A^3IDENT: A Two-phased Approach to Identify the Leading Authors of Android Apps

Wei Wang[1*], Guozhu Meng[2,3*], Haoyu Wang[4], Kai Chen[2,3], Weimin Ge[1], and Xiaohong Li[1]

[1]Tianjin Key Laboratory of Advanced Networking (TANK), School of Computer Science and Technology, College of Intelligence and Computing, Tianjin University, Tianjin, China, {wei2018, gewm, xiaohongli}@tju.edu.cn
[2]Institute of Information Engineering, Chinese Academy of Sciences, China, {mengguozhu, chenkai}@iie.ac.cn
[3]School of Cyber Security, University of Chinese Academy of Sciences, China
[4]Beijing University of Posts and Telecommunications, Beijing, China, haoyuwang@bupt.edu.cn

***Abstract*—Authorship identification is the process of identifying and classifying authors through given codes. Authorship identification can be used in a wide range of software domains, e.g., code authorship disputes, plagiarism detection, exposure of attackers' identity. Besides the inherent challenges from legacy software development, framework programming and crowdsourcing mode in Android raise the difficulties of authorship identification significantly. More specifically, widespread third party libraries and inherited components (e.g., classes, methods, and variables) dilute the primary code within the entire Android app and blur the boundaries of code written by different authors. However, prior research has not well addressed these challenges.**

To this end, we design a two-phased approach to attribute the primary code of an Android app to the specific developer. In the first phase, we put forward three types of strategies to identify the relationships between Java packages in an app, which consist of context, semantic and structural relationships. A package aggregation algorithm is developed to cluster all packages that are of high probability written by the same authors. In the second phase, we develop three types of features to capture authors' coding habits and code stylometry. Based on that, we generate fingerprints for an author from its developed Android apps and employ several machine learning algorithms for authorship classification. We evaluate our approach in three datasets that contain 15,666 apps from 257 distinct developers and achieve a 92.5% accuracy rate on average. Additionally, we test it on 2,900 obfuscated apps and our approach can classify apps with an accuracy rate of 80.4%.

***Index Terms*—authorship identification; authorship decoupling; android app; package relation graph; leading author**

I. INTRODUCTION

Code authorship identification is a generic technique of determining the author for a specific piece of code. It has been widely used in multiple areas including code authorship dispute, plagiarism detection [1], app clone detection [2], software forensics [3], and malware analysis [4], [5]. Taking malware analysis [6] in Android as an example, the cost of making and evolving malware is relatively low due to automated code generation techniques [7] and amounts of reusable code. As a consequence, manually analyzing these malware samples becomes a laborious and tedious task when anti-malware tools [8], [9] cannot effectively capture malicious behaviors inside. By applying authorship identification, security analysts can determine the author of malware and further infer the contained malicious behaviors and attack targets.

Software developers usually leave personal and identifiable information on the code in terms of distinguishable programming habits. This information is the pivot for accurately identifying its author. However, it is very challenging due to the scarcity of datasets, evolving programming style, code obfuscation, etc [10]. The research on code authorship of modern software can date back to the 1980s, where Oman and Cook employed typographic characteristics to distinguish Pascal programs [11]. Subsequently, more stylistic features have been extracted and utilized for authorship identification. Lexical and syntactic features of source code, e.g., variable naming, layout style, and the use of data structure can deanonymize the authors of code [12]–[14]. Semantic features such as an abstract syntax tree, control flow graph, and program dependence graph [15], [16] are also found effective in authorship identification. However, these techniques cannot be fully applied in attributing authors of Android apps owing to the distinct development mode.

Android app is a combination of functional modules, and its code is written by following the development rules by either an individual or a team. Therefore, the first challenge comes from the code influence of the collaborative group. To be more specific, Android apps may be developed in teams, and the different modules can be designed by multiple authors. All the legacies from the teamwork dilute authors' programming styles. The second challenge is due to numerous reusable libraries for easing development difficulty [17]–[19]. For example, developers are prone to use advertisement libraries (e.g., *AdMob* and *Facebook*), social networks (e.g., *Twitter* and *Wechat*), development libraries like *okHttp* and *Google GSON*. According to an empirical study on 100 F-Droid apps, we found that only 8% of them were produced independently by developers without any third-party libraries. As a consequence, the introduction of third-party libraries certainly exerts a negative influence on authorship identification. Last, Android apps do not fully retain lexical and syntactic features of source code after compilation, constituting the third challenge. Besides, the use of PROGUARD during compilation

* the authors contributed equally to this work

can obfuscate source code drastically. The studies [20], [21] make the pioneering efforts to distill recognizable features like strings, used data structures, and statistics of methods and classes from an apk file. However, all of them fail to address all the three challenges as aforementioned. Hence, we aim at studying segmented pieces of the app code instead of the entire code to solve the challenge.

In this study, we propose a two-phased approach, termed as A^3IDENT, to identify the leading authors for Android apps which consists of *authorship decoupling* and *authorship identification*. The leading author is an Android developer who primarily implements the advertised functional model, i.e., the primary module [22]. In the first phase, we construct a package relation graph for a given app, and group all packages that are likely created by one single author based on three assumptions (see Section IV). We propose semantic and structural similarities to quantify the distance between packages of the app. The packages are further aggregated with the *Louvain* model [23]. In this manner, we can identify the primary module created by the leading author and address the one and second challenges. In the second phase, we extract three types of styling features and get rid of problems brought by the Android framework and obfuscation, such as overloaded methods, identifier rules, etc (see Section V). For these extracted features, we employ *word2vec* [24] to embed authors' profiles. At last, we use three machine learning models (i.e., Linear SVM [25], Random Forest [26], and Logistic Regression [27]) to determine the possible authors for the test apps. A^3IDENT is evaluated extensively on four datasets: F-Droid, benignware, malware, and obfuscated apps. As indicated by experimental results, A^3IDENT has achieved an accuracy of 96%, 98.9%, 82.7%, and 80.4%, respectively. By comparing an open-source authorship attribution tool APPAUTH [2], our approach makes a 3.4% improvement in authorship identification with an accuracy of 87.8%. In addition, we identify four types of external code that are mixed with the primary code during app development and compilation. The experiment on obfuscated apps proves that our approach stays a high accuracy in authorship identification against obfuscation.

Contributions. We have made the following contributions.

- We propose authorship decoupling in the granularity of package to group the correlated code as per its authors, which is never considered in prior research on Android. Based on this, we find that several classes have been integrated into the apk file during compilation, e.g., configuration classes and third-party libraries.
- For authorship identification, we extract three types of features from the primary module, i.e., dex-level, lib-level, and manifest-level features. Differently, we eliminate the influence of the Android framework of feature extraction and combine *TF-IDF* and word2vec techniques to embed the features. Three classifiers are then employed for identifying authorship.
- We implement an automated tool A^3IDENT which is extensively evaluated on 257 authors with their 15,666 apps. Authorship decoupling achieves an accuracy of 96.11% on 416 F-Droid apps, and authorship identification achieves a 92.5% accurate rate on average on the whole set. Our approach proves to be also effective in handling obfuscated apps with only a 7.2% reduction in accuracy.

