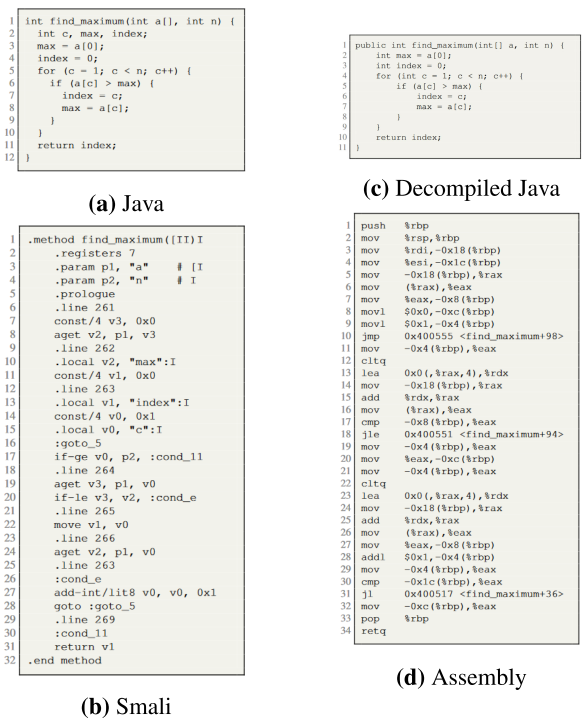
III. ANDROID BACKGROUND

Android, based on the Linux kernel, is the most widely used mobile operating system (OS), with a usage of 70.76% in October 2023 worldwide [15] and a cornerstone in the mobile industry. It offers an official market, Google Play [16], for developers to easily deploy their applications. Additionally, Android allows developers to upload their applications to alternative markets, such as Aptoide [17] and APKMirror [18]. Therefore, Android applications, namely *apk* files, can be easily gathered from such markets.

Alternative Android marketplaces like APKMirror and Aptoide are popular for offering a wider range of apps than the Google Play Store, including older versions of apps not available anywhere else. They appeal to users looking for specific features or applications that are not available in their region. However, while these markets expand options, they also bring risks. Unlike the Google Play Store, which has strict security controls, these alternative platforms may not have the same level of inspection. This may raise concerns about the security of some applications. Users who trust these sites generally need to be more careful and make sure that they download safe and trustworthy applications. Despite these concerns, these alternative markets remain valuable resources for those looking for unique or hard-to-find applications.

Android applications are written in Java or Kotlin programming languages. They are then compiled into Java bytecodes for the Java virtual machine. Finally, Java bytecodes are translated into Dalvik bytecodes and stored in .dex files. Dalvik bytecode corresponds to hexadecimal sequences of executables for Android; hence, it is the format that Android understands. However, Dalvik bytecode is difficult to read or modify. Hence, the smali intermediate language is generally used to look inside class files. Smali is a human-readable representation of the Dalvik bytecode and can be directly extracted from the apk files.

An Android application, an *apk* file, consists of several elements including dex files, a manifest file, resource files, and string files. All of these files can be used to gather information regarding the application. For example, while the manifest file contains information such as the number of services, activities, and permissions used in applications, we can find images, libraries, and XML files and folders used in applications in resource folders. String resources provide text strings for applications [19]. There are three types of strings: String, String Array, and Quantity Strings (Plurals). All strings were capable of applying styling markup and formatting arguments. The dex file contains Java/Kotlin class files. It is possible to convert *class* files to *smali* files, which are intermediate representations of dex files. Even though some linguistic and formatting features of the source code are lost when compilation is carried out, Dalvik-based executables, unlike assembly-based executables, contain some insightful information in the smali files. Fig. 1 shows a sample Java/C++ compatible code fragment, its corresponding smali, and assembly codes, respectively. While the smali code preserves information such as variable names and method signatures, the assembly code loses such information after the compilation process. Therefore, some features related to source code AA can be obtained from smali files. The impact of such features were explored for the first time in this study.



```
int find_maximum(int a[], int n) {
  int c, max, index;
  max = a[0];
  index = 0;
  for (c = 1; c < n; c++) {
    if (a[c] > max) {
      index = c;
      max = a[c];
    }
  }
  return index;
}
```

(a) Java

```
.method find_maximum([II)I
    .registers 7
    .param p1, "a"    # [I
    .param p2, "n"    # I
    .prologue
    .line 261
    const/4 v3, 0x0
    aget v2, p1, v3
    .line 262
    .local v2, "max":I
    const/4 v1, 0x0
    .line 263
    .local v1, "index":I
    const/4 v0, 0x1
    .local v0, "c":I
    :goto_5
    if-ge v0, p2, :cond_11
    .line 264
    aget v3, p1, v0
    if-le v3, v2, :cond_e
    .line 265
    move v1, v0
    .line 266
    aget v2, p1, v0
    .line 263
    :cond_e
    add-int/lit8 v0, v0, 0x1
    goto :goto_5
    .line 269
    :cond_11
    return v1
.end method
```

(b) Smali

```
public int find_maximum(int[] a, int n) {
    int max = a[0];
    int index = 0;
    for (int c = 1; c < n; c++) {
        if (a[c] > max) {
            index = c;
            max = a[c];
        }
    }
    return index;
}
```

(c) Decompiled Java

```
push   %rbp
mov    %rsp,%rbp
mov    %rdi,-0x18(%rbp)
mov    %esi,-0x1c(%rbp)
mov    -0x18(%rbp),%rax
mov    (%rax),%eax
mov    %eax,-0x8(%rbp)
movl   $0x0,-0xc(%rbp)
movl   $0x1,-0x4(%rbp)
jmp    0x400555 <find_maximum+98>
mov    -0x4(%rbp),%eax
cltq
lea    0x0(,%rax,4),%rdx
mov    -0x18(%rbp),%rax
add    %rdx,%rax
mov    (%rax),%eax
cmp    -0x8(%rbp),%eax
jle    0x400551 <find_maximum+94>
mov    -0x4(%rbp),%eax
mov    %eax,-0xc(%rbp)
mov    -0x4(%rbp),%eax
cltq
lea    0x0(,%rax,4),%rdx
mov    -0x18(%rbp),%rax
add    %rdx,%rax
mov    (%rax),%eax
mov    %eax,-0x8(%rbp)
addl   $0x1,-0x4(%rbp)
mov    -0x4(%rbp),%eax
cmp    -0x1c(%rbp),%eax
jl     0x400517 <find_maximum+36>
mov    -0xc(%rbp),%eax
pop    %rbp
retq
```

(d) Assembly

FIGURE 1. Sample code fragments.

IV. ANDROID AUTHORSHIP ATTRIBUTION

There are only a few studies [4], [5], [6], [14] on AA for Android. AppAuth [6] extracts distinguishing features for AA from the dex file, manifest file, and resource files of applications. In [4], they used all string information placed in the string XML files and dex files of applications. Unlike these studies, in this study, source code-based features extracted from smali files were used for the first time. In addition to the source code-based features, the effects of other proposed features (usage of permissions, third-party libraries, and metadata-based features) were analyzed in this study. All of these features are given as SPL, which stands for Source code-based features, Permissions, and third-party Libraries.

The proposed model is illustrated in Fig. 2. Initially, apk files were decompiled using apktool [20], and the smali files

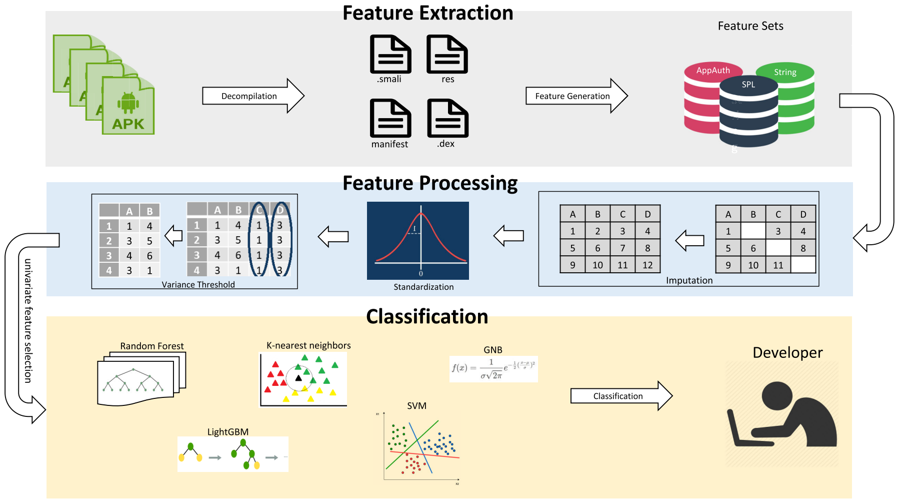


FIGURE 2. Overview of model.

were obtained. Subsequently, the String, AppAuth, and SPL features of the applications are collected from these files and the apk files themselves, and a fixed-size feature vector per application is assigned to the classifiers. SimpleImputer from the scikit-learn library was used to fill in the missing values in the feature vectors. After this imputation process, these features were standardized by removing the mean and scaling to the unit variance using StandardScale in the scikit-learn library.

A two-phase dimensional reduction is used before the classification algorithms are executed. First, the features with zero variance were eliminated from the dataset. Then, univariate feature selection was applied, selecting the best features based on the univariate statistical tests. The Random Forest (RF) classification algorithm [21] was applied using the scikit-learn library [22]. Then, stratified 10-fold cross-validation was employed. Because stratified 10-fold cross-validation ensures that the proportion of positive to negative examples from the original distribution is preserved in all folds, it is especially useful when the dataset is unbalanced [23]. Therefore, it was used five times, and the average of these five runs is presented in the results.

V. FEATURE SETS

The main objective of this study is to find effective features to solve the AA problem in Android. Therefore, new features based on source code-based AA extracted from smali files are presented. Furthermore, new features unique to the Android system have been analyzed, such as the usage of permissions and metadata-based features that have been increasingly used in Android security in recent years [24]. In addition to these features, the impact of third-party libraries on the problem were explored. It has been shown that many applications use a large number of third-party libraries [25] and on average, 60% of the code belongs to these libraries [26], [27]. As third-party libraries can create noise, they are shown to affect repackage and malware detection on Android [28]. This could also affect AAA because source code-based features were employed in this study. Therefore, here, the effects of extracting source code-based features only from custom code on the AAA are investigated. Moreover, the effects of third-party libraries as features are explored. Our proposed features are compared with those in the literature, which are based on binary AA [4], [6]. We did not include Gonzalez et al.'s [5] work in the comparison because AppAuth [6] also used the same features used in that study [5]. All features considered in this study are listed in Table 1, and their explanations are listed in Table 2. Features that are being explored for the first time in this study are highlighted in red. In the following sections, the features used in the literature for the problem are first presented, and the newly proposed features in this study are presented in detail.

TABLE 1. Features proposed for AAA in the literature.

Features	Abbr.	Our Study	AppAuth [6]	StringAA [4]
Configuration	conf	✓	✓	
Dex Code-Based	dex	✓	✓	
Resource	rsrc	✓	✓	
String-Based	str	✓		✓
Source Code-Based	src	✓		
Permission	perm	✓		
Third-Party Library	lib	✓		
Metadata	meta	✓		

A. FEATURES USED IN APPAUTH

AppAuth groups its features into three sets: Configuration, Resource, and Dex Code.

