Authorship Attribution of Android Apps

Hugo Gonzalez
Polytechnic University of San Luis Potosi
San Luis Potosi, SLP, MX
hugo.gonzalez@upslp.edu.mx

Natalia Stakhanova
Faculty of Computer Science, University of New Brunswick
Fredericton, New Brunswick, CA
natalia@unb.ca

Ali A. Ghorbani
Faculty of Computer Science, University of New Brunswick
Fredericton, New Brunswick, CA
ghorbani@unb.ca

ABSTRACT

Since the first computer virus hit the Advanced Research Projects Agency Network (ARPANET) in the early 1970s, the security community interest revolved around ways to expose the identities of malware writers. Knowledge of the adversarial identities promised additional leverage to security experts in their ongoing battle against those perpetrators. At the dawn of computing era, when malware writers and malicious software were characterized by the lack of experience and relative simplicity, the task of uncovering the identities of virus writers was more or less straightforward. Manual analysis of source code often revealed personal, identifiable information embedded by authors themselves. But these times have long gone. Modern days malware writers extensively use numerous malware code generators to mass produce new variants and employ advanced obfuscation techniques to hide their identities. As a result the work of security experts trying to uncover the identities of malware writers became significantly more challenging and time consuming.

To gain insight into the identity of an adversary, we turn our attention to authorship attribution research, which offers a broad spectrum of techniques for identifying an author of a document, based on the analysis of an author's writing style. Equipped with these methods, we explore attribution of Android binaries and the role of features related to the development process on the determination of Android binary authorship.

Within this context, we propose an incremental approach to perform authorship attribution of Android apps. First to a set of known authors and then the generation of new profiles for unknown apps. We assess the effectiveness of our approach on several sets of malicious and legitimate Android binaries produced by actual developers, as opposed to using artificially created authors' data. We achieve 97.5% accuracy on these authors' data. We further evaluate our approach on more than 131,000 apps collected from various sources including 10 different markets around the globe.

CODASPY '18, March 19–21, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5632-9/18/03…$15.00
https://doi.org/10.1145/3176258.3176322

CCS CONCEPTS

• **Security and privacy** → **Malware and its mitigation**; **Mobile and wireless security**;

KEYWORDS

Android, authorship attribution, suspicious authors

ACM Reference Format:
Hugo Gonzalez, Natalia Stakhanova, and Ali A. Ghorbani. 2018. Authorship Attribution of Android Apps. In *CODASPY '18: Eighth ACM Conference on Data and Application Security and Privacy, March 19–21, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3176258.3176322

1 INTRODUCTION

Since the appearance of the first computer virus, the security community interest revolved around ways to expose an identity of an adversary. In the early days this was often possible due to relative simplicity of malicious software and inexperience of malware writers. Manual analysis of a code often revealed personal, identifiable information embedded by authors themselves [20]. However, with the extensive use of advanced obfuscation techniques and wide availability of malware code generators that allow malware authors to rapidly produce mass numbers of new variants, this process became significantly more challenging requiring advanced methodologies. These methods found in authorship attribution research, are referred to as stylometry. Well-established in social science, it offers a broad spectrum of techniques aiming to characterize an author of a document given a set of textual features that quantify an author's writing style [22]. The underlying assumption of attribution is an existence of an inherent distinctive writing style, unique to an author and easily distinguishable among others. Quantified representation of this style can be viewed as a fingerprint.

One of the main difficulties in the field lies in compiling such a fingerprint that provides efficient and accurate characterization of an author style. In the traditional setting, authorship attribution relies heavily on information that allows deeper linguistic analysis of author's works (e.g., richness of vocabulary, tense of verbs, semantic analysis of sentences). In software field, emphasis is mostly put on surface characteristics such as variable naming, program layout, and spacing, that reflect textual nature of source code [5]. Such approach is merely dictated by the nature of the field that in many cases fails to provide original source code of software (e.g., malware analysis, commercial software theft) leaving researchers with its binary representation. Unfortunately, such binary code

retains very few of the surface characteristics. As a result in the recent years a few studies shifted its focus towards analysis of program binaries [7, 18, 19],

This shift was also driven by practical objectives. Such digital fingerprint of an author retrieved from a malicious program binary can potentially serve as a single signature for attributing suspicious executable to that author. From a practical perspective, the immediate benefit of malware author attribution is clear: instead of detecting every malware strain using narrow specialized signatures, we could effectively characterize all malware variants generated by its author. This approach allows us to significantly reduce the computational resources required for malware detection.