II. BACKGROUND

A. Android app Authorship

Android developers compile their source code and other resource files, e.g., layout files, into an Android application package (APK) and deliver it into the Android platform. An apk file contains AndroidManifest.xml, *.dex, *.arsc, res directory, META-INF directory, etc. Developers are required to sign Android apps with their own certificates. The signing certificate is stored in the "META-INF" folder. Android signatures guarantee the integrity of apps and prevent tampering and replacement. Except for certificates are exposed in the Android source code published by AOSP [28] and the private key is somehow leaked [29], developers' certificates are confidential and cannot be known by others [30]. Hence, apps signed with the same certificate are supposed to be created by the same developer.

The study of authorship identification stems from the field of literature, which aims to identify the author of a controversial text based on his/her unique linguistic styles (e.g., verb, vocabulary, sentence length). It is also very significant in the domain of Android. For example, authorship identification can assist the confirmation of adversaries' identity behind zero-day attacks or variants in the wild. Generally, it elicits a set of characteristics as fingerprints to quantify the stylometry (e.g., programming styles, and naming conventions) of malware authors. Different from other software systems, one apk file is most likely composed of multiple pieces of code from different authors and compilation. As a result, we have to separate the primary code from external code.

B. Louvain model

Louvain model is a graph clustering algorithm based on multi-level modularity optimization to identify communities from a large network, which we use for authorship decoupling (see Section IV-C). The nodes and the weight between nodes as input, Louvain model outputs the cluster to which each node belongs. Given a weighted graph, its goal is to maximize the modularity guided by Q as follows [23]:

$$Q = \frac{1}{2m} * \sum_{ij} \left[A_{i,j} - \frac{k_i * k_j}{2m} \right] * \sigma(C_i, C_j) \qquad (1)$$

where Q denotes the current status of modularity, A_{ij} represents the weight between node i and node j, k_i represents the weighted sum of all edges connected with node i, C_i is the cluster number of node i, and $\sigma(C_i, C_j)$ represents that if node i and j are in the same cluster, the value is 1, otherwise the value is 0. At first, each node is treated as an independent category, and Louvain algorithm is divided into two steps:

Step 1 Traversing all nodes of the graph, grouping similar nodes together, and labeling them until the cluster of all nodes does not change.

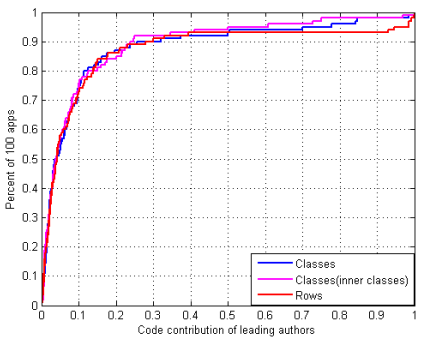

Fig. 1. The distribution of code contribution of authors.

Step 2 Re-initializing the graph and merging the nodes from the same cluster into a supernode. The supernode contains a self-connected edge whose weight doubles the sum of weights of all edges in the original. The weight between supernodes is the sum of weights of the edges whose connected nodes locate different clusters. Then, repeat the first step until the modularity Q does not change.

III. Authorship Analysis and System Overview

In this section, we conduct an empirical analysis of Android apps and then present the system overview of A^3IDENT.

A. Component Analysis

A large number of apps have structural complexity and it is not realistic to analyze all the code of apk for authorship attribution. Besides, due to multi-module development, the code in an app may be attributed to an individual or a team with cooperation. Each team develops its own module code, the coupling between modules is poor, and the aggregation within modules is high. Even worse, it is observed that the proportion of code written by the author is small. As shown in Fig. 1, among the 100 apps randomly selected from F-Droid, this proportion is less than 30% in about 90% of apps. Due to some limitations, LIBSCOUT [17] and LIBD [18] can only detect a subset of third-party libraries from apps. Consequently, it is non-trivial to classify a mix of code to specific authors. Prior studies [20], [21] fail to consider this phenomenon in their approaches although they also achieve good results.

Motivated by the above reasons, our goal is to develop a new authorship identification approach. Before authorship identification, we establish the correlation between packages for the app analyzed, and divide packages into independent modules. Among these modules, there is a module that includes MainActivity and it can be easily identified by querying AndroidManifet.xml. As the entry point of activities, MainActivity is the core class of an Android app and implements the main functions by means of function calls, ICCs, etc. Therefore, according to [22], the module where this activity resides is regarded as the primary module designed by the leading author and can be used for authorship identification.

TABLE I
THE RESULT OF GRANULARITY COMPARISON EXPERIMENT

Granularity	Precision	Recall	F1-score	Accuracy	Time(s)
Class	94.43%	97.01%	94.43%	94.94%	5113.29
Package	96.00%	97.69%	95.99%	95.81%	1357.46

B. System Overview

We propose a framework A^3IDENT with two major phases to identify the primary module of given apps and generate the representations with regard to authors' programming styles for authorship attribution. Fig. 2 presents the overview of our model. Given an app, it proceeds as follows:

- **Authorship decoupling**: We treat Java packages as units of analysis and divide them into different modules for each app. We create a package relation graph for a given app in terms of call relationship, inheritance relation, and ICC connection. We employ *package aggregation* to group all packages created by the authors and design semantic and structural similarities to calculate the weight of each pair of them. Last, we utilize the Louvain model and associated weights between packages to divide packages into modules. The primary module can be determined according to where MainActivity is located.
- **Authorship identification**: We extract three types of features from the primary module of an app and make use of the TF-IDF algorithm to identify the important sequences of these features. Then, we form feature vectors based on the *word2vec* model and select three machine learning models as supervised classifiers to predict a potential author.

Granularity of package. In this study, we take *packages* rather than *classes* as atomic units of module since classes under the same package, especially with multiple levels, are likely written by the same author. To prove this hypothesis, we extracted 100 authors from the F-Droid dataset and randomly selected one app from each of them. Then, according to the method in Section IV, we carry out the authorship decoupling with class granularity and that with package granularity. Table I shows the comparison result, and it is observed that authorship decoupling in the granularity of package achieves higher recall, accuracy, and F1-score than that with class granularity. Moreover, it costs only one-fourth time.

IV. Authorship Decoupling

An Android app is composed of several functionality-independent modules [22]. Similarly, the app oftentimes integrates diverse code from multiple developers [10], [31], including third-party libraries, the integrity of Android SDKs, and repackaged code. Therefore, we propose authorship decoupling to attribute part of code to specific developers.