1) CONFIGURATION

Android applications consist of four components: activities, services, broadcast receivers, and content providers. Each component has its own purpose and life cycle, which define

TABLE 2. Feature set descriptions.

Category (# of features)	Description
Source Code-Based (18)	# of average character per line
	# of average character per local variable
	# of average character per global variable
	# of average character per function name
	# of average character per function parameter name
	Ratio of global variables to lines of code
	Ratio of local variables to lines of code
	# of average lines of code per function
	Ratio of variables to lines of code
	Ratio of if to all codes
	# of average lines of code per class or interface
	# of average number of functions per class or interface
	Ratio of invoke to all codes
	Ratio of move to all codes
	Percentage of function names starting with an uppercase letter
	Percentage of int function definitions to all
	Percentage of void function definitions to all
	Percentage of identifiers beginning with an uppercase character
Dex Code (14)	Ratio of abstract classes
	Ratio of classes containing annotations
	Ratio of direct methods
	Ratio of virtual methods
	Ratio of methods containing try and catch
	Ratio of methods containing debug information
	Ratio of static fields
	Ratio of classes containing interfaces
	Ratio of mathematical instructions to functional instructions
	Ratio of **aget** instructions to **aput** instructions
	Average of the length of all the arrays
	Median of the length of all the arrays
	Standard deviation of the length of all the arrays
	Ratio of arrays with constant length
Configuration (9)	Number of activity
	Number of service
	Number of receiver
	Number of provider
	Number of intent-filters
	Number of meta-data
	Number of uses-permission
	Number of sensitive uses-permission
	Number of uses-feature
Resource (11)	Number of directories in the **res** directory
	Number of directories whose name containing **drawable** in the **res** directory
	Number of files in the directories whose name containing **drawable**
	Number of directories whose name containing **layout** in the **res** directory
	Number of files in the directories whose name containing **layout**
	Number of directories whose name containing **values** in the **res** directory
	Number of files in the directories whose name containing **values**
	Number of files in the **assets** directory
	Number of files in the **lib** directory
	Number of .so files in the **assets/lib** directory
	Number of **xml** files in the **res** directory
Permission (158)	Android permissions
Library (500)	A whitelist of third-party libraries
Metadata (50,000)	Application descriptions placed in Android markets
String/n-gram (10,000)	Dex - Strings present in the DEX file
	Application - Strings extracted from the strings.xml file
	All strings (Dex + Application)

how it is created and destroyed. Activities define how users interact with applications. They are used to define user interfaces that allow users to use the system resources. Every single application screen refers to activities on Android. Some activities may trigger other activities. Services perform long-term operations in the background and do not contain a user interface. Broadcast receivers allow an application to be linked to a system or an application event. The Android operating system notifies the connected application when an event is triggered. Content providers are used to access data in the application databases. These are also used by an application to share its database with other applications. Thus, a single piece of content can be distributed across multiple applications. All the components are declared in the Android manifest file. Developers tend to reuse the same Android manifest file when developing their applications. Therefore, information in the manifest file, such as the number of components, can help distinguish developers from each other. Based on this assumption, AppAuth uses the configuration features collected from the manifest file, as listed in Table 2.

2) RESOURCE

Android applications may contain resource files such as images, sounds, icons, and native libraries stored in the *res* directory. These files can be used for various reasons, such as language support and the provision of images for UI. Some files in the *res* directory, like dynamic libraries, database files, and payload files, are accessible at runtime. Developers embed some specific and critical information into these files. Therefore, it is important to analyze the characteristics of this folder. Therefore, AppAuth uses features related to resource files, such as the number of files/folders in the *res* directory.

3) DEX CODE

The Dex file contains information about the structure of the applications, as shown in Fig. 3. AppAuth mainly analyzes the data section in .dex files and focuses on methods, classes, and field structures, as well as annotation, interface, debug information, and the like. Due to application sizes could vary significantly, they use ratio values instead of obtaining the numerical values of these features, such as the ratio of the number of abstract classes to all classes.

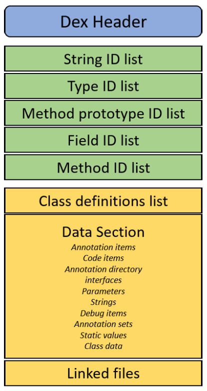


FIGURE 3. Format of a .dex file.

B. FEATURES USED IN STRINGAA

1) STRING ANALYSIS

In [4], features based on strings are used. The strings in the strings.xml file are extracted line by line as app string features. Strings in the strings.xml file contain references in the source code or other resource files of the applications. These are static strings shown to users, such as the application's name. The dex file consists of different parts, such as the string id list, type id list, and method id list. The string id list mainly contains the strings used in the source code of an application. However, malware developers embed their payloads in the string ID list to run them at runtime and avoid analysis. Strings in the string id list part of a dex file are extracted as dex string features.

In [4], preliminary experiments were first conducted to determine the optimal ‘n‘ value for n-gram analysis, which was selected to be 3. Therefore, in our study, 3-grams are also used for string-based features. Similarly, as done in [4], not lose a line that has one or two words, we insert a tag at the beginning and end of each string, ensuring that only empty lines are eliminated. As a result, line-bounded 3-grams were obtained for each string type. Table 3 shows the 3-gram features for the two different types of strings. Because of the massive amount of 3-grams extracted from our dataset, redundant 3-gram features were eliminated using the HashingVectorizer technique. As a result, the best 3-grams are selected.

TABLE 3. Sample 3-gram features.

Type	String	3-gram
app	Enable Google Play services	<LB>Enable Google Enable Google Play Google Play services Play services <LB>
dex	mContainer=	<LB>mContainer= <LB>

C. PROPOSED FEATURES

In this study, the effects of the proposed features on AAA were investigated. Four groups of feature were introduced: source code-based, permissions, third-party libraries, and metadata-based features. Apart from those related to the usage of third-party libraries, the features are unique to the Android environment.

1) SOURCE CODE-BASED

Source code-based features are extracted from smali files, which contain Dalvik-byte/smali codes generated from the Java bytecodes in class files. Smali codes can be considered as assembly codes for C/C++ programs. Both are generated from a high-level language and converted to machine code via Android Runtime (ART) for smali. The operating system then executes these machine codes to run the applications. In Fig. 1, smali and assembly codes corresponding to a Java function named *find_maximum* are shown. As seen in the figure, the names of the function parameters, global and local variables in the original source codes are not preserved in the assembly files but in the smali files. For example, the local variables *n, max, index* and function parameters *a, n* are preserved in the smali code given in the figure. It is shown that developers tend to use the same format in naming identifiers, such as using $ character or digits in variable names. Therefore, preserving such names in decompiled code can help identify developers. Hence, because the smali files are more similar to the original source codes, they were used to extract the source code-based features in this study.

The source code of the software can provide us with beneficial features for identifying the author of unknown software, one of which is the authors' coding style. An author can be identified by its coding style. On the one hand, some salient features, such as the usage of comments, loops, and braces in the source code, are lost during the transformation from source code to assembly code. On the other hand, as shown in [9], the coding style still remains in the binary code. Based on this assumption, we inherit some source code-based features listed in [29] because of the nature of the Dalvik bytecode, which preserves some stylistic features of AA. All 18 source code-based features used in our study are listed in Table 2. The features listed in [29] are primarily proposed for languages such as Java and C/C++; therefore, features compatible with smali codes from that list are selected. In the future, more source code-based features, such as comments, annotations, bracket position, indentation styles, etc., could be included using the original source code of applications.

Decompiled Java code can also be extracted from the apk files. To obtain Java files from apk files, dex files must first be generated from the apk files, and Java files must then be extracted from these dex files using decompiler tools such as jadx [30]. The same source code-based features extracted from smali files can also be extracted from the decompiled Java files. However, because the Java language does not contain some smali-specific instructions such as move and invoke, two features related to these instructions are not considered in Java codes. It is important to note that the transformation process from apk files to Java files is prone to errors, such as the inability to convert some classes, methods, or variables correctly. Due to such errors encountered in the conversion, we were able to extract features from Java codes from 80% of the samples in the market dataset. Note that only the source code-based features extracted from the decompiled Java files are used in Table 13 to show and compare the effects of the source code-based features in both the decompiled Java files and the smali files.

2) PERMISSION

Permissions are required to use the system resources of mobile devices. Applications can access and use device resources, such as cameras and GPS, only through the permissions requested in their manifest files. While normal permissions can be granted automatically, dangerous permissions require explicit user approval. Most applications request more permissions than they use. A study that evaluated the gap between the requested permissions and those used at runtime found that applications, on average, request 30% more permissions than they need [31]. Although a developer might develop applications belonging to different categories, they could reuse the same manifest file in these applications because of its convenience. Moreover, because of the use of the same third-party libraries in applications developed by the same author, they need to use the same permissions that these libraries require. While AppAuth [6] used the number of permissions in the manifest file as a single feature, in our study, the existence of each feature is considered as a separate feature. Therefore, 158 binary features corresponding to the 158 permissions are added to the feature set.

3) LIBRARY

Developers prefer using third-party libraries to implement certain functionalities in their applications, rather than implementing them from scratch. It is shown that most applications use more than 20 third-party libraries [32] and a large part of the application code belongs to such libraries [26], [27]. For the same functionality, developers tend to use the same library that they are familiar with. Therefore, third-party libraries are included in the feature dataset.

To extract library features, a whitelist of libraries was constructed. Initially, we downloaded all library names from the mvnrepository website [33]. We then try to determine which common libraries are used in both malware and benign datasets using the downloaded library name list. When an apk file is decompiled using apktool, all smali codes of the libraries used in the application are placed in the src directory according to their package name. For example, if the jsoup library is used, all smali codes related to jsoup are placed in the *src/org/jsoup* directory because the jsoup package name is "org.jsoup". Hence, we can find the directories of libraries used from their package names and exclude them. Thus, we could only obtain custom smali codes written by the author of the application. Some libraries may be obfuscated; therefore their names consist of arbitrary character sequences rather than meaningful words. We also eliminated these libraries by using basic heuristic approaches. Libraries used in very few applications were also not considered in this study. For each application, the existence of each library from our list was used as a feature. In a recently proposed study [14], third-party libraries were used for AAA. However, unlike our approach, the names of third-party libraries were used as string features in [14].

4) METADATA
Android applications are distributed centrally in mobile markets differently from traditional application distribution mechanisms. In addition to the application itself, these markets provide information about the application, such as application description, application rating, and user reviews, known as metadata. In this study, the application descriptions generated by the developers were analyzed for the problem, as this textual information might help solve AA. The same method applied to extract a string of applications was employed here. However, it is important to note that unlike textual data used for AA in the literature, application descriptions are short texts. For example, application descriptions on Google Play are limited to 4,000 characters.