In this work, we explore this angle focusing specifically on mobile domain. Just in the first quarter of 2017 G-data labs reported registering around 8,400 new malware samples daily [16]. They estimated that 3.5 million of new malware samples will be discovered this year. In their analysis from 2016, Kaspersky Labs [15] reported a big grown in Banking malware and trojan-ransomware that are able to bypass security mechanisms in GooglePlay bouncer and new versions of Android.

In this study, we propose an approach for authorship attribution of Android apps using a set of features related to author's decision over an application development process, specifically, metrics about the structure of the app extracted from .dex files, such as number of employed methods, classes, fields, strings, etc. We specially focus on the usage of data structures and opcodes associated with data structures manipulations. Data structure features were among the basic set of programmer's attribution measures proposed in fundamental work of Spafford et al [21] for attribution of binary code.

We validate our approach on a set of manually collected Android applications published from 37 authors in different markets. We further evaluate it with over 30,000 apps from known different malicious sources, official market and eight third-party markets.

Finally, we perform a large-scale study of over 160,000 Android applications from official Google market and Virus Total stream service.

The remainder of the paper discusses the related work in Section 2. It introduces features employed in Section 3, presents the details of proposed approach in Section 4. Data employed is described in Section 5. Validation and evaluation results are in Section 6. Section 7 concludes the paper.

2 RELATED WORK

The problem of program authorship attribution is not new. Its feasibility has been shown in a small pilot study by [14] and has since revolved around the idea of attributing source code through various characteristics of a programmer style [11]. The consistency of programming style formed a foundation of software author attribution [11]. Recently, Burrows et al [4] systematized techniques for attributing source code. One of the largest focuses in this context is plagiarism detection [5, 23]. Since source code is available, such detection mostly entails analysis of syntactical features (e.g., format alterations, renaming, control replacement). However, due to the lack of malware source code, these features have limited applicability beyond the plagiarism domain.

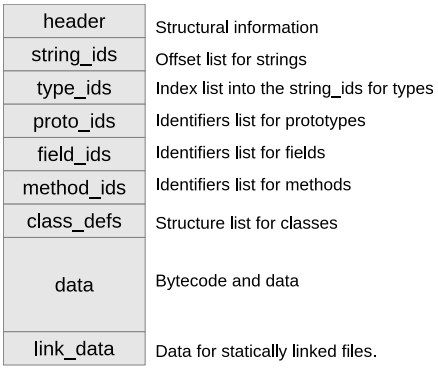

Figure 1: The structure of a .dex file

In malware analysis, the majority of research focused on analysis of binaries, e.g., behaviour of the software [24], API calls [6], and control flow [12]. Although the primary focus of these approaches was extraction of 'software birthmark', i.e., a combination of unique software characteristics, the results of these studies can be complementary to author attribution and should be further explored in this context.

The problem of binary attribution has been explored by Rosenblum et al. in a series of studies [18, 19] that investigated tool-chain provenance. The studies confirmed feasibility of automating a discovery of details that characterize the production process of a given binary (e.g., the compiler family, versions, optimization options and source languages). This approach was later extended by Alrabaee et al. [1]. The authors employed a multilayer approach integrating syntax-based and semantics-based features. Binary analysis was also explored in attribution of metamorphic malware to malware generation engines by Chouchane et al. [7].

Although all these results showed high accuracy of attribution among well defined engines/tool-chain components, the question of applicability of these results to extracting unknown developers' fingerprints remains. We propose a framework to attribute Android apps to a known group of authors and to unknown groups.