A. Package Relation Graph

Different from functionality decoupling, authorship decoupling clusters code in terms of authors rather than functionality. Since one author can develop several functions that may not have explicit semantic relationships (e.g., call relationship)

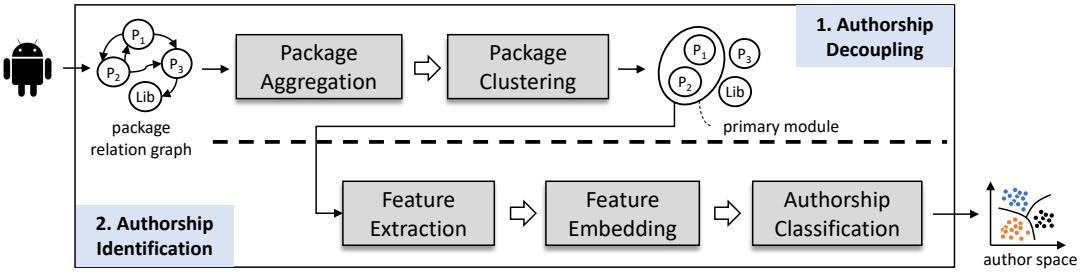

Fig. 2. System overview of A^3IDENT

in between, we propose several coarse-grained strategies to distinguish code developed by varying authors. Without loss of generality, we propose a package relation graph to represent an Android app as follows.

***Definition** 1:* An Android app can be represented as a weighted directed multigraph $G = (V, E, \phi)$, where V is the set of packages in the app, E is the set of edges between V, and ϕ is authorship function that $\phi(u)$ represents the author for package u.

In the graph G, we use an edge e, where $e = (u, v) \wedge u, v \in V$, to express the semantic relationships between two packages. Here we consider three types of semantic relationships: *call relationship*. If there is an invocation from one construct (e.g., method or variable) in package u to another method or variable in package v, we treat that there is a call relationship between u and v; *inheritance relationship*. If one class in package u is inherited from another class in package v, it implies that u and v have an inheritance relationship; and *ICC connection* reveals a type of connection between two packages u and v if one component class in u has an Inter-Component Communication with another class in v.

B. Package Aggregation

To group all packages created by the same authors, we make three assumptions in consideration of app development. First, packages defined in known third-party libraries are regarded as being owned by one author. It is rational since third-party libraries are developed by single organizations and should be separated from the primary module in apps. Second, components defined in AndroidManifest are probably involved with one author. AndroidManifest defines all essential information about an app, of which the components are Activity, Service, Broadcast Receiver, and Content Provider for underlying apps' functionality. As such, these components are created by the same author of high probability. We investigated 100 apps from dataset F-Droid and found that 89% of apps had components in the same module. Third, vertices in a circle are probably created by one single author [32]. Authors in one app have asymmetric knowledge, as the caller of third-party libraries is aware of all exposed interfaces by libraries, however, the callee does not know the caller's interfaces at all. Therefore, if there are bidirectional relations between vertices u and v (i.e., u and v are in a circle), it is most likely that u and v are created by the same author. We take an app called "MinCal Widget" as an example. Its package structure is shown in Fig. 3. We filter out the third-party packages "Landroid/*" and "Landroidx/*", and generate a directed graph G with the remaining ones. Then, the components declared in AndroidManifest belong to the same author. All the components of this app are located in the package"minimalcalendarwidget" and "activity", so the two packages belong to the same author. Then, we look for circles and find that packages "minimalcalendarwidget", "activity", "external", "receiver", "service" are in the same circle. As such, the four packages are generated by the same author.

Based on the above assumptions, we perform an algorithm (as Algorithm 1) to aggregate packages by the same authors. In the beginning, we assign a unique id to each package as a distinguishable author (lines 1-2). We recognize all contained libraries in this app (line 3), and treat the libraries within the same author (lines 4-6). We retrieve all the components defined in AndroidManifest, and re-assign the same author id to these packages from line 7 to 9. Then we take advantage of the depth-first search algorithm to find all circles in the graph G at line 10-11. More specifically, for the last node v_n passed to the function DFS_CIRCLE_DETECT, we traverse all the outgoing edges to examine whether there is a backtracking edge to previously visited nodes. A circle is detected if found (line 14), and we determine that all the packages in a circle are created by the same author. Otherwise, a recursive process is performed at line 18. At last, ϕ is updated for the app.

C. Package Clustering

Part of the same authors' packages have been found in the previous steps. But there are still some packages that cannot be glued with the assumptions. Hence, we cluster packages in this section to cope with the rest. To this end, we develop two weights for distance between packages in the graph.

Semantic Distance. As there are three types of semantics (i.e., call relationship, inheritance relationship and ICC connection) between two packages. Given an edge $(u, v) \wedge \phi(u) \neq \phi(v)$, we use n^c_{uv}, n^h_{uv} and n^i_{uv} to represent the number of invocations, inheritance cases, and ICC links, respectively. Hence, the semantic distance can be computed as:

$$dist(u, v) = \frac{1}{n^c_{uv} + n^h_{uv} + n^i_{uv}} \tag{2}$$

Noted that, $dist(u, v)$ is assigned with the maximal value between u and v. For a relation graph, we employ Floyd's

Algorithm 1: Package Aggregation

Input: $G = (V, E, \phi)$: an Android app
Output: ϕ: authorship function

```
for u in V do
    φ(u) = uniq_id; //assign unique author id to nodes
libs ⊂ V ← identify contained libraries;
for lib in libs do
    for u, v in lib do
        φ(u) = φ(v);
comps ← retrieve components in AndroidManifest;
for u, v in comps do
    φ(u) = φ(v);
for u in V do
    DFS_CIRCLE_DETECT(⟨u⟩);
Function DFS_CIRCLE_DETECT(⟨v_0, v_1, ...v_n⟩):
    for u in {u|(u, v_n) ∈ E} do
        if φ(u) ≠ φ(v_n) ∧ u ∈ ⟨v_0, v_1, ...v_{n-1}⟩ then
            for v_i in ⟨v_t, ...v_n⟩ where v_t = u do
                φ(v_i) = φ(u);
        else if φ(u) ≠ φ(v_n) then
            DFS_CIRCLE_DETECT(⟨v_0, v_1, ...v_n, u⟩);
```

algorithm [33] to calculate the shortest path between every two packages and then update $dist(u, v)$ with their Floyd distance. Based on that, we propose the following to represent the correlation between u and v.

$$corr(u, v) = e^{-min(dist(u,v),dist(v,u))} \tag{3}$$

The larger the value of $corr(u, v)$ is, the greater the correlation between package u and package v.

Structural Distance. Package structure can also aid in authorship decoupling. As shown in Fig. 3, there are two packages "cat.mvmike.minimalcalendarwidget.external" and "cat.mvmike.minimalcalendarwidget.status". They are likely created by the same author considering the long common prefix of their names. Therefore, we put forward structural features for authorship decoupling. We calculate the structural similarity based on the nearest common parent (NCP) between every two packages. Given two packages u and v, their structural similarity can be computed as:

$$struc(u, v) = \sum_{i=1}^{n} \frac{1}{i} \tag{4}$$

Where n is the depth of the NCP of package u and v. The app name is defined as the root node and its depth is one. Noted that if u is the parent of v, the NCP of u and v is u.