VI. DATASET
In this study, two different datasets were constructed: a market and a malware dataset, consisting of benign and malicious applications, respectively. To construct the market dataset, a Web crawler was implemented using the Scrapy framework [34] to collect benign applications from various alternative markets. In addition, applications from other studies [4], [5], [35], [36] were included in this dataset to compare our approach with theirs. Each application must be signed with a developer certificate for installation on an Android device. Therefore, applications sharing the same signature are assigned to the same developer, and each application is grouped according to its related signature, as in [4]. The signatures were extracted from the apk files using the print-apk-signature tool [37].

The benign dataset contains binaries collected from different alternative Android markets, namely Apkpure [38], Apkmirror [18], Onemobile [39] and Aptoide [17] between January 2020 and August 2020. However, this dataset includes applications written much earlier than 2020 because of the availability of the early versions of contemporary applications. The distribution of the datasets over the years is presented in Table 4.

TABLE 4. Year distribution of dataset.

Year	Percentage (%)
2014	4.9
2015	5.4
2016	8.6
2017	39.7
2018	41.4

Initially, 200,000 applications were downloaded for our study. Then, developers with fewer than ten applications were eliminated because other studies that we use to compare our approach use developers with at least ten applications [4], [6]. Moreover, we must have sufficient applications for each developer to generate a effective model that can differentiate between developers. The malware dataset contains malicious binaries belonging to various malware families, namely Ransomware, Adware, and SMS [40], and some binaries from datasets introduced and used in security-related studies, namely Rmvdroid [41], Drebin [35], Genome [5] and Koodous [4]. The number of authors and applications collected from different repositories within our dataset is shown in Fig. 4. Owing to the elimination of authors with fewer than ten applications, the dataset consists mainly of applications that are mostly written by multiple developers or companies. Fig. 5 displays a histogram representing the distribution of APK sizes, measured in megabytes (MB), for both the market and malware datasets.

Some authors were eliminated during the feature extraction step due to errors. If an application exists in more than one dataset, only one of them is randomly chosen and kept in the dataset. However, if an application had multiple versions, they are all kept in the dataset. As a result, the dataset given in Table 5 was used in all experiments other than the version analysis, in which the effect of versions on AA was analyzed, and metadata analysis, in which features extracted from application metadata were analyzed on AA. Given that not all applications have metadata information or multiple versions, a subset of the original dataset was used for the version and metadata analysis.

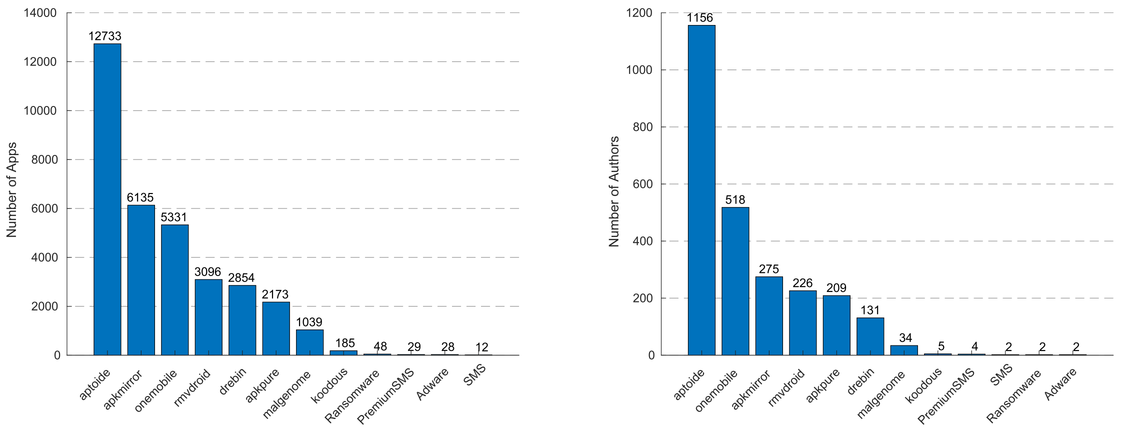


FIGURE 4. Summary of datasets.

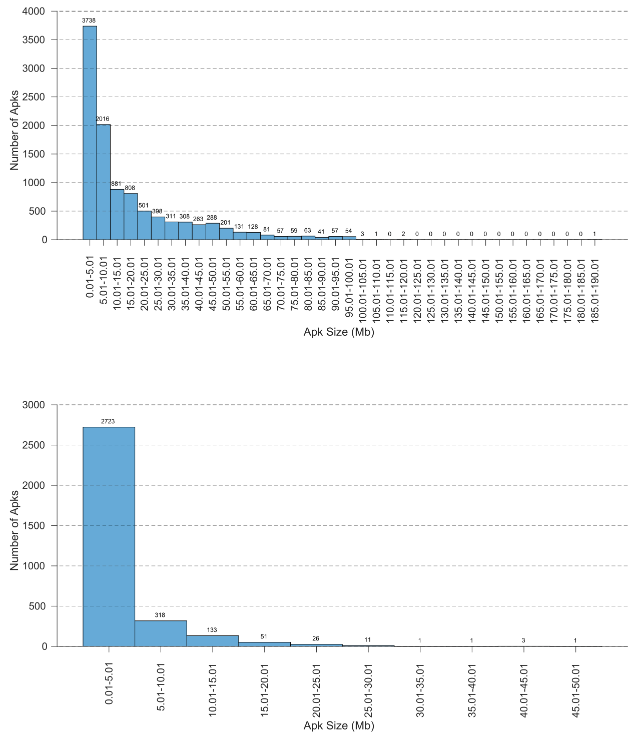


FIGURE 5. APK size distribution (market at the top, malware at the bottom).

TABLE 5. Dataset information.

Dataset	# of app	# of author
Market	10,385	488
Malware	3,268	153
Genome	1,530	39

VII. EVALUATION

In this study, we conducted a series of experiments to evaluate our framework and investigate the impact of the newly introduced features. The performance of our proposed approach is also explored for different types of applications, such as malicious, benign, and obfuscated ones. Various experiments were performed to evaluate the performance of the proposed approach. For each experiment, 10-fold cross-validation and five epochs were employed, and an average of 50 results were obtained. All experiments were run on a CentOS 7.7 server equipped with 128 GB RAM and Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz.

A. RESEARCH QUESTIONS

We aim to find answers to the following research questions (RQs) on how our model identifies the author of Android applications, including both benign and malicious ones:

- **RQ1** - *What is the performance of classification algorithms in solving the authorship attribution problem?*
- **RQ2** - *What is the ideal set size for n-gram features?*
- **RQ3** - *Does the use of TPLs by applications bring improvements in solving the authorship problem?*
- **RQ4** - *How effective is the proposed approach in identifying the authors of applications?*
- **RQ5** - *Does the metadata of applications help to attribute the developers of applications?*
- **RQ6** - *What are the most important features for attributing applications to their developers?*
- **RQ7** - *Can we reduce the number of features without decreasing the performance of the model?*
- **RQ8** - *Does the number of applications per developer affect classification performance?*
- **RQ9** - *Does the proposed model successfully identify different versions of applications developed by the same author?*
- **RQ10** - *What is the effect of obfuscation on AAA?*
- **RQ11** - *Are there any clone applications in the datasets? How do they affect the performance on AAA?*

1) RQ1 - PERFORMANCE OF CLASSIFICATION ALGORITHMS ON AAA

a: MOTIVATION

Machine learning algorithms may exhibit different performances for different problems. With this motivation, we aim to compare the performance of machine learning algorithms on the problem at hand so that the algorithm that shows the best performance can be used in subsequent experiments.

b: METHOD

The following classification algorithms were used in the experiments: Random Forest, K-Nearest Neighbors, Support Vector Machines, Gaussian Naive Bayes, and LightGBM. These supervised machine learning techniques are among the popular methods used for code authorship attribution in the literature [7]. To tune the hyperparameters of these algorithms in the market and malware datasets, the GridSearchCV function in scikit-learn is used. GridSearchCV performs an exhaustive search of the specified parameter values for an estimator. In this experiment, all the feature groups were used, including the features used in AppAuth [6] and String Analysis [4]. Only metadata features were excluded because some applications did not have descriptions. The algorithms were run on both the benign and malware datasets. Accuracy and f1-score were used as performance metrics because an efficient model should have good precision and high recall. We also calculated the classification time for each algorithm.

c: RESULT

The comparison results are shown in Table 6. In addition to the high accuracy and f1-scores, the Random Forest classification time is also more reasonable than that of other algorithms; therefore, the Random Forest algorithm was used in the subsequent experiments. In addition, the default parameters of RF produced a performance very similar to the results obtained by its optimal parameters. Although LightGBM performs better than Random Forest, its classification time is high. Moreover, the results were highly variable for the different parameters. However, owing to its high accuracy, the use of LightGBM is worth investigating in the future.

TABLE 6. Comparison of classification algorithms.

	Market			Malware		
	acc	f1	time(s)	acc	f1	time(s)
Random Forest	82.4%	80.3%	124.46	95.4%	94.5%	14.18
KNeighbors	64.8%	62.0%	76.30	82.6%	79.1%	9.40
Support Vector	51.3%	50.5%	549.83	79.7%	76.6%	26.09
Gaussian Naive Bayes	49.5%	47.6%	45.97	60.3%	59.0%	8.23
LightGBM	**86.4%**	**85.1%**	1563.35	**97.5%**	**97.0%**	370.81

Findings #1: *The results indicate that the Random Forest algorithm performs considerably better than other algorithms in terms of accuracy and time.*

2) RQ2 - THE IDEAL SET SIZE FOR N-GRAM

a: MOTIVATION

As the number of applications increases, the number of 3-grams to be processed will also increase; in our case, tens of millions of 3-grams in the market dataset, could create computational and storage problems. Therefore, finding the maximum number of n-grams we can obtain results will improve our model's performance.

b: METHOD

As suggested in [4], 3-grams were used here. A preliminary analysis was performed to determine the optimal value of n. They observed that their system performed better with an increase in the n-gram size from 1 to 3. They also found that the system performance degraded with a further increase in the size of the n-grams. Because we compared our approach with that in [4], we decided to use the same n value for n-grams. We then used a HashingVectorizer to restrict the number of n-grams used in the experiments to reduce the memory consumption.

c: RESULT

The effects of the number of 3-grams on the classification accuracy and training time are shown in Fig. 6a and 6b, respectively. As clearly shown in the figure, the training time increased exponentially when more than 10,000 3-grams are used in the classification. However, this does not result in a significant improvement from an accuracy point of perspective. Please note that owing to its size and generation of a massive amount of 3-grams, the market dataset could be run with 50,000 3-grams at most.

Finding #2: *The best 10,000 3-grams yield sufficiently good accuracy in a reasonable training time and hence were employed in the subsequent experiments.*

3) RQ3 - CUSTOM CODE VS ALL CODE INCLUDING TPLS

a: MOTIVATION

Developers have certain coding habits and tend to use the same coding style in their codes, such as using *while* loop over *for* loop, using object-oriented modularization over procedural programming [42]. These characteristics are preserved in the custom code. Third-party libraries are also widely used when developing applications. Therefore, although the custom code shows the characteristics of the developer, the TPL code might create noise in the AA. However, developers tend to use the same TPLs in their applications, which may positively affect AA. They usually do not update the version of TPLs, even in the case of security vulnerabilities in the older versions they used [43]. Therefore, the impact of the TPL code should be investigated.

b: METHOD

two settings were used to demonstrate the effect of TPLs on AA. In the first setting, the source code-based features were extracted only from the custom code implemented by the developers. In the second setting, the features are extracted from the same applications, but the TPLs are included (**all** codes).