3 BINARY ATTRIBUTION

One of the first cases of authorship attribution over malware binary code for forensics purposes was offered by Spafford and Weeber in 1992 [21]. Spafford and Weeber believed that every programmer has its own unique style. Investigating a security breach, this authors connected the manual writing recognition used by law-enforcement to identify people, with the task of analysis of residual code in a security incident to identify an adversary. As a result of this study, several groups of features for binary code attribution were proposed, among

them are features related to programmer's style and expertise (e.g., presence of error handling) and features related to the development process (e.g., employed data structures, systems calls, tools).

The tell-tale signs of tools indicating the origin of executable have been successfully explored in previous studies [7, 9]. In this work we also focus on the features related to the development process and explore the impact of the number of employed methods, classes and data structures, and opcodes associated with data structures on attributing Android apps to an author.

Android background. An Android application, an apk file, is a compressed folder that contains a variety of files including an executable .dex file; AndroidManifest.xml file that describes the content of the package, resource files (e.g., image, sound files), and optional native code that is usually called from the classes.dex file.

A classes.dex file is a binary that results from compiling the app's Java source code. As illustrated by Figure 1, this file is partitioned into different sections that describe the structure of the file.

Among them are several identifier lists that contain offsets pointing to the corresponding entries in the data section. As such string identifiers list (string_ids section) provides offsets to all strings used by .dex file, while class identifier list contains offsets to the information related with classes. This class information list contains offsets pointing to method's information, this method's information contains offsets to the actual code that belongs to it. The code section include binary information (bytecode) about opcode and employed in-line data structures.

In this work, we focus on the use of types, methods, classes, fields, data structures employed and the opcodes related to data structure usage contained in method's code section. To reduce the amount of opcodes to analyze, we employ a filter to discard previously known or analyzed common code [10]. The statistics from this percentages of methods are also included in our feature vector.

Employed features. While programming in Java, an author could choose between several data structures to fulfill her purposes. After compiling the code data structures look different from basic arrays. Let us review as an example the Java code in Figure 2 that includes data structures and arrays. Line 28 that uses a hashtable is transformed in the following smali representation:

```
const/4 v8, 0x1
iget-object v1, p0,
          Lcom/gsg/test1/MainActivity;->
          numbers:Ljava/util/Hashtable;
const-string v2, "one"
invoke-static {v8}, Ljava/lang/Integer;->
          valueOf(I)Ljava/lang/Integer;
move-result-object v3
invoke-virtual {v1, v2, v3},
          Ljava/util/Hashtable;->
          put(Ljava/lang/Object;Ljava/lang/Object;)
          Ljava/lang/Object;
```

```
13  public class MainActivity extends ActionBarActivity {
14      Hashtable<String, Integer> numbers
15              = new Hashtable<String, Integer>();
16      ArrayList<String> MyArrayList
17              = new ArrayList<String>();
18
19      double aPowersOfTwo[] = new double[5];
20
21      int aNums[] = new int[5];
22      @Override
23      protected void onCreate(Bundle savedInstanceState) {
24          super.onCreate(savedInstanceState);
25          setContentView(R.layout.activity_main);
26
27
28          numbers.put("one", 1);
29          numbers.put("two", 2);
30          numbers.put("three", 3);
31
32          MyArrayList.clear();
33          MyArrayList.add("one");
34          MyArrayList.add("two");
35          MyArrayList.add("three");
36          MyArrayList.remove(2);
37
38
39          aNums = new int[]{2, 4, 6};
40          aPowersOfTwo[2] = 4;
41          aPowersOfTwo[3] = 9;
42          aPowersOfTwo[4] = 16;
43          String aStooges[] = {"Larry", "Moe", "Curly"};
44      }
45
```

Figure 2: Basic Android java code that include data structures and arrays.