According to the semantic and structural similarities, we compute the normalized probability of package u and v being under the same author as:

$$sim(u, v) = max(nor(corr(u, v)), nor(struc(u, v))) \tag{5}$$

TABLE II
SUMMARY OF DIFFERENT TYPES OF STYLOMETRIC FEATURES

Category	Name	Description
Dex	Identifier Name	Identifiers of classes, fields, and methods
	API Calls	The sequence of APIs invoked in a class
	Instructions	Instruction sequences from .dex file
Lib	Library name	The name of third-party libraries
Manifest	component	Naming rules of activities, services, providers, and receivers
	uses-feature	External hardware/software features

where $nor(\cdot)$ is a normalization function via *Min-Max Feature Scaling*. The degree of association between packages can be computed via $sim(u, v)$. To evaluate its effectiveness, we propose another combination method as $\alpha * corr + (1 - \alpha) * struc$, where α represents the proportion of two similarities. We randomly selected 100 apps from dataset F-Droid and measured the accuracy in authorship decoupling, respectively. The comparison result is shown in Fig 4, and it is seen that using the maximal value of these metrics works better than the parameterized.

At this point, we get the correlation weight between packages in a given app. Given two packages u and v, if $\phi(v) = \phi(u)$, the weight between u and v is one, otherwise, the weight between u and v is the value of $sim(u, v)$. Then, we use the Louvain [23] model to represent the tightness between packages in an app, and divide the packages into different clusters. Taking packages and their associated weights as input, Louvain outputs the module to which each package belongs. After that, we select the primary module where the activity is located as the module designed by the leading author, and then identify the code authorship of the primary module.

V. AUTHORSHIP IDENTIFICATION

In this section, we utilize the authors' distinctive writing styles to identify the most likely author for a particular workpiece from a group of candidates. We describe how to extract features, embed features, and make predictions.

A. Feature Extraction

Feature extraction is a key part of authorship identification, and its goal is to extract features with author's distinct writing style. Instead of traditional code authorship attribution, we should not only extract features from .dex file and consider other resource files, e.g., AndroidManifest.xml. Note that we only extract the source code features from the primary module identified in Section IV and exclude the influence of Android APIs. In order to better reflect the characteristics of features as fingerprints, the features we analyzed are described as follows:

- **Dex-level features**. The .dex file contains all the Java/Kotlin code of an app, and we extract three types of string features: *identifier names*, including class name, method name, and field name. To eliminate the influence of the Android framework, we do not take into account overloaded methods, onCreate and onPause; *instruction sequence*, that is how

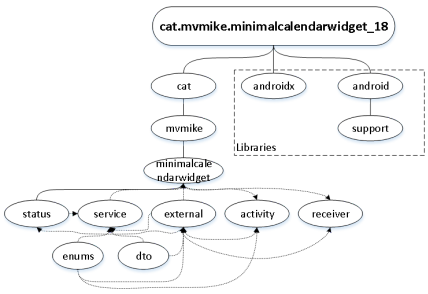

Fig. 3. An illustrative example for package structure of one app

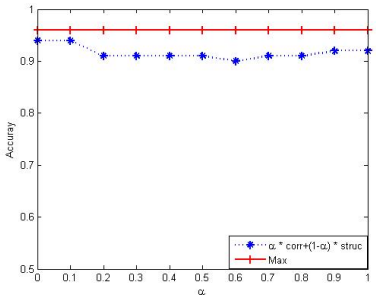

Fig. 4. Different experimental accuracy of $sim(u, v)$ for authorship decoupling

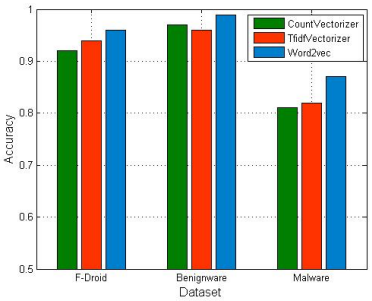

Fig. 5. Detection accuracy for authorship identification, under three vectorization methods

authors write code to implement their functionality. It is worth mentioning that this is literal sequences while not sequences in a control flow graph, as they preserve authors' programming habits; *the use of Android APIs*, which reveals to some extent authors' familiarity and preferences in Android development. For example, to build an HTTP connection, authors may use HttpURLConnection or Apache HttpClient offered by *Apache*.

- **Manifest-level features**. Each app contains an AndroidManifest.xml file, which defines the package name, components, permissions, etc. Different components serve for different purposes. The uses-feature declares external hardware or software features at runtime. These configurations are related to the functionality of the app, and authors tend to reuse the same setting when publishing the same type of apps. Hence, the naming of each component is related to the author's naming rules and functionality provided. Here we extract the names of activities, providers, services, broadcast receivers, and use-features from AndroidManifest.xml.
- **Lib-level features**. Third-party libraries are required to perform additional functions at runtime. Generally, there are several candidates for the desired function. For example, developers can integrate either *AdMob* or *Unity Ads* for advertisement, and either *Google Analytics* or *Crashlytics* for diagnosing crashes of apps. The selection of third-party libraries can reveal the habits and personality of app authors.

Table II summaries the features used for authorship identification. These features contain a lot of noise and are not suitable for analysis of large datasets. Hence, we use the TFIDFVectorizer [34] to extract key sequences of word-level strings from string features. TFIDFVectorizer constructs n-grams features from three sources, i.e., API calls, identifier names, and instructions. During the dex-level feature extraction process, we extract the three features in turn and use it to generate high-frequency sequences for vectorization. Then we tune proper hyper-parameters for TFIDFVectorizer. We set *max_features* as 50, *min_df* as 3, *n-grams* range from 3 to 5. This provides the best trade-off between accuracy and processing time.

B. Feature Embedding

A^3IDENT uses the *word2vec* model to process text vectorization for extracted features. As a distributed representation, *word2vec* uses low-dimensional dense vectors to represent the semantic information of words through text learning, which is a good measure to measure the similarity between words. To verify its effectiveness, we compare the accuracy of CountVectorizer [34], TFIDFVectorizer, and *word2vec* on the experimental results. These three methods are common methods of text vectorization. Among them, CountVectorizer converts the words into a word frequency vector, and then counts the number of times each word appears. TFIDFVectorizer converts the text into *tf-idf* feature vector. Fig. 5 shows the results of these three embedded techniques. In this experiment, we select F-Droid, benignware, and 2,900 malware apps as datasets, and extracted the same number of *top-k* features. We found that the performance of *word2vec* was a litter higher than the other two methods. In consequence, we choose *word2vec* for feature embedding in this study. At last, in order to improve the accuracy of prediction, we set *windows* as 3, *min_count* as 10, and set the maximum number of columns of the feature vector to 1,000.