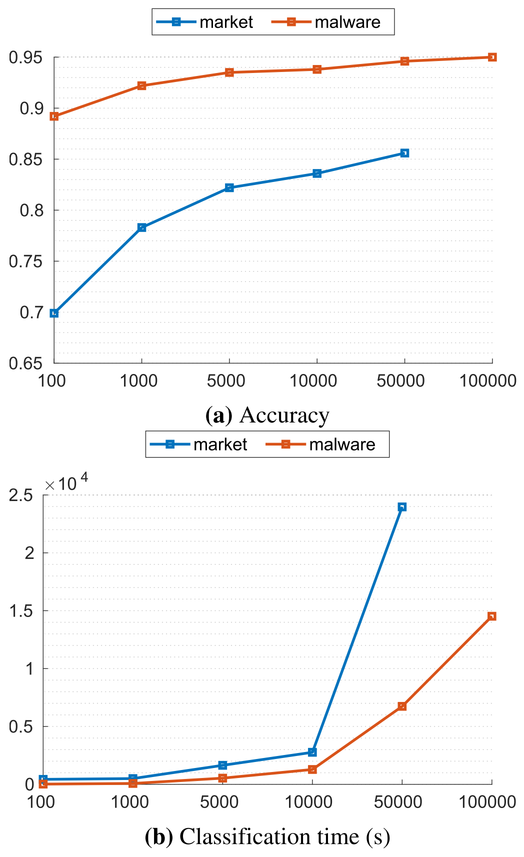


FIGURE 6. **Effect of the number of 3-grams.**

c: RESULT #1

As shown in the results in Table 7, including TPLs when extracting source code-based features yields a much better performance (custom src vs. all src). Then, the custom code was enriched with other features presented in Table 1 and compared with all code in Table 7. The results clearly show that, rather than extracting source code-based features from TPLs, the existence of TPLs is sufficient for AA and produces much better results than when including the TPLs' code. Therefore, in the subsequent experiments given below, the source code-based features were only extracted from the custom code and were used with the library features (custom src+lib).

d: RESULT #2

The confusion matrix for the developers is shown in Fig. 7. Authors with at least 40 applications in the market dataset were shown to fit the matrix. As seen in the figure, some applications of developer auth9 are wrongly matched to developer auth11. We further analyzed these authors using SimiDroid [44], which reveals the similarity between mismatched applications of developer auth9 and all applications of developer auth11. Third-party libraries were removed first. The results show a high similarity between such applications owing to the SmaliHook library, which is not eliminated because it is not on our TPL list. When the feature vectors of such applications were analyzed using cosine similarity, very high similarities (98%) were observed. Because these overlooked third-party libraries affect the source code-based features, they severely affect the results. This inference shows that we must expand the third-party libraries used in the library feature set to improve the performance of our model. As stated above, in addition to the whitelist approach, machine learning-based approaches could be employed to detect more libraries, including obfuscated ones, in the future.

Findings #3: *TPLs in applications increase the accuracy of authorship attribution. However, source code-based features do not need to be extracted from TPLs, which decreases the time required for feature extraction. Only the existence of TPLs, together with the source code-based features extracted from custom code, produces sufficiently good results.*

4) RQ4 - EFFECTIVENESS OF THE PROPOSED APPROACH

a: MOTIVATION

One of the main objectives of this study was to identify effective features of AAA. Therefore, the proposed features were compared with those of other proposed feature sets [4], [6] in the literature. Therefore, the effectiveness of the proposed approach in identifying the authors of applications is given comparatively.

b: METHOD

After determining the experimental settings in the previous research questions, the newly proposed feature sets, except metadata (src+perm+lib, or SPL), were compared with other feature sets used for AAA in the literature [4], [6]. For this purpose, we re-implemented the code to extract the string-related features given in [4]. However, thanks to the authors of AppAuth [6], they share their source code to extract the features listed in AppAuth [6]. Therefore, the same code was applied to extract AppAuth features.

c: RESULT #1

The comparison results are presented in Table 8. As shown in the results, the newly proposed features perform better than AppAuth, whereas the proposed features produce less accuracy than string analysis [4]. However, as stated above, extracting all 3-gram features may not be applicable from a classification time point of view. Therefore, only 10,000 3-grams are employed here. However, as a relatively small dataset, all 3-grams were extracted from the Genome dataset. Moreover, it is the dataset used for evaluation in [4]. In the Genome dataset, the proposed features showed slightly better performance than string-related features. When all the features are included, the best performance is obtained in the market and malware datasets. We also calculated the time required to extract each set of features. Table 9 shows that n-gram features require at least one and a half

TABLE 7. Accuracy results of custom source code enriched with other feature groups.

Dataset	Custom Src	Custom Src+Lib	All Src	Custom SPL	All Src +Perm	Custom SPL +AppAuth	All Src+Perm +AppAuth
Market	42.3%	70.9%	66.8%	75.1%	71.8%	**78.7%**	76.1%
Malware	80.5%	87.1%	87.5%	91.5%	91.1%	**93.4%**	93.3%
Genome	80.7%	97.1%	95.0%	97.9%	96.4%	**98.0%**	97.4%

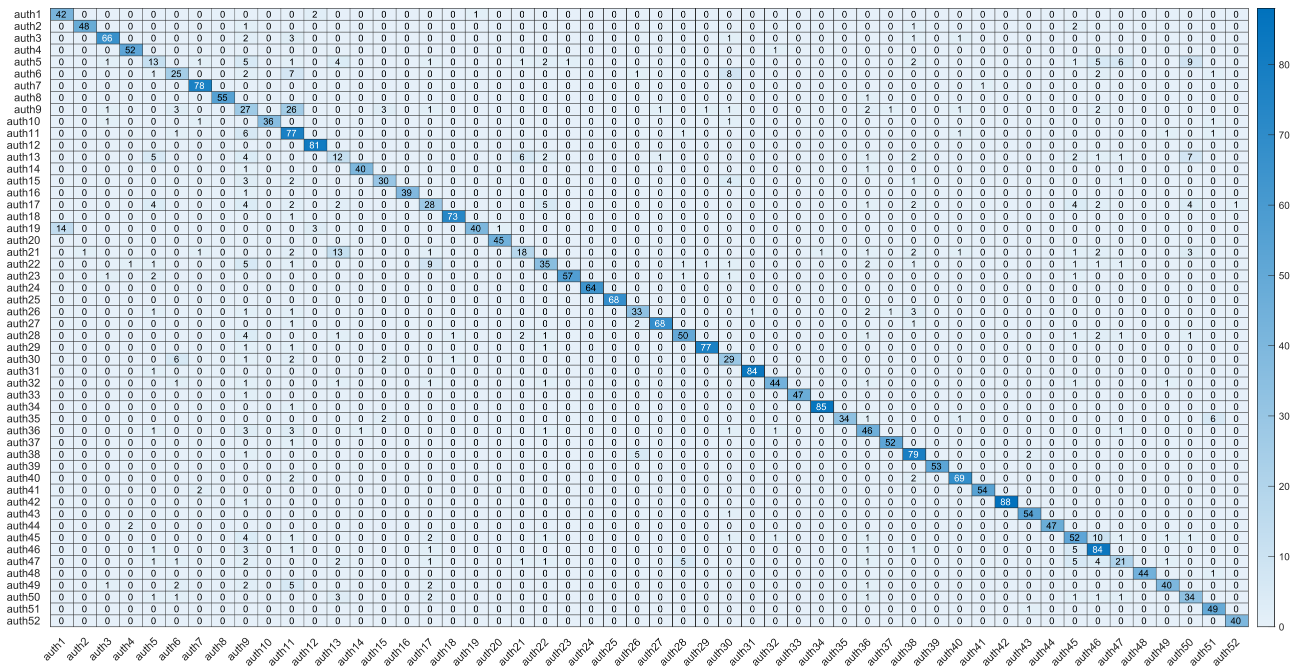


FIGURE 7. Confusion Matrix for the 52 developers who have at least 40 applications.

times more time than other feature sets. The extraction times of SPL and AppAuth features were relatively close. Although a feature extractor should be performed only once, the feature extraction process, when generating a considerable volume of n-grams, can be notably memory-intensive. This can result in memory-related challenges, particularly for systems with limited RAM availability. In [45], it was highlighted that a primary limitation of the n-gram approach is its tendency for exponential n-gram growth as the text size increases. This complexity often renders the method unviable for systems with limited computational capabilities.

d: RESULT #2

In this study, the effect of each feature group on AAA was also explored. The results are presented in Table 10. In addition to the n-grams in the code, the source code-based and resource features showed the highest accuracy. Because n-grams are extracted from strings, such as the names of variables/methods in the application code, they can include unique identifiers that distinguish developers. However, as shown in Table 9, the time required to extract and process such features was considerably high. Moreover, such features might not be resilient to obfuscation techniques such as renaming and encryption.

In general, all feature sets other than library features produce higher accuracy on the malware dataset, which is a smaller dataset. The average number of libraries per application varied significantly between the two datasets, as shown in Table 11. While applications collected from the market include 9.37 third-party libraries on average, this number is only 2.72 for malware applications. This may be the result of the use of obfuscation in malicious applications. In this study, a whitelist approach was used to extract TPLs used in the Apk package. Common libraries used in the market and malware datasets were selected and used as features. However, because of obfuscation, the names of some libraries are simply random words, and such libraries are eliminated in the whitelist approach. In the future, recent studies such as LibID [46] and LibRadar [32] can be used to find obfuscated libraries in the code. Then, the effect of the library features can be re-evaluated on the malware dataset.

The effects of permissions on the two datasets were also significantly different. When the average number of all 158 permissions was analyzed in both datasets, it was found that the use of different permissions was higher

TABLE 8. Comparison with AppAuth and stringAA - random forest.

Dataset	SPL+AppAuth+ StringAA		SPL+AppAuth		SPL		AppAuth		StringAA	
	acc	f1	acc	f1	acc	f1	acc	f1	acc	f1
Market	**82.5%**	**80.4%**	79.0%	76.6%	75.0%	72.4%	73.9%	71.2%	81.7%	79.9%
Malware	**95.6%**	**94.5%**	93.9%	92.5%	92.1%	90.7%	90.6%	89.4%	95.1%	94.2%
Genome	97.0%	96.7%	**98.2%**	**98.0%**	98.1%	97.9%	97.0%	96.9%	96.9%	96.7%

TABLE 9. Feature extraction time in minutes.