The creation of array of integers in line 39 is represented in smali as:

```
const/4 v7, 0x3
new-array v1, v7, [I
fill-array-data v1, :array_0
:array_0
.array-data 4
   0x2
   0x4
   0x6
.end array-data
```

For the final example, string array from line 43 is represented in smali as:

```
new-array v0, v7, [Ljava/lang/String;
const/4 v1, 0x0
const-string v2, "Larry"
aput-object v2, v0, v1
const-string v1, "Moe"
aput-object v1, v0, v8
const-string v1, "Curly"
aput-object v1, v0, v6
.local v0, "aStooges":[Ljava/lang/String;
```

The decisions and experience of a developer will impact in the way data structures are used in the Android app. This usage could be related to the functionality of the app. However, it is our believe that developers decisions influence more on the usage of data structures than the functionality of the app. Basic arrays are meant to store data of similar type, these are very common in Android development and are used in almost all Android apps. A quick analysis of app extracted from GooglePlay market confirms this intuition. Figure 3 gives us an overview on how Android programmers

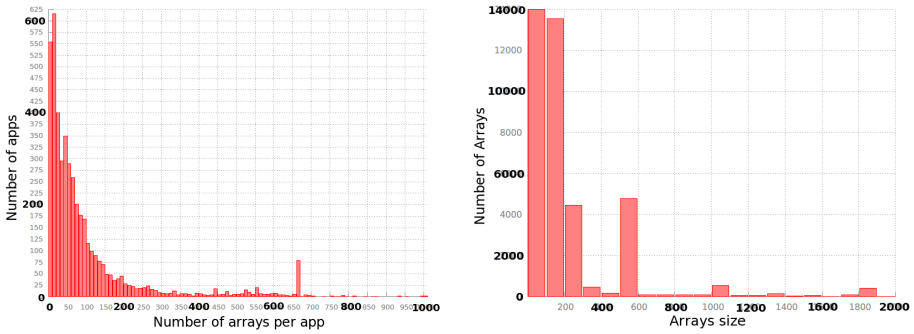

Figure 3: Array usage and array's size distribution from GooglePlay market.

Table 1: Array- unrelated features

total number of classes
total number of classes containing interfaces
total number of classes containing information about source file
total number of classes containing annotations
total number of classes containing data
total number of direct methods
total number of virtual methods
total number of abstract methods
total number of methods containing try and catch
total number of methods containing debug information
total number of static fields
total number of instanced fields
total size of instructions expressed in bytes

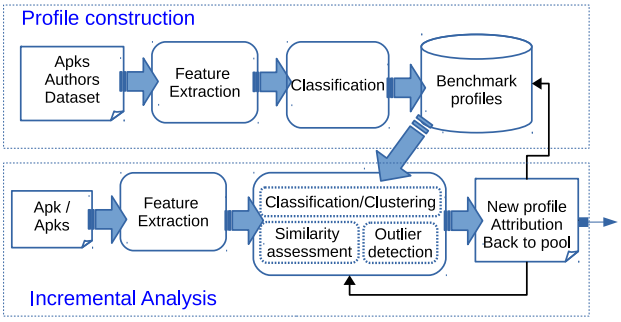

Figure 4: Automatic incremental learning for attribution framework

use array data structures in legitimate apps (see GooglePlay-2015 dataset in Table 3 for details). Out of 4517 apps only 106 (2%) did not employ arrays. Among the rest, the majority of apps use at least 300 array structures.

Figure 3 shows distribution of array's size. The largest size includes more than 14,000 elements while the smallest array has only 5 elements. The average size of an array is 37.54 elements with an standard deviation of 239.29 elements. This large deviation shows that the usage of arrays can be a potential indicator of an author's distinctive development style. Along with the rest of the data structures employed in the app.

Table 2: Array related bytecode mnemonics

Opcode bytecode	Description
ARRAY LENGTH	Store the length of the indicated array in the given destination register .
NEW ARRAY	Construct a new array of the indicated type and size.
FILLED NEW ARRAY	Construct an array of the given type and size, filling it with the supplied contents.
FILLED NEW ARRAY RANGE	Similar as previous.
FILL ARRAY DATA	Fill the given array with the indicated data
AGET, AGET BOOLEAN, AGET BYTE, AGET CHAR, AGET SHORT, AGET WIDE, AGET OBJECT	Perform array operation at the identified index of the given array, loading into the value register.
APUT, APUT BOOLEAN, APUT BYTE, APUT CHAR, APUT SHORT, APUT WIDE, APUT OBJECT	Perform array operation at the identified index of the given array, storing into the value register.