C. Authorship Classification

The generated developers' fingerprints are used for the author prediction. The feature vectors generated in the previous stage as input data, the author attribution problem is seen as a supervised learning classification problem, which classifies unknown apps to their corresponding developers. As a comparison, we also investigate and build three typical supervised machining learning models for our classification tasks and evaluate their effectiveness. As our problem is a typical multi-classification problem, we choose support vector machine (Linear SVM), random forest, and logistic regression to measure the performance of each machine learning classifier.

VI. EVALUATION

In this section, we first propose four research questions to answer, and then introduce the experiment setting. We aim to address the following questions:

RQ1. How effective is authorship decoupling, and what extra-essential code resides in an apk file (see Section VI-B)?
RQ2. How effective of A^3IDENT in authorship identification, and what is the improvement by authorship decoupling (see Section VI-C)?
RQ3. How resilient is A^3IDENT to obfuscation (see Section VI-D)?
RQ4. Compared with APPAUTH, how effective is A^3IDENT in identifying the author of a given app (see Section VI-E)?

A. Experimental Setup

Dataset. To evaluate our methodology, we collected a total of 32,968 Android apps from various sources as shown in Table III. The following describes how we collect these apps:

- **F-Droid**. It is an open-source repository for free Android apps [35]. To date, there are thousands of apps maintained by F-Droid as well as their source code. We obtain 2,296 apps with source code via its API in total.
- **Benignware**. We obtained 1,672 whitelisted apps from ANVA [36]. ANVA is an industry alliance that is responsible for monitoring, detecting, and responding to network threats. Every year, it publishes a white list of Android apps that are thoroughly examined by 11 professional security assessment institutes.
- **Malware**. We collect 29,000 Android apps from the wild (including Google Play [37], ApkPure [38], Anzhi [39], etc) which are labeled by VIRUSTOTAL [40] as malware.
- **Obfuscated Apps**. We select a number of malware samples and obfuscate 10% of them per author as a test set via PROGUARD [41]. PROGUARD is used for string obfuscation and code shrinking. This dataset is used to evaluate how resilient our approach is to obfuscation.

As not all authors create plenty of apps (some of them only create one app), it raises the difficulty of fingerprinting their programming styles. Therefore, we employ a *least-apps* policy by which we only retain the authors having at least n apps. We set n with 3 for dataset F-Droid considering the relatively low app number owned by one author. For the other three datasets, n is set to 10. In addition, to enable a more accurate evaluation, we discarded 8,691 apps signed by the public certificates [30] and duplicate apps published in multiple markets. Hence, we obtain 492 apps from 164 authors for dataset F-Droid, 686 apps from 17 authors for dataset Benignware, and 14,564 apps from 100 authors for dataset malware. Obfuscated apps are created by sampling 2,900 malware apps, each author contains 100 apps.

Implementation. We implement A^3IDENT on top of several state-of-the-art tools. The decompilation of Android apps mainly relies on ANDROGUARD, for extracting authorship features, call relations, and inheritance relations. ICC connections are extracted via IC3-DIALDROID [42], and we re-allocate the connections from Java classes to their belonging packages. The classification task is fulfilled by building applications on top of Python library SKLEARN [43] and GENSIM [44]. All the evaluation experiments were conducted on the environment with 8-core i7 Intel CPU and 12GB of RAM.

TABLE III
STATISTICS OF EXPERIMENTAL DATASETS

Dataset	Total	Least-Apps Filtering		
		# Least Apps	Authors	Apps
F-Droid	2,296	3	140	416
Benignware	1,672	10	17	686
Malware	29,000	10	100	14,564
Obfuscated	2,900	100	29	2,900

TABLE IV
EFFECTIVENESS OF AUTHORSHIP DECOUPLING ON F-DROID APPS

Sample	Accuracy	Precision	Recall	F1-score
A^3IDENT	96.11%	97.35%	96.11%	95.70%
PIGGYAPP	81.18%	86.01%	81.87%	76.77%

Model configuration and metrics. For different research tasks, we use precision, recall, F1-score, and accuracy as metrics for comparison according to [45]. From RQ2 to RQ4, we group authors based on certificates and generate true results. We employ A^3IDENT to generate predicted results. We use 90% of the data as the training set and the remaining 10% as the test set. For every experiment, we perform 10-fold cross validation.

B. Authorship Decoupling Evaluation (RQ1)

As an apk file contains code from one or more authors, we propose an authorship decoupling technique to distinguish the primary code from the others. To evaluate its effectiveness, we first build the ground truth from dataset F-Droid, and then examine how accurately our approach decouples the authorship of code.

Building the ground truth. Since F-Droid hosts thousands of free Android apps and the corresponding source code, we retrieve the files in source code to establish a ground truth for authorship decoupling evaluation. In the ground truth, all the classes of a given app are divided into two parts, i.e., the primary module generated by the leading author and the non-primary module that contains the remaining classes.

Step 1 We first use ANDROGUARD to decompile an APK file to get its classes and MainActivity. Note that we do not consider inner classes, e.g., Test$1.class.
Step 2 We take advantage of "settings.gradle" to determine the primary module where MainActivity resides, and then we get the classes contained in the primary module. In an Android project, settings.gradle is a configuration file for subprojects, whose goal is to manage multiple modules. Each module name corresponds to its root directory. We can walk through all the .java and .kt files to determine which module they belong to.
Step 3 Last, we classify the classes that do not appear in the primary module as the non-primary module. Besides, we check whether any classes are not compiled into

the *.dex* file classes. This is reasonable because some test code has been retained in the published source code and it could negatively influence the reliability of our ground truth.

Based on the above method, we build the ground truth of 416 F-Droid apps for authorship decoupling.

Result analysis. After authorship decoupling, we get the primary module and classes it contains. The remaining classes are merged into the non-primary module. Therefore, it can be treated as a binary classification for a given app. We use metrics (e.g., precision) to evaluate the package decoupling performance of each app, and then calculate the average of all apps as average metrics. Moreover, we re-implement PIGGYAPP [22], which provides a method for module decoupling based on class inheritance, method calls, and etc. The comparison result is shown in Table IV. Compared to PIGGYAPP, our model gets much better results, e.g., accuracy is increased by 14.93% and F1-score by 18.93%. The improvement stems from higher quality features employed in this study, particularly the structural and semantic similarities. These features prove to be effective in module decoupling. In addition to authorship attribution, authorship decoupling can be used for code plagiarism, functionalities classification, and app version detection, with reducing noise interference.

In addition, on the premise of excluding the interference of inner class, we compare the classes of the same package in the source code and the apk file, and note that the classes of the APK package do not all exist in the source code. These classes are from four sources as follows:

- **Auto generation**. During compilation, an Android app automatically generates classes R, BR and BuildConfig. Files R, and BR record the resource information. The BuildConfig file records the configuration information of build.gradle.
- **Lambda expression**. Java 8 introduces *Lambda* expressions, a compact way to represent behaviors. Its use reduces the template code and makes the data flow processing logic clear. These generated .class files are found in the apk file, but not exist in the source code. For example, when analyzing the app "app.fedilab.nitterizeme_9", we found a series of classes like "Lapp/fedilab/lite/helper/-$$Lambda...".
- **Classes generated by the author instructions**. When writing code, developers may define a new class or upload classes to specified packages from the Internet in the statement, e.g., new STTFragment(). These files may be irrelevant to developers.
- **Different modules with the same package**. It is possible that the same package may contain different module classes. Taking the app "com.termoneplus_324" for example, it contains three modules–*samples*, *term* and *libtermexec*. However, modules *term* and *libtermexec* have the same package "Lcom/termoneplus". When the Android project is compiled into an APK, the classes under the package of these two modules are packaged into the same package, although they do not belong to the same module.