	SPL	AppAuth	StringAA (10,000)
Market	125.02	95.93	219.87
Malware	11.02	11.61	13.45
Genome	30.04	27.30	40.47

in the malware dataset (all permissions: 9.99, dangerous permissions: 4.5) than in the market dataset (all permissions: 8.05, dangerous permissions: 3).

e: RESULT #3

To show the statistical significance of the improvements, each approach (namely SPL+AppAuth+StringAA and StringAA) was run five times using 10-fold cross-validation. Thus, 50 results were obtained for each dataset. Note that the same folds were used for each approach in the experiment. A t-test, with an alpha value of 0.05, was then applied, and the results are shown in Table 12. α value represents the significance level, which is the probability of rejecting the null hypothesis when it is true. Table 12 shows that the p-values for the market and malware datasets are lower than α value of 0.05, which implies that the difference is statistically significant.

f: RESULT #4

To our knowledge, this is the first study to explore the use of source code-based features on smali codes for AAA. As shown in Table 10, such features performed much better than the other features proposed in the literature, except for n-grams. Such features can also be obtained from Java files. In the literature, it has been shown that more than 94% of Java classes can be successfully decompiled [47]. Therefore, the same source code-based features listed in Table 2 were collected from Java files and are compared in Table 13. However, only 80% of the applications were successfully decompiled. Please note that the following features specific to smali code are not available in Java files: the ratio of invoke to all code (the ratio of the number of invoking instructions to the number of all other instructions) and the ratio of move to all code. The comparison results show that the source code-based features collected from the smali files are more effective. However, it is worth mentioning that the decompilation of apk files into Java files could be erroneous. Here, the jadx decompiler [30] is used to convert apk files to Java files; however, we encountered some unsuccessful decompilations. In future, different decompilers should be evaluated.

Findings #4: *The experiments above show that the newly proposed features are highly effective for authorship attribution. Although string features [4] usually achieve better results than other features, it may not be practical to extract such features, particularly in large datasets. However, if possible, a combination of all features is recommended.*

5) RQ5 - METADATA FEATURES

a: MOTIVATION

As humans have different writing styles, text data are used largely for AA in different domains, such as social media, e-mail, and literature [48], [49], [50]. Android markets provide metadata such as descriptions, version numbers, and download counts. This information can also shed light on new findings. Therefore, the effect of metadata is first investigated for AAA in this study.

b: METHOD

Two different experiments were conducted in this RQ. First, we investigated the optimal n value and the number of n-grams. To extract n-grams, each n consecutive words in the descriptions are extracted, each time shifting one word to the right. Preliminary experiments were carried out with different n-gram sizes (from 1 to 5) to obtain the optimal value of n. Subsequently, the number of n-grams required to obtain the best results was determined. Second, we combined the metadata features with other feature sets to determine whether the metadata features improved accuracy.

c: RESULT #1

We first investigated the n value for the n-gram. The entire market dataset and only Rmvdroid [41] among the malware datasets were used because these datasets contain application descriptions. Fig. 8a shows the accuracy results for different n values. We set the number of n-grams to 10,000. It can be observed that 1-grams produce better results than the other n-grams. After determining the value of n, we investigated the optimal number of 1-gram features in further experiments. Fig. 8b shows the accuracy results obtained using only metadata features under different number of 1-grams. As shown in Fig. 8b, the number of 1-grams shows an important effect on the results. Because n-gram analysis requires a significant amount of system resources, we could not obtain results after 100,000 1-grams. If 100,000 1-grams are included, accuracies of approximately 84% and 74% are obtained for

TABLE 10. Effect of feature sets on AAA.

Dataset	SPL						AppAuth						StringAA	
	Src		Perm		Lib		Conf		Dex		Rsc		Str	
	acc	f1	acc	f1	acc	f1	acc	f1	acc	f1	acc	f1	acc	f1
Market	**66.8%**	**63.5%**	47.6%	42.4%	60.5%	57.6%	59.6%	55.8%	63.6%	60.0%	66.7%	63.4%	**81.7%**	**79.7%**
Malware	**87.3%**	**85.9%**	80.6%	76.5%	51.2%	46.2%	82.3%	79.3%	85.1%	82.3%	83.7%	80.7%	**95.1%**	**94.2%**
Genome	95.0%	94.7%	83.0%	80.6%	88.1%	86.8%	95.0%	94.6%	88.2%	87.7%	**96.2%**	**95.8%**	**96.8%**	**96.4%**

TABLE 11. Avg # of library and permission per applications.

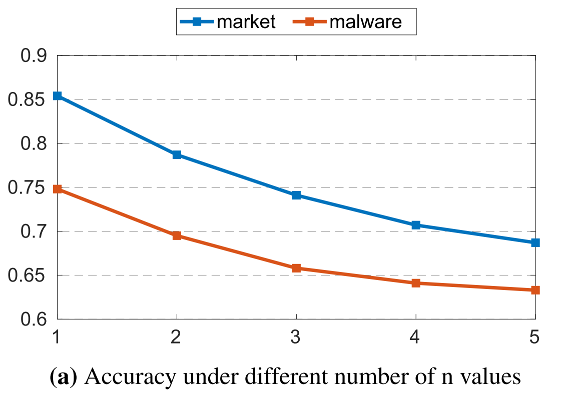
	Market	Malware
Library	9.37	2.72
Permissions (158)	8.05	9.99
Dangerous Permissions (26)	3	4.5

TABLE 12. Results of t-test.

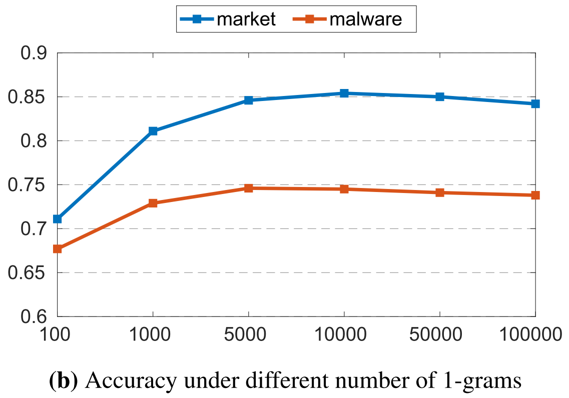
	p Value
Market	3.54622E-14
Malware	0.000562331
Genome	0.09986412

TABLE 13. Accuracy results of Smali vs. java: source code-based features.

	Smali	Java
Market	**66.2%**	52.9%
Malware	**87.0%**	86.8%



(a) Accuracy under different number of n values



(b) Accuracy under different number of 1-grams

FIGURE 8. Effect of n-grams on metadata features.

the market and malware datasets, respectively. This clearly shows that the application descriptions are beneficial to AAA. However, it can be seen that the results of 10,000 1-grams are also quite sufficient, and the time required is much less than 100,000 1-grams. Therefore, we chose 10,000 1-grams for further experiments in the metadata analysis.

d: RESULT #2

Table 14 shows the effects of metadata features in the market and malware datasets, respectively. Because not all applications have descriptions, the dataset used in this experiment was smaller than the original dataset. Application descriptions can be written in any language. Our datasets also include different languages in the application descriptions, such as English, Chinese, and French. It is observed that accuracy increases significantly when we use descriptions in English only. Therefore, descriptions other than English were eliminated from both the datasets. In the market dataset, metadata features produce approximately 6% better accuracy when combined with the proposed SPL features, whereas an increase of approximately 3% is observed for other feature sets. In the malware dataset, metadata features showed a clear positive effect, with an increase of 3% in the proposed SPL features. In contrast, the increase was not significant for the other feature sets.

Findings #5: *Because humans have different writing styles, application descriptions could help attribute the developers of applications. This is proved by the experimental results presented herein. However, having multiple language descriptions in the dataset severely affects the classification performance.*

6) RQ6 - MOST EFFECTIVE FEATURES ON AAA

a: MOTIVATION

In addition, to know why our prediction is high or low, we also want to know which features contribute more and which are irrelevant in improving our prediction. Therefore, in this experiment, we investigated the features that had the greatest influence on the results.

TABLE 14. Metadata analysis.

Dataset	SPL		SPL+Meta		StringAA		StringAA+Meta		AppAuth		AppAuth+Meta	
	acc	f1	acc	f1	acc	f1	acc	f1	acc	f1	acc	f1
Market	81.4%	78.4%	87.0%	85.0%	86.6%	84.5%	89.0%	87.0%	82.6%	80.1%	85.6%	83.3%
Malware	89.1%	87.1%	92.0%	90.3%	94.7%	93.5%	95.0%	93.8%	89.0%	86.7%	89.1%	87.0%

Dataset	SPL+AppAuth +StringAA		SPL+AppAuth+ StringAA+Meta		SPL+AppAuth		SPL+AppAuth +Meta	
	acc	f1	acc	f1	acc	f1	acc	f1
Market	86.8%	84.7%	89.0%	87.1%	84.5%	82.3%	88.4%	86.6%
Malware	94.9%	93.7%	95.4%	94.3%	91.9%	90.5%	93.6%	92.3%

b: METHOD

The results were obtained using all features except metadata and the Random Forest algorithm. Then the importance values of all the features are extracted from scikit-learn library and shown in Fig. 9a and 9b, for the market and malware datasets, respectively.

c: RESULT

It is shown that the source code-based and string/n-gram features have a greater influence than the other features on both datasets. The features used in AppAuth (conf, dex, rsrc) have a positive effect, particularly on the malware dataset. Table 15 shows the number of features per dataset placed in the top 50 features in terms of importance. The order of the 50 most important features is listed in Table 22 in the Appendix. These results also show that source code-based features have much more impact than other features, especially in the malware dataset, whereas n-gram features dominate the market dataset.

TABLE 15. Distribution of top 50 features per feature sets.

Dataset	Src	Perm	Lib	Conf	Dex	Rsrc	Str
Market	8	1	2	0	0	1	38
Malware	14	1	0	2	6	0	27

Findings #6: *The proposed source code and string/ngram features are much more effective than the other features of AAA.*

7) RQ7 - THE EFFECT OF THE NUMBER OF FEATURES

a: MOTIVATION

As we use over 10,000 different attributes, some features may be redundant and have nan values, or certain features may have the same value within themselves, meaning that they have zero variance. Such features must be removed from the feature vectors to positively affect the classifier. Therefore, the effect of the number of features was also explored.

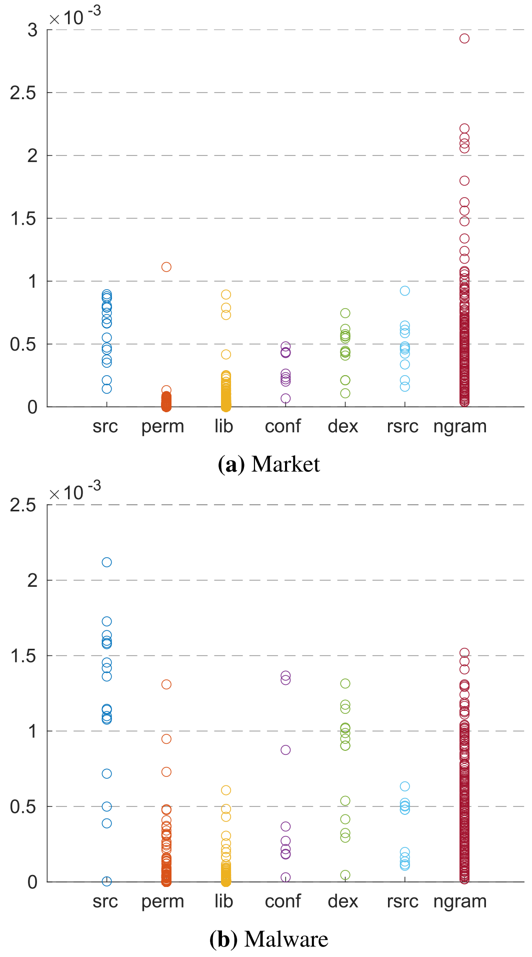


FIGURE 9. Importance values of each features.

b: METHOD

The best 20%, 40%, 60%, 80%, and 100% of all features were extracted by computing the ANOVA F-values using *f_classif()* in scikit-learn. ANOVA is a parametric statistical hypothesis test that is used for feature selection. The means of two or more samples of data are calculated first, and then if these data samples come from the same distribution is determined.

c: RESULT

The performance of each feature group is shown in Fig. 10. Please note that zero-impact features are also present. Becuase the malware dataset has more zero-impact features (435) than the market dataset (174), there was not much improvement in the last percentile. This figure also shows that if time and resources matter, fewer features (for example, 60%) can be used as replacements for all features.