4 AUTHORSHIP ATTRIBUTION FRAMEWORK

The goal of the proposed approach is to systematically attribute Android apps to corresponding author's profiles. Typically, attribution studies in literary domain focus on identifying an author of a sample out of a set of candidates based on sample's similarity to stylistic discriminators found in benchmark profiles. In malware domain, this has a limited value as benchmark profiles only represent a small set of known authors. As such, it is necessary to step beyond the known set and group stylistically similar binaries not attributed to the existing profiles as they might potentially be linked to a new author.

The proposed framework incorporates two components: a *profile construction* focusing on creating benchmark profiles for known authors, and an *incremental analysis*, responsible for a ongoing analysis of Android binaries, their attribution to the existing benchmark profiles and the generation of new profiles for stylistically similar binaries refereed to as emerging profiles. The overview of the framework is given in Figure 4.

The framework takes as an input an Android app. As the first step, the list of data structures used in the incoming app is extracted, then this app is parsed to retrieve bytecode from .dex file which is then analyzed to extract the rest of the features. To reduce the overhead in classification and speed up the process, common code is discarded. To determine and extract common code we follow a procedure explained in [10]. From the remaining bytecode, values related to array size, array operations (see Table 2), and values related to classes, methods and fields (Table 1) are extracted. These raw values are abstracted into a feature vector composed of four parts: (1) a fixed number of features from the numerical values of classes, methods, and fields; (2) the statistic values from size of array definitions, average, mean, median, standard deviation and variance; (3) variable length features extracted from the values of n-grams created from array operations opcodes and (4) the n-grams created from data structures detected. Since the resulting vector of features has variable lenght, we employ a mapping technique presented by Rieck et al [17] to convert it into a fixed length array. In this process each n-gram is hashed, then a number of bytes is chosen to represent the index on a sparse vector. This feature vector serves as a basis for grouping stylistically similar binaries.

Profile construction. The first framework component is responsible for generating the baseline model for further detection and clustering analysis. It can be viewed as a supervised authorship attribution problem; that is given a set of Android authors $A = a_1; a_2; a_3; \ldots; a_i; \ldots; a_n$ and their respective apps $a_i = app_1; app_2; app_3; \ldots; app_m$ to generate a model that can be used to attribute Android binaries to a given set of fixed authors. The aim is, given an apk file app_x, to determine who among these authors wrote it. We use machine learning techniques with prediction probability capabilities to build this model. We use Random Forest prediction as a classifier to build this model.

Random Forest classification algorithm was proposed as a bagging technique with an additional layer of randomness. When building standard trees, each node is split using the best split among all variables. In a Random Forest each node is split using the best split among a subset of variables randomly chosen at this node. Although this strategy might seem counter intuitive, it is quite robust and turns out to perform very well even in the presence of overfitting [3].

Incremental analysis. Once the model outlining benchmark profiles is generated, an incremental component is responsible for attributing new apps to the existing profiles (classification) and discovering new possible profiles (novelty detection and clustering) for Android apps that were not attributed yet to existing authors. Novelty detection of multiple classes is not studied as well as detection for one class at a time [8]. Here we propose an approach that let us create multiple classes from apps not seen before, and cluster them using same results from class prediction. This process is outlined in Algorithm 1. The algorithm takes as input apps to be processed (*apps*), thresholds (*NoveltyThreshold*, *GroupThreshold*) and *buckets* to group similar apps. It offers as output apps attributed (*appsAttributed*) and apps to create new profiles ($appsForNewProfiles$). The assignment of new apps to a *bucket* is based on the probability score of an *app* belonging to an existing profile, or similarity between incoming apps, if a new profile needs to be formed.

Similar to profile construction step, we employ Random Forest classification to calculate a probability score of a given app being a part of each of the benchmark profiles. However, any machine learning method that gives a probability could be employed.

If a probability exceeds a defined *NoveltyThreshold* (we experimented with several values), we attribute this instance to a predicted author profile (line 9), otherwise this probability is used to analyze app similarity with other apps(line 12). In other words, if app is scored to be a part of profile A with 98%, it is automatically assigned to that author's profile. However, if the probability score is 20%, then it is clustered with other apps that are also given 20% probability of being a part of profile A. The intuition behind this is simple: even though these apps are not close enough to profile A, they share common attributes and can potentially form a new candidate profile. How the apps are clustered is guided by bucket size, e.g., with a bucket size of 10, for any given profile apps with a score 0-10%, 10-20%, 20-30%, etc. will be grouped together.