Fig. 6 depicts the proportion of the four causes mentioned above. The classes generated by the author's instructions are the most important factor of authorship decoupling error, accounting for 64%. In the future study, we should eliminate the noise effect of these four classes on authorship identification.

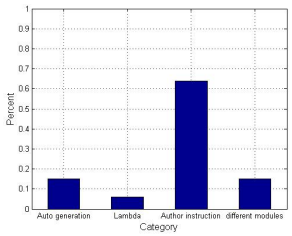

Fig. 6. The percentage of each type in the total errors in authorship decoupling

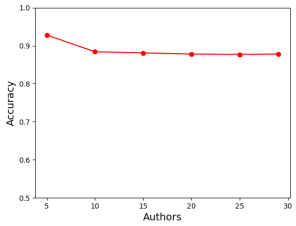

Fig. 7. The accuracy of authorship identification under different number of authors

C. Authorship Identification Evaluation (RQ2)

In this section, we evaluate whether A^3IDENT is effective in identifying the author of a specific piece of code.

Result Analysis. We use F-Droid, benignware, and malware datasets to evaluate the accuracy of authorship identification. As shown in Table V, for these three datasets, we get the accuracy of around 96%, 98.9%, 82.7%, respectively. Overall, our model provides high accuracy results in authorship identification. We also compare the performance of the three supervised machine learning models in authorship identification. We find that the accuracy of the three classifiers is similar. Linear SVM has gained a narrow lead in average accuracy but runs far longer than the other two classifiers. Random forest and logistic regression are more suitable for our model in terms of running time and effectiveness. Besides, when reviewing the package names and versions of apps in dataset F-Droid, we notice that the result is more like the detection of different versions of the same app. The considerably high accuracy enables A^3IDENT to be efficient in detecting highly similar code such as plagiarism detection.

In order to evaluate the model more comprehensively, we select random forest to compare the performance of authorship identification based on the primary module and the whole apk (without authorship decoupling). Table VI describes the results of the comparison. A^3IDENT achieves the accuracy of 95.7%, 99.0%, and 87.4% for datasets F-Droid, benignware, and malware samples, respectively. It is observed that authorship decoupling makes an increase of 4.6% in accuracy on dataset F-Droid, but only makes an improvement of 1.2% and 1.4% for benignware and 2,900 malware samples. The similar accuracy may be attributed that there are a number of apps created by the same author for only one module without third-party libraries. Besides, the result can also be caused by code clone among apps. We have investigated 10 authors that create at least 10 apps, and 12% of apps are found with a similar code structure. Besides, we further evaluate the impact of the number of authors on authorship identification. With this purpose, we used the obfuscated dataset with 29 authors. Each author has the same number of apps, and this excludes the

TABLE V
AUTHORSHIP IDENTIFICATION ACCURACY FOR DIFFERENT DATASETS

Classifier	F-Droid		Benignware		Malware	
	Accuracy	Runtime(s)	Accuracy	Runtime(s)	Accuracy	Runtime(s)
Linear SVM	96.5%	6.33	98.9%	1.62	83.1%	277.0
Random forest	95.7%	1.44	99.0%	0.23	82.4%	35.56
Logistic regression	95.8%	1.78	98.8%	0.18	82.6%	6.94

TABLE VI
ACCURACY OF AUTHORSHIP IDENTIFICATION IN THREE DATASETS.

Method	F-Droid	Benignware	Malware
Primary code	95.7%	99.0%	87.4%
Whole apk	91.1%	97.8%	86.0%

TABLE VII
EVALUATION ON OBFUSCATION RESILIENCY IN AUTHORSHIP IDENTIFICATION.

Sample	Precision	Recall	F1-score	Accuracy
Non-obfuscated	85.0%	86.0%	85.0%	87.6%
obfuscated	80.4%	81.4%	80.2%	80.4%

interference of different authors having different numbers of apps to the result. The experimental result is shown in Fig. 7. We find that the accuracy of authorship decreases slightly with the increase of author number. Overall, accuracy does not change significantly. This result shows that our model is not disturbed by the number of authors, and has strong stability.

D. Obfuscation-resilient Analysis (RQ3)

In this section, we evaluate whether A^3IDENT is resilient to obfuscation in Android apps. Considering the effectiveness and running time, we choose the random forest for the comparative experiment on the obfuscated dataset. For this dataset, we randomly select 90% as the training set and the remaining 10% as the test set for each author. For the test set, we use PROGUARD to obfuscate apps. There are three steps for PROGUARD to obfuscate Android apps. Initially, unused code (e.g., unused methods and variables) is removed to shrink the size of code. Then, Java bytecode optimization is performed. Finally, it performs name obfuscation, i.e., it obfuscates code by renaming variables, classes, and methods into short, random, meaningless words. Note that we do not perform obfuscation on Android APIs. After obfuscation, we take the obfuscated apps and their non-obfuscated apps as the test set, respectively, and measure the resulting accuracy of authorship identification.

Result analysis. Table VII shows the experimental results of the obfuscated and non-obfuscated test sets, with the accuracy of 87.6 and 80.4%, respectively. Compared with the non-obfuscated apps, the accuracy of the obfuscated apps decreases slightly, but not significantly. That's because manifest-level features are not obfuscated. Moreover, we investigate five apps and their obfuscated apps. We compare instruction sequences of each pair of apps and find that they are less affected by code obfuscation. The result suggests that A^3IDENT is somewhat resistant to code obfuscation.

TABLE VIII
AUTHORSHIP IDENTIFICATION COMPARED WITH APPAUTH.

Sample	Precision	Recall	F1-score	Accuracy
A^3IDENT	88.4%	87.6%	87.1%	87.8%
APPAUTH	86.4%	84.5%	84.2%	84.4%

E. Comparison with APPAUTH (RQ4)

To validate the performance of A^3IDENT more fully, we compare it with APPAUTH in terms of authorship attribution. APPAUTH is an Android authorship detector and a novel learning-based approach for predicting the authorship of app clones. We select a dataset of 2,900 non-obfuscated apps and then use the two tools in turn. Table VIII presents the performance of both APPAUTH and A^3IDENT. Compared with APPAUTH, our approach raises a 3.4% improvement to identification with the accuracy of 87.8%. Therefore, our two-phased approach proves to be effective in Android authorship attribution.