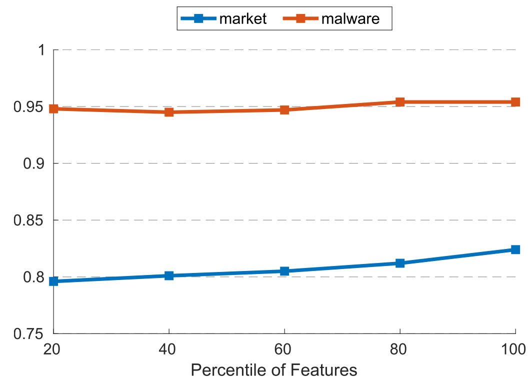


FIGURE 10. **Impact of different percentiles of features.**

Findings #7: *Good enough results can be obtained by using fewer features (e.g. 60% of all features). This reduces the time and resources required for classification.*

8) RQ 8 - THE EFFECT OF THE NUMBER OF APPLICATIONS PER DEVELOPER

a: MOTIVATION

As we use supervised machine learning to get results, we need prior information on developers. Therefore, the number of applications per developer may affect our prediction model. Here, we conduct an experiment to show how the number of applications per developer affects our prediction model.

b: METHOD

We would like to emphasize that our dataset includes mainly company applications written by multiple developers since it is difficult to obtain individual developers with more than ten applications in the market. Moreover, our dataset is imbalanced since developers have different numbers of applications. Therefore, developers with at least 40 applications in both datasets are first extracted, and 37 and 19 authors are obtained for the market and malware datasets, respectively. Then 10, 20, 30, and 40 applications are randomly selected per developer ten times. Therefore, each time, different groups of applications are selected.

c: RESULT

Fig. 11 shows that when the number of applications per developer is increased, the accuracy of AAA is also increased. Due to being a smaller dataset with fewer authors, the results on the malware dataset are very high, even when trained with 20 applications per developer. Therefore, the differences among the models trained using 20, 30, and 40 applications per developer in the malware dataset are similar.

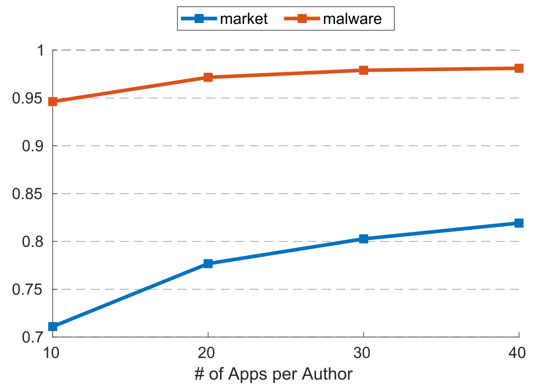


FIGURE 11. **Impact of different # of applications per developer.**

Findings #8: *If a more balanced dataset with enough samples per developer is employed, the performance of the model gets even higher.*

9) RQ 9 - EFFECT OF APPLICATION VERSIONS

a: MOTIVATION

Developers often update their applications due to bug fixes, security patches, adding new functionalities, and the like. Therefore, there can be multiple versions of an application on the developer's market page. As our market dataset also consists of different versions of some applications, it is worth investigating whether the versions of the applications developed by the same author can be identified by the proposed method.

b: METHOD

Therefore, two datasets were constructed in this study. First, if applications have more than one version in the market dataset, duplicate applications are eliminated, and only the first available version of the application is left in the market dataset. This training dataset is called the no-version dataset; their versions are put into another dataset called the testing set. Therefore, 1,193 training and 1,183 testing applications implemented by 42 developers were used. All developers in the no-version dataset had at least ten applications, as in the previous experiments.

c: RESULT

We first obtained the result by applying 10-fold cross-validation on the no-version set and achieved 80.7% accuracy. As shown in Table 8, if all versions were included, the accuracy is slightly higher (82.6%). Then, to answer the question "if a version of an application is included in the training, could new versions of this application be detected with the proposed approach?", the model was trained using a no-version dataset and evaluated on the testing dataset. Here, high accuracy (91.7%) was obtained. Similarities between the different versions of the applications were obtained using SimiDroid [44]. However, there was no significant correlation between the unidentified versions of applications and their similarities to the first available versions in training. Although similarities to the first available version generally decreased proportionally to the version numbers, there were exceptional cases in the testing set. Although the similarity between a version of the application and the first available version in the training set is low, the proposed approach can successfully identify the authors of such versions. These results show that the developer's signature is preserved, even if the similarity between different versions of the same application is low.

Finding #9: *The proposed model can detect applications and their different versions that preserve the developer's signature.*

10) RQ 10 - EFFECT OF OBFUSCATION

a: MOTIVATION

Because both benign and malicious applications apply obfuscation techniques, AAA has been evaluated in obfuscated applications in the literature [4], [14]. In [14], obfuscated applications were obtained using ProGuard [51], which provides simple obfuscation techniques, such as method, class, and identifier renaming, along with code shrinking and code optimization. It is shown that the authors of some applications (7%) could not be identified when they were obfuscated. AppAuth [6] also claims that its features are not robust against encryption and shell package obfuscation. Kalgutkar et al. [4] analyzed obfuscated applications using three different obfuscation tools: ProGuard, Allatori [52], and Dasho [53]. They employed different types of obfuscation techniques, such as string obfuscation, string encryption, and control flow obfuscation to the source code of applications. However, they worked on a very small dataset with 96 applications from nine different authors. Their results showed that the accuracy results of AA in applications obfuscated by ProGuard are unexpectedly 6% better than those of the original applications. String features are expected to be less robust against string obfuscation and encryption techniques; however, it is shown that there is no significant change in the results when applications are obfuscated using the other two obfuscation tools.

b: METHOD

In this study, to better understand the performance of AAA in obfuscated applications, the Obfuscapk [54] tool was used in applications (smali files) in the market and Genome datasets. Obfuscapk [54] works in a black-box fashion and supports advanced obfuscation features, and has a modular architecture that is easily extensible with new techniques. Note that because of some errors encountered during the application of obfuscation techniques, the number of applications used in this experiment was much lower than that of the original dataset (≈40%). However, it is a much larger dataset (6055 applications from 320 authors) than [4]. The six different obfuscation techniques listed in Table 16 were used. These obfuscation techniques can be grouped into two categories: encryption and renaming.

c: RESULT

The effects of different obfuscation techniques are shown in Table 17. Although encryption obfuscation techniques do not affect the results, the newly proposed source code-based feature sets are susceptible to renaming obfuscation techniques, as shown in Table 17, because they are extracted from the variable, method, and class names of applications. String features [4] are also affected by renaming techniques, but less than the newly proposed feature sets, because string features use all strings placed in the apk files, not only strings extracted from the source codes.

TABLE 16. **Obfuscation abbreviations.**

Techniques	Abbrv.	Description
ConstStringEncryption	CSE	Encrypt constant strings in code
LibEncryption	LE	Encrypt native libs
ResStringEncryption	RSE	Encrypt strings in resources (only those called inside code)
ClassRename	CR	Change the package name and rename classes (even in the manifest file)
MethodRename	MR	Rename methods
FieldRename	FR	Rename fields

Findings #10: *All features, especially those extracted from strings in the source code, such as variable names, are affected by obfuscation techniques, particularly renaming techniques.*

11) RQ 11 - ANALYSIS OF CLONE APPLICATIONS

a: MOTIVATION

To ensure data quality, we checked whether our dataset contained application clones and identical descriptions.

b: METHOD

The Romadroid tool [55] was used to detect application clones in the dataset. Romadroid creates a string from each manifest file of two applications to be compared and measures the similarity between the two strings using the longest common subsequence (LCS) algorithm. The authors of Romadroid compared their tool with SimiDroid [44] and claimed that Romadroid performed better than SimiDroid over a 60% threshold. Their results showed that Simidroid produced a much lower recall value at their best threshold values than did Romadroid (60.64% vs. 98.38%) in the same dataset.

We used the 70% and 90% threshold values in our experiments, so we calculated the similarity scores of applications of 488, 153, and 39 developers in the market, malware, and Genome datasets using Romadroid. As a result, $n * (n - 1)/2$ similarity results were obtained by pairwise comparison of applications. Here, n denotes the total number of applications. Because we encountered errors in some applications, we could not obtain similarity scores for each

TABLE 17. Obfuscation results.

Dataset	Technique	SPL+AppAuth+StringAA	SPL+AppAuth	SPL	AppAuth	StringAA
Genome (1332 applications)	**Original**	97.0%	**97.9%**	97.7%	97.2%	96.8%
	CR+MR+FR	95.9%	**97.6%**	96.5%	97.0%	96.0%
	CSE+LE+RSE	96.9%	**98.1%**	97.8%	97.3%	96.7%
	CSE+LE+RSE+CR+MR+FR	96.3%	**97.7%**	96.5%	97.4%	95.9%
Market (6055 applications)	**Original**	**82.8%**	78.4%	75.3%	73.7%	82.3%
	CR+MR+FR	**81.4%**	75.3%	66.2%	73.5%	80.9%
	CSE+LE+RSE	**82.3%**	78.5%	75.2%	73.9%	81.6%
	CSE+LE+RSE+CR+MR+FR	**81.5%**	75.4%	66.4%	73.6%	80.8%

pair. Many studies typically consider an application pair as an application clone if their similarity scores exceed either %70 or %90 [44], [55], [56]. In our approach, we excluded all applications that exhibited a similarity score above %70 with another.

c: RESULT

As shown in Table 18, 20.6% of the market and 22.3% of the malware datasets had similar applications above the 70% similarity rate. The effects of clone apps on the results are given in Tables 20 and 19 for the malware and market datasets, respectively. The results indicate that removing similar applications improves the accuracy and F1 scores, especially for the market dataset. As similar applications from different developers can mislead the model during training, removing app clones has a positive effect on the results.

TABLE 18. Ratio of clone applications in the datasets.

similarity threshold	>%70	>%80	>%90
Market	20.6%	14.0%	8.5%
Malware	22.3%	13.0%	9.0%
Genome	10.6%	11.0%	0.3%

TABLE 19. Differences (%) on accuracy and f1 score when clone apps are removed from the market dataset.

	SPL	AppAuth	StringAA
acc	+3.211	+1.608	+2.729
f1	+3.608	+1.860	+3.161

TABLE 20. Differences(%) on accuracy and f1 score when clone apps are removed from the malware dataset.

	SPL	AppAuth	StringAA
acc	+0.246	+0.337	+0.477
f1	+0.207	+0.322	+0.476

Findings #11: *App clones that are distributed to multiple users may mislead training the model.*

VIII. THREATS TO VALIDITY

In this study, we explore the use of source code-based features and compare different feature sets in the literature on AAA. However, this study has a few limitations, detailed in below:

A. USAGE OF NATIVE CODE

Today, developers are increasingly using native code within Android application packages, where they co-exist and interact with Dex bytecode through the Java Native Interface [57]. However, analyzing native code requires high-fidelity execution traces and memory data with a low overhead. This resulted in incomplete and biased results [58]. Our approach requires a complete view of the executable code in applications. Only the non-native code parts of applications are used to extract features, as in [6]. On the other hand, developers can use Android NDK, which can help them to reuse code libraries to embed in Android apps; therefore, native code written in C/C++ could carry developers' fingerprints. Because our focus is on the use of source code-based features for AAA in this study, the features that could be extracted from .smali files are used. However, native codes could be included in the AAA in the future.

B. OBFUSCATION

As indicated in Table 17, our proposed source code-based features are susceptible to renaming obfuscation techniques. Therefore, different groups of features, such as metadata and string features, can be combined to identify authors.

A whitelist approach was adapted to find third-party libraries used in an application. However, this approach cannot detect obfuscated third-party libraries. When such libraries cannot be identified correctly, they are considered a developer's custom code, which causes an increase in the similarities between applications developed by different authors that use the same libraries. Additionally, such libraries cannot be used as features to distinguish developers. Therefore, recent studies such as LibID [46] and LibRadar [32] can be used to find obfuscated libraries in the future.

C. DATASET

The size of the dataset is important factor in any machine learning-based approach. Even though the dataset in this study is larger and more comprehensive than those in other studies in the literature as shown in Table 21, further studies could extend it. One limitation is finding developers with sufficient applications so that the model can learn their styles. Another limitation is the need to find applications with proper descriptions. In this study, we considered only applications that have descriptions written in English. The size of the dataset can be increased by including other languages.

TABLE 21. Dataset information.

Dataset	# of app	# of author
Our paper	15,183	680
AppAuth [6]	3,871	273
String Analysis [4]	1,917	59

D. CLONE/REPACKAGED APPLICATIONS

Our study assumes that if an application is signed by a developer, it is written by that developer. This is the same approach taken in previous studies [4], [6]. However, as indicated earlier, there might be clone applications in the market stores; hence, in our dataset, their existence might have affected our results. However, in this study, a simple approach based on code similarity was employed to detect app clones. However, further research is necessary, which is beyond the scope of this study.

Many researchers have been working on detecting app clones or repackaged applications effectively [56]. It is stated that [56] while there is widespread recognition of the app repackaging issue in both academic and industrial circles, there's a notable lack of datasets to aid research. Creating a comprehensive set of repackaging pairs, serving as ground truth, requires significant effort. With the introduction of a new, large dataset in [56], we expect such studies to accelerate. For these reasons, the investigation of the effects of app clones on AAA is left for future work.

E. APPLICATIONS DEVELOPED BY MULTIPLE AUTHORS

Kalgutkar et al. [7] highlighted that many studies operate under the assumption that individual applications in their dataset are authored by a singular developer. This is in contrast to real-world scenarios where software applications often involve contributions from multiple developers. Gong et al. further expressed this point, indicating that between 10% and 60% of source files can contain contributions from unattributed authors and as many as 75.4% of source files exhibit multiple authorships [59].

For this study, we assume that a single developer is responsible for a given application. However, it is imperative to acknowledge that many applications, particularly large-scale commercial software, are collective efforts of multiple developers. This multiplicity adds layers of complexity to the issue at hand. Notably, when developers adhere to institutionalized guidelines for software development, the overarching organization can be treated as a singular entity or developer within the scope of this study. For instance, even on Github, some developers specify a set of rules or guidelines that contributors need to follow to contribute to a project, this is typically called a CONTRIBUTING guide or CONTRIBUTING.md file. Additionally, companies may implement regular code review processes. This ensures that all codes are checked for quality and adherence to the standards before they are merged into the main codebase. The unified coding standards establish a set of coding standards that all developers in the company must follow. Therefore, if some of the codes are written by multiple developers, the total code of the application can be seen as the work of a single developer. This ensures consistency in the codebase, thus making it easier to read, maintain, and debug.

TABLE 22. The most important 50 features per dataset.

Market		Malware	
Name	**Type**	**Name**	**Type**
interstitial is already	ngram	avgCharPerGlobalVar	src
1s kjrer ikk	ngram	avgLinePerClass	src
lb itt lb	ngram	avgFuncPerClass	src
saved state of	ngram	ratioInvokeToAllCodes	src
elle sera bient	ngram	ratioVoidToAllFunc	src
metadata tag in	ngram	ratioVarToAllCodes	src
lb samplerate lb	ngram	lb getruntime lb	ngram
lb cannot find	ngram	or zero length	ngram
invalid ad size	ngram	lb replace lb	ngram
lb rgb lb	ngram	ratioIfToAllCodes	src
lb isinterface lb	ngram	avgLinePerFunc	src
freeing fragment index	ngram	dapatkan perkhidmatan google	ngram
change.component.enabled.state	perm	susesPermissionNum	conf
container view with	ngram	ratioGlobalVarToAllCodes	src
lb krko lb	ngram	usesPermissionNum	conf
lb score lb	ngram	vtlMethodRatio	dex
lb readbyte lb	ngram	lb plusclient must	ngram
lb pair lb	ngram	receive.mms	perm
lb aan lb	ngram	lb marketsearchqpnamecomgoogle lb	ngram
be null instead	ngram	this message lb	ngram
lb zzafm lb	ngram	lb initializing adview	ngram
lb onanimationend lb	ngram	permissions are not	ngram
lb zc lb	ngram	lb settextalign lb	ngram
lb landroidviewsurface lb	ngram	antClassRatio	dex
lb zzti lb	ngram	lb ljavalangcharsequence lb	ngram
cannot call this	ngram	lb lapp non	ngram
resDrawableFileNum	rsrc	statFieldRatio	dex
como 1s lb	ngram	ratioMoveToAllCodes	src
share via lb	ngram	ratioLocalVarToAllCodes	src
avgCharPerLocalVar	src	lb writedouble lb	ngram
com/appyet	lib	lb iil lb	ngram
lb failure lb	ngram	lb row lb	ngram
ratioInvokeToAllCodes	src	avgCharPerLocalVar	src
vi cc dch	ngram	avgCharPerFuncName	src
avgCharPerFuncName	src	ratioIntToAllFunc	src
avgLinePerClass	src	lb getconfig lb	ngram
key cannot be	ngram	lb landroidgraphicsbitmapconfig lb	ngram
lb multiply lb	ngram	lb stopplayback lb	ngram
not forward oncreate	ngram	dbiMethodRatio	dex
ratioVarToAllCodes	src	lb zllil lb	ngram
giving up on	ngram	agetRatio	dex
ratioGlobalVarToAllCodes	src	the main ui	ngram
avgCharPerGlobalVar	src	lb iso88591 lb	ngram
com/doapps	lib	lb packagename lb	ngram
or out of	ngram	lb getsnippet lb	ngram
lb case_insensitive_order lb	ngram	lb module without	ngram
at least one	ngram	lb zu lb	ngram
lb ce lb	ngram	drtMethodRatio	dex
lb landroidappalarmmanager lb	ngram	lb getlastpathsegment lb	ngram
ratioMoveToAllCodes	src	lipsesc de pe	ngram

IX. CONCLUSION

The present study investigates the use of various features for Android Authorship Attribution (AAA). Specifically, the features explored in this research include source code-based attributes extracted from smali files, as well as metadata, libraries, and permission-based features. The source code-based attributes are extracted from the intermediate representation of Android applications, which retain the source code-based information. To evaluate the effectiveness of these features, extensive experiments were conducted on large datasets containing benign, malicious, and obfuscated applications sourced from various platforms. The results demonstrate that the newly proposed feature sets exhibit high accuracy in AAA, comparable to string-based features that utilize greater computational resources to extract information from application strings. Notably, it is observed that metadata features contribute to improve accuracy, particularly when combined with source code-based features. Additionally, this study examines the impact of application versions, obfuscation, and third-party libraries on AAA. Of particular significance, this is the first study to explore the impact of metadata information, which describes applications, on AAA. Overall, the present work provides a rigorous analysis of AAA and is compared with the state-of-the-art techniques. Future research should investigate the use of this approach for clone applications, particularly those with malicious intent.

APPENDIX
MOST IMPORTANT FEATURES
See Table 22.

REFERENCES

[1] N. Rosenblum, X. Zhu, and B. P. Miller, "Who wrote this code? Identifying the authors of program binaries," in *Computer Security—ESORICS 2011*, V. Atluri and C. Diaz, Eds. Berlin, Germany: Springer, 2011, pp. 172–189.

[2] S. Alrabaee, P. Shirani, M. Debbabi, and L. Wang, "On the feasibility of malware authorship attribution," in *Foundations and Practice of Security*, F. Cuppens, L. Wang, N. Cuppens-Boulahia, N. Tawbi, and J. Garcia-Alfaro, Eds. Cham, Switzerland: Springer, 2017, pp. 256–272.

[3] S. Alrabaee, L. Wang, and M. Debbabi, "BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs)," *Digit. Invest.*, vol. 18, pp. S11–S22, Aug. 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287616300330

[4] V. Kalgutkar, N. Stakhanova, P. Cook, and A. Matyukhina, "Android authorship attribution through string analysis," in *Proc. 13th Int. Conf. Availability, Rel. Secur. (ARES)*. New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 1–10, doi: 10.1145/3230833.3230849.

[5] H. Gonzalez, N. Stakhanova, and A. A. Ghorbani, "Authorship attribution of Android apps," in *Proc. 8th ACM Conf. Data Appl. Secur. Privacy (CODASPY)*. New York, NY, USA: Association for Computing Machinery, Mar. 2018, pp. 277–286, doi: 10.1145/3176258.3176322.

[6] G. Xu, C. Zhang, B. Sun, X. Yang, Y. Guo, C. Li, and H. Wang, "AppAuth: Authorship attribution for Android app clones," *IEEE Access*, vol. 7, pp. 141850–141867, 2019.

[7] V. Kalgutkar, R. Kaur, H. Gonzalez, N. Stakhanova, and A. Matyukhina, "Code authorship attribution: Methods and challenges," *ACM Comput. Surv.*, vol. 52, no. 1, pp. 1–36, Feb. 2019, doi: 10.1145/3292577.