VII. THREATS TO VALIDITY

Internal threats. First, our authorship decoupling heavily relies on the extraction of semantic relationships between Java/Kotlin packages. Tool and technical limitations can lead to the incorrect (or missing) relationships between pairs of classes. In this study, we use IC3-DialDroid to build ICC connections, and ANDROGUARD to identify call relations which may miss some implicit connections due to, for example, implicit Intent, reflection, and callbacks. Besides, some relations can only be discovered during execution, e.g., late binding. Second, not all components in an app are generated by the same author, e.g., repackage. This can affect module decoupling and the determination of the primary module. All experiments in this work were performed under the assumption that all code in the package was generated by the same author and that the primary module is where MainActivity resides. Android architect may also initialize its UI architecture and other developers develop its content part. The Android architecture has a little bit to do with the author's writing habits. Last, the elimination of legacy code from the Android framework is mainly based on a pre-defined list, which may be influenced by different Android SDKs or platforms. In future research, we intend to further refine our authorship decoupling approach, and develop a more robust approach to identify and track the legacy code of the Android framework.

External threats. External threats are more from the test datasets. First, Our test apps have a certain number of cloned apps, which may be caused by version upgrade. Although this cloned code is also attributed to the same author, it causes that any code clone detection technique can perform well. We have measured the influence of author number to authorship identification, as we think if there are more authors, the similarity of their code probably increases, raising the difficulty of authorship identification. According to the result in Section VI-C, only 5% drop occurs along with the increase of author number. However, it may be not able to represent the massive number of Android developers in the wild. Second, class imbalance may influence our result. For example, in dataset ANVA, there is one author that creates 103 apps, while another author has 17 apps. However, our approach proves to be performing very well in the other datasets. Considering the above, we plan to collect more comprehensive apps and further compare them with relevant tools such as code clone detection and authorship identification.

VIII. Related Work

A. Android Static Analysis

Libraries Analysis. Li *et. al* [18] developed a prototypical tool called LibD, which utilizes code dependencies of Android apps to detect candidate libraries and handles multi-package third-party libraries in the presence of name-based obfuscation. Derr *et. al* [17] devised a light-weight and effective approach to detect third-party libraries that is resilient to common obfuscation techniques and capable of pinpointing exact library versions. Zhang *et. al* [46] presented a novel library detection tool named LibID, which is more resilient to code shrinking and package modification than state-of-the-art tools.

ICCs analysis. Octeau *et. al* [47] designed an Android ICC analysis tool named IC3 to extract information related to the ICC sources, sinks and intent-based communication channels. Li *et. al* [48] proposed a taint analyzer named IccTA to detect privacy leaks among components in Android applications. Bosu *et. al* [49] developed DIALDroid, an Android security tool for analyzing data flows between sensitive applications based on ICC. It leverages the relational database for a scalable matching of ICC entry and exit points, and fast analysis.

B. Code Authorship Attribution

Oman and Cook [11] conducted a statistical analysis of authors' programming style and extracted typographic characteristics such as indentation, spacing, comments, and upper/lower cases of letters. Based on these characteristics, a clustering method is performed to successfully group the source code for several algorithms in textbooks as per authors. Frantzeskou *et. al* [13] proposed a SCAP (Source Code Author Profiles) approach based on byte-level characteristics. In particular, they extracted n-grams from source code including all non-printable characters, and used the highest n-grams as the marker for authors. Kalgutkar *et. al* [20] identified from Android apks all present strings including strings referenced by identifiers, string components, and strings in XML files. Then they built n-grams and leveraged SVM (Support Vector Machine) to classify the authors of code. Caliskan *et. al* [50] identified the stylistic features from source code, i.e., lexical features, layout features and syntactic features. With random forest, they outperform the accuracy of authorship attributions than prior works. Meng *et. al* [51] presented four types of features i.e., instruction, control flow, data flow and context features. With a proposed joint classification model, they managed to identify the author for a single basic block in binary. Abuhamad *et. al* [12] proposed a Deep Learning-based Code Authorship Identification System (DL-CAIS) for code authorship attribution. DL-CAIS proceeds with TF-IDF representation using deep neural networks and an author classifier based on Random Forest. *Our study contributes to the contemporary authorship identification twofold. First, we propose authorship decoupling to separate code by different authors from the third-party libraries, compilation, Android framework, etc. To the best of our knowledge, none of the works in the Android field have considered this problem. On the other hand, we empirically analyze the noise brought by the Android system including the inherited classes and methods, invoked APIs, and try to solicit stylometric features that better characterize one author rather than the Android system. These can, from both theory and practice, benefit the researchers and practitioners in authorship identification of Android apps.*

IX. Conclusion

In this paper, we propose the A^3IDENT approach, which consists of two steps for authorship attribution. First, we conduct authorship decoupling through constructing package relation graph and cluster packages into different modules. In such a manner, the primary module can be further determined by the location where the entry point activity resides. In the course of authorship identification, we distill three types of features, retaining the stylometric features of app authors while removing imprints by the Android framework. An embedding algorithm and three types of machine learning algorithms are conducted to identify the leading authors of given apps. Our approach has been evaluated in four datasets, and the result shows that A^3IDENT can effectively identify authorship with an average accuracy of 92.8% with Linear SVM, 92.4% with Random Forest, and 92.4% with Logistic Regression. It also proves that A^3IDENT performs still effectively on obfuscated code. Compared to an open-source authorship attribution tool, our approach makes a 3.4% improvement in accuracy for authorship identification.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers and shepherd for their valuable comments for this paper. This work has partially been sponsored by the National Natural Science Foundation of China (No. 61872262, 61702045, 61572349, 61902395, and U1836211) and National Key R&D Programmes of China (No. 2019AAA0104301). Xiaohong Li and Weimin Ge are the corresponding authors.

REFERENCES

[1] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories," *Softw. Pract. Exp.*, vol. 37, no. 2, pp. 151–175, 2007.

[2] 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. 141 850–141 867, 2019.

[3] R. C. Lange and S. Mancoridis, "Using code metric histograms and genetic algorithms to perform author identification for software forensics," in *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, H. Lipson, Ed. ACM, 2007, pp. 2082–2089.

[4] R. Chouchane, N. Stakhanova, A. Walenstein, and A. Lakhotia, "Detecting machine-morphed malware variants via engine attribution," *J. Comput. Virol. Hacking Tech.*, vol. 9, no. 3, pp. 137–157, 2013.

[5] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic modelling of android malware for effective malware comprehension, detection, and classification," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 306–317.

[6] G. Meng, R. Feng, G. Bai, K. Chen, and Y. Liu, "Droidecho: an in-depth dissection of malicious behaviors in android applications," *Cybersecurity*, vol. 1, no. 1, p. 4, 2018.

[7] M. Oltrogge, E. Derr, C. Stransky, Y. Acar, S. Fahl, C. Rossow, G. Pellegrino, S. Bugiel, and M. Backes, "The rise of the citizen developer: Assessing the security impact of online app generators," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 634–647.