[8] S. Alrabaee, N. Saleem, S. Preda, L. Wang, and M. Debbabi, "OBA2: An onion approach to binary code authorship attribution," *Digit. Invest.*, vol. 11, pp. S94–S103, May 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287614000176

[9] A. Caliskan, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan, "When coding style survives compilation: De-anonymizing programmers from executable binaries," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2018, doi: 10.14722/ndss.2018.23304.

[10] R. Layton and A. Azab, "Authorship analysis of the Zeus botnet source code," in *Proc. 5th Cybercrime Trustworthy Comput. Conf.*, Nov. 2014, pp. 38–43.

[11] R. Layton, S. McCombie, and P. Watters, "Authorship attribution of IRC messages using inverse author frequency," in *Proc. 3rd Cybercrime Trustworthy Comput. Workshop*, Oct. 2012, pp. 7–13.

[12] M. Alazab, R. Layton, R. Broadhurst, and B. Bouhours, "Malicious spam emails developments and authorship attribution," in *Proc. 4th Cybercrime Trustworthy Comput. Workshop*, Nov. 2013, pp. 58–68.

[13] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *Proc. 24th USENIX Conf. Secur. Symp. (SEC)*. Berkeley, CA, USA: USENIX Association, 2015, pp. 255–270.

[14] W. Wang, G. Meng, H. Wang, K. Chen, W. Ge, and X. Li, "A3Ident: A two-phased approach to identify the leading authors of Android apps," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2020, pp. 617–628.

[15] StatCounter. (2023). *Mobile Operating System Market Share Worldwide*. Accessed: Dec. 12, 2023. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[16] Google. (2023). *Android Apps on Google Play*. Accessed: Dec. 12, 2023. [Online]. Available: https://play.google.com/store/apps

[17] Aptoide. (2023). *Aptoide—The Alternative Android App Store*. Accessed: Dec. 12, 2023. [Online]. Available: https://en.aptoide.com

[18] APKMirror. (2023). *APKMirror—Free APK Downloads—Free and Safe Android APK Downloads*. Accessed: Dec. 12, 2023. [Online]. Available: https://www.apkmirror.com

[19] Android Developers. (2023). *String Resources | Android Developers*. Accessed: Dec. 12, 2023. [Online]. Available: https://developer.android.com/guide/topics/resources/string-resource

[20] Connor Tumbleson. (2023). *Apktool—A Tool for Reverse Engineering Android APK Files*. Accessed: Dec. 12, 2023. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[21] T. K. Ho, "Random decision forests," in *Proc. 3rd Int. Conf. Document Anal. Recognit.*, vol. 1, 1995, pp. 278–282.

[22] Scikit-Learn Developers. (2023). *Scikit-Learn: Machine Learning in Python—Scikit-Learn 1.3.2 Documentation*. Accessed: Dec. 12, 2023. [Online]. Available: https://scikit-learn.org/stable/index.html

[23] H. He and Y. Ma. *Imbalanced Learning: Foundations, Algorithms, and Applications*. Hoboken, NJ, USA: Wiley, 2013, pp. 1–11. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118646106

[24] S. Sen and B. Can, "Android security using NLP techniques: A review," 2021, *arXiv:2107.03072*.

[25] PrivacyGrade. (2021). *Privacygrade: Grading the Privacy of Smartphone Apps*. Accessed: 2021. [Online]. Available: http://privacygrade.org

[26] H. Wang, Y. Guo, Z. Ma, and X. Chen, "WuKong: A scalable and accurate two-phase approach to Android app clone detection," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*. New York, NY, USA: Association for Computing Machinery, Jul. 2015, pp. 71–82, doi: 10.1145/2771783.2771795.

[27] H. Wang and Y. Guo, "Understanding third-party libraries in mobile app analysis," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2017, pp. 515–516.

[28] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "An investigation into the use of common libraries in Android apps," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, vol. 1, Mar. 2016, pp. 403–414.

[29] S. Burrows, A. L. Uitdenbogerd, and A. Turpin, "Comparing techniques for authorship attribution of source code," *Softw., Pract. Exper.*, vol. 44, no. 1, pp. 1–32, Jan. 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2146

[30] Skylot. (2023). *GitHub—Skylot/JADX: Dex to Java Decompiler*. Accessed: Dec. 12, 2023. [Online]. Available: https://github.com/skylot/jadx

[31] H. Wang, Y. Guo, Z. Tang, G. Bai, and X. Chen, "Reevaluating Android permission gaps with static and dynamic analysis," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2015, pp. 1–6.

[32] Z. Ma, H. Wang, Y. Guo, and X. Chen, "LibRadar: Fast and accurate detection of third-party libraries in Android apps," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2016, pp. 653–656.

[33] MVNRepository. (2023). *Maven repository: Central*. Accessed: Dec. 12, 2023. [Online]. Available: https://mvnrepository.com/repos/central

[34] Scrapy. (2023). *Scrapy | A Fast and Powerful Scraping and Web Crawling Framework*. Accessed: Dec. 12, 2023. [Online]. Available: https://scrapy.org

[35] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, Feb. 2014.

[36] PRALab. (2021). *Android Praguard Dataset*. Accessed: 2021. [Online]. Available: http://pralab.diee.unica.it/en/AndroidPRAGuardDataset

[37] Warren-Bank. (2023). *GitHub—Warren-Bank/Print-APK-Signature: Print Information About the Certificate Used to Sign an Android APK File*. Accessed: Dec. 12, 2023. [Online]. Available: https://github.com/warren-bank/print-apk-signature

[38] APKPure. (2023). *Download APK on Android With Free Online APK Downloader—APKPure*. Accessed: Dec. 12, 2023. [Online]. Available: https://apkpure.com

[39] 1Mobile-Market. (2018). *1Mobile Market—Best Google Android Apps Market*. Accessed: 2018. [Online]. Available: http://market.1mobile.com

[40] Canadian Institute for Cybersecurity. (2023). *Datasets | Research | Canadian Institute for Cybersecurity | UNB*. Accessed: Dec. 12, 2023. [Online]. Available: https://www.unb.ca/cic/datasets/index.html

[41] H. Wang, J. Si, H. Li, and Y. Guo, "RmvDroid: Towards a reliable Android malware dataset with app metadata," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, May 2019, pp. 404–408.

[42] S. Alrabaee, P. Shirani, L. Wang, M. Debbabi, and A. Hanna, "Decoupling coding habits from functionality for effective binary authorship attribution," *J. Comput. Secur.*, vol. 27, no. 6, pp. 613–648, Oct. 2019.

[43] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in Android and its security applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 356–367, doi: 10.1145/2976749.2978333.

[44] L. Li, T. F. Bissyandé, and J. Klein, "SimiDroid: Identifying and explaining similarities in Android apps," in *Proc. IEEE Trustcom/BigDataSE/ICESS*, Aug. 2017, pp. 136–143.

[45] G. Goel, H. Bhardwaj, I. Hooda, and S. Kumar, "Optimal N-gram subset extraction for accelerating evaluation using genetic algorithm," in *Proc. Int. Conf. Emerg. Technol. (INCET)*, Jun. 2020, pp. 1–5.

[46] J. Zhang, A. R. Beresford, and S. A. Kollmann, "LibID: Reliable identification of obfuscated third-party Android libraries," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 55–65, doi: 10.1145/3293882.3330563.

[47] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proc. 20th USENIX Conf. Secur. (SEC)*. Berkeley, CA, USA: USENIX Association, 2011, p. 21.

[48] D. Boyd, S. Golder, and G. Lotan, "Tweet, tweet, retweet: Conversational aspects of retweeting on Twitter," in *Proc. 43rd Hawaii Int. Conf. Syst. Sci.*, Jan. 2010, pp. 1–10.

[49] N. Cheng, R. Chandramouli, and K. Subbalakshmi, "Author gender identification from text," *Digit. Invest.*, vol. 8, no. 1, pp. 78–88, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287611000247

[50] Y. Zhao and J. Zobel, "Searching with style: Authorship attribution in classic literature," in *Proc. 13th Australas. Conf. Comput. Sci. (ACSC)*, vol. 62. Docklands, VIC, Australia: Australian Computer Society, 2007, pp. 59–68.

[51] Android Developers. (2023). *Shrink, Obfuscate, and Optimize Your App | Android Studio | Android Developers*. Accessed: Dec. 12, 2023. [Online]. Available: https://developer.android.com/studio/build/shrink-code

[52] Smardec Inc. (2023). *Allatori Java Obfuscator—Professional Java Obfuscation*. Accessed: Dec. 12, 2023. [Online]. Available: http://www.allatori.com

[53] PreEmptive Solutions. (2023). *Dasho | Preemptive*. Accessed: Dec. 12, 2023. [Online]. Available: https://www.preemptive.com/products/dasho

[54] C. Georgiu. (2023). *GitHub—Claudiugeorgiu/Obfuscapk: An Automatic Obfuscation Tool for Android Apps That Works in a Black-Box Fashion, Supports Advanced Obfuscation Features and has a Modular Architecture Easily Extensible With New Techniques*. Accessed: Dec. 12, 2023. [Online]. Available: https://github.com/ClaudiuGeorgiu/Obfuscapk

[55] B. Kim, K. Lim, S.-J. Cho, and M. Park, "RomaDroid: A robust and efficient technique for detecting Android app clones using a tree structure and components of each app's manifest file," *IEEE Access*, vol. 7, pp. 72182–72196, 2019.

[56] L. Li, T. F. Bissyandé, and J. Klein, "Rebooting research on detecting repackaged Android apps: Literature review and benchmark," *IEEE Trans. Softw. Eng.*, vol. 47, no. 4, pp. 676–693, Apr. 2021.

[57] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "JuCify: A step towards Android code unification for enhanced static analysis," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 1232–1244, doi: 10.1145/3510003.3512766.

[58] H. Zhou, S. Wu, X. Luo, T. Wang, Y. Zhou, C. Zhang, and H. Cai, "NCScope: Hardware-assisted analyzer for native code in Android apps," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*. New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 629–641, doi: 10.1145/3533767.3534410.

[59] S. Gong and H. Zhong, "Code authors hidden in file revision histories: An empirical study," in *Proc. IEEE/ACM 29th Int. Conf. Program Comprehension (ICPC)*, May 2021, pp. 71–82.

EMRE AYDOGAN received the B.S. degree in computer engineering from Erciyes University and the M.S. degree in computer engineering from Hacettepe University, Turkey, where he is currently pursuing the Ph.D. degree in computer engineering, under the supervision of Dr. Sevil Sen. He was a Research Assistant with the Department of Computer Engineering, Hacettepe University, from 2014 to 2018, and Akdeniz University, Turkey, from 2018 to 2022. He is a member of the Wireless Networks and Intelligent Secure Systems (WISE) Laboratory, Hacettepe University. His main research interests include the Internet of Things (IoT) security and malware analysis.

SEVIL SEN is currently a Professor with the Department of Computer Engineering, Hacettepe University, and leads the Wireless Networks and Intelligent Secure Systems (WISE) Laboratory. Her research interests include computer networks and networks and systems security. Her focus is mainly in the area of mobile systems and wireless networks. She is currently serving as an Area Editor for *Ad Hoc Networks* and *Genetic Programming and Evolvable Machines*.