[8] G. Meng, Y. Xue, M. Chandramohan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving android malware for auditing anti-malware tools," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, X. Chen, X. Wang, and X. Huang, Eds. ACM, 2016, pp. 365–376. [Online]. Available: https://doi.org/10.1145/2897845.2897856

[9] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang, "Auditing anti-malware tools by evolving android malware and dynamic loading technique," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 7, pp. 1529–1544, 2017. [Online]. Available: https://doi.org/10.1109/TIFS.2017.2661723

[10] 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. 3:1–3:36, 2019.

[11] P. W. Oman and C. R. Cook, "Programming style authorship analysis," in *Computer Trends in the 1990s - Proceedings of the 1989 ACM 17th Annual Computer Science Conference, Louisville, Kentucky, USA, February 21-23, 1989*, A. M. Riehl, Ed. ACM, 1989, pp. 320–326.

[12] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang, "Large-scale and language-oblivious code authorship identification," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 101–114.

[13] G. Frantzeskou, E. Stamatatos, S. Gritzalis, C. E. Chaski, and B. S. Howald, "Identifying authorship by byte-level n-grams: The source code author profile (SCAP) method," *IJDE*, vol. 6, no. 1, 2007.

[14] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. K. Katsikas, "Source code author identification based on n-gram author profiles," in *Artificial Intelligence Applications and Innovations, 3rd IFIP Conference on Artificial Intelligence Applications and Innovations (AIAI) 2006, June 7-9, 2006, Athens, Greece*, ser. IFIP, I. Maglogiannis, K. Karpouzis, and M. Bramer, Eds., vol. 204. Springer, 2006, pp. 508–515.

[15] X. Meng, "Fine-grained binary code authorship identification." New York, NY, USA: Association for Computing Machinery, 2016.

[16] L. Simko, L. Zettlemoyer, and T. Kohno, "Recognizing and imitating programmer style: Adversaries in program authorship attribution," *PoPETs*, vol. 2018, no. 1, pp. 127–144, 2018.

[17] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 356–367. [Online]. Available: https://doi.org/10.1145/2976749.2978333

[18] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: scalable and precise third-party library detection in android markets," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 335–346. [Online]. Available: https://doi.org/10.1109/ICSE.2017.38

[19] L. Li, T. Riom, T. F. Bissyandé, H. Wang, J. Klein, and Y. L. Traon, "Revisiting the impact of common libraries for android-related investigations," *J. Syst. Softw.*, vol. 154, pp. 157–175, 2019.

[20] V. Kalgutkar, N. Stakhanova, P. Cook, and A. Matyukhina, "Android authorship attribution through string analysis," in *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018, Hamburg, Germany, August 27-30, 2018*, S. Doerr, M. Fischer, S. Schrittwieser, and D. Herrmann, Eds. ACM, 2018, pp. 4:1–4:10.

[21] H. Gonzalez, N. Stakhanova, and A. A. Ghorbani, "Authorship attribution of android apps," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*, 2018, pp. 277–286.

[22] W. Zhou, Y. Zhou, M. C. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "piggybacked" mobile applications," in *Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013*, E. Bertino, R. S. Sandhu, L. Bauer, and J. Park, Eds. ACM, 2013, pp. 185–196.

[23] Louvain, "Louvain community detection," https://github.com/taynaud/python-louvain, May 2020.

[24] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method," *CoRR*, vol. abs/1402.3722, 2014.

[25] M. Pontil, S. Rogai, and A. Verri, "Recognizing 3-d objects with linear support vector machines," in *Computer Vision - ECCV'98, 5th European Conference on Computer Vision, Freiburg, Germany, June 2-6, 1998, Proceedings, Volume II*, ser. Lecture Notes in Computer Science, H. Burkhardt and B. Neumann, Eds., vol. 1407. Springer, 1998, pp. 469–483. [Online]. Available: https://doi.org/10.1007/BFb0054759

[26] A. Prinzie and D. Van den Poel, "Random multiclass classification: Generalizing random forests to random mnl and random nb," vol. 4653, 08 2007, pp. 349–358.

[27] J. Tolles and W. J. Meurer, "Logistic Regression: Relating Patient Characteristics to Outcomes," *JAMA*, vol. 316, no. 5, pp. 533–534, 08 2016.

[28] A. Oruganti, "Android platform," https://github.com/aosp-mirror/platform_build/, Nov. 2020.

[29] K. Allix, Q. Jérome, T. F. Bissyandé, J. Klein, R. State, and Y. L. Traon, "A forensic analysis of android malware - how is malware written and how it could be detected?" in *IEEE 38th Annual Computer Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21-25, 2014*. IEEE Computer Society, 2014, pp. 384–393.

[30] H. Wang, H. Liu, X. Xiao, G. Meng, and Y. Guo, "Characterizing Android App Signing Issues," in *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2019, 2019.

[31] 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, 2019.

[32] Z. Tang, M. Xue, G. Meng, C. Ying, Y. Liu, J. He, H. Zhu, and Y. Liu, "Securing android applications via edge assistant third-party library detection," *Computers & Security*, vol. 80, pp. 257 – 272, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404818311301

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[34] J. Han, M. Kamer, and J. Pei, *Data Mining: Concepts and Techqniues*, 3rd ed., 2012.

[35] "F-droid," https://f-droid.org/, Dec. 2019.

[36] "Anti network-virus alliance of china," https://www.anva.org.cn/, Feb. 2020.

[37] "Google play," https://play.google.com/store?hl=en, Dec. 2019.

[38] "Apkpure," https://apkpure.com/, Dec. 2019.

[39] "Anzhi," http://www.anzhi.com/, Dec. 2019.

[40] "Virustotal," https://virustotal.com/, Aug. 2019.

[41] Google, "Shrink, obfuscate, and optimize your app," https://developer.android.com/studio/build/shrink-code, Nov. 2019.

[42] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, 2017, pp. 71–85.

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, 2011.

[44] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[45] S. Easterbrook, J. Singer, M. D. Storey, and D. E. Damian, "Selecting empirical methods for software engineering research," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. Springer, 2008, pp. 285–311.

[46] J. Zhang, A. R. Beresford, and S. A. Kollmann, "Libid: reliable identification of obfuscated third-party android libraries," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 55–65.

[47] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. D. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 77–88. [Online]. Available: https://doi.org/10.1109/ICSE.2015.30

[48] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 280–291. [Online]. Available: https://doi.org/10.1109/ICSE.2015.48

[49] A. Lee, J. C. Carver, and A. Bosu, "Understanding the impressions, motivations, and barriers of one time code contributors to FLOSS projects: a survey," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 187–197. [Online]. Available: https://doi.org/10.1109/ICSE.2017.25

[50] A. C. Islam, R. E. Harang, A. Liu, A. Narayanan, C. R. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 255–270.

[51] X. Meng, B. P. Miller, and K. Jun, "Identifying multiple authors in a binary program," in *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds., vol. 10493. Springer, 2017, pp. 286–304. [Online]. Available: https://doi.org/10.1007/978-3-319-66399-9_16