Algorithm 1 Minibatch Cluster and Classify algorithm

```
 1: function Incremental profile discovery
 2:     input : apps
 3:     input : incrementalThreshold
 4:     input : benchmarkThreshold
 5:     input : buckets
 6:     output : appsAttributed, appsNewProfiles
 7:     for each app in apps do
 8:         Calculate ProbabilityScore
 9:         if ProbabilityScore > benchmarkThreshold then
10:             Add app to appsAttributed
11:         else
12:             Grouping app into buckets
13:         end if
14:     end for
15:     for each bucket in buckets do
16:         if number of apps in bucket > incrementalThreshold
    then
17:             Remove outliers from bucket
18:             Create NewProfile from bucket
19:             add NewProfile to appsNewProfile
20:         end if
21:     end for
        return appsAttributed, appsNewProfiles
22: end function
```

The next step is to assess how similar these unassigned apps are within their respective groups and filter out potential outliers (line 17). The similarity between grouped apps is calculated based on Cosine Similarity ($K(X, Y)$) which computes the normalized dot product of X and Y as follows:

$$K(X,Y) = \frac{\langle X,Y\rangle}{(\|X\| * \|Y\|)} \quad (1)$$

To define outliers, first we found maximum distance among instances, calculate average, standard deviation, and quartiles $(q1, q2, q3)$. Then we decide on an outlier threshold between a naive approach inspired by standard deviation detection and the standard box plot rule (Equation 3). Instead of deciding 2 standard deviations as a threshold for upper outlier, we defined the average of maximun distance and average distances(Equation 2).

$$outlier = \frac{maximun\_distance + average}{2} \quad (2)$$

$$outlierboxplot = q3 + iqd * 3 \quad (3)$$

In Equation 3 $q1, q3$ represent lower and upper quartile, and $iqd = q3 - q1$ is interquartile distance.

Finally, an outlier is flagged if more than half of similarity distances are over this outlier threshold (we experiment with different outlier thresholds as such as maximum or minimum of the previous values, and with no outlier detection at all.)

If the resulting clusters of apps is over the $GroupThreshold$ (several values were explored), it forms a new candidate profile(line 18). It is important to note that this probability-based clustering is employed to avoid expensive pair-wise comparison among all unassigned apps. Apps not attributed in this round, will be assessed in next round, this time including new candidate profiles discovered in the last round.

Our model is retrained to incorporate new candidate profiles discovered in previous step. Note that as opposed to benchmark profiles that include validated information about their authors, these new profiles are limited to a label that is created automatically from originating profiles, probability, and number of apps in a newly created group.

5 DATA

To evaluate our proposed approach and study for the automatic creation of author label profiles, we gathered a large collection of 196,385 unique Android applications from different sources that we employed for various stages of analysis (Table 3).

To validate our approach we carefully selected some apps from "markets dataset" that contain the same serial certificate number used to sign these apps. We omitted public, leaked and debug certificates which anyone can have access to. Then we manually verified apps from the remaining 43 authors. To ensure that only unique apps were used in our analysis, all apps with duplicate .dex files were removed, authors with less than 20 apps were also removed. We finally obtained a set with 33 authors and 1428 apps.

Since these authors were deemed to be legitimate (verified through VirusTotal website), we also searched for those who produce malicious apps. For this purpose we employed Kooduos collaborative system [13], collecting groups of apps with the same certificates and flagged as malicious. This process produced a set of 222 apps from 10 different authors. In addition to this, we retrieved 131 malware apps produced by Hacking Team, identified in Kooduos system by a community yara rule. These apps presumably from Hacking Team contain different certificates. If they are written by the same set of people, our system should be able to recognize this.

To explore the benefits of our approach in practice, we also collected apps from a number of other sources. Specifically, several malware families from Drebin dataset [2]; 1,395 apps from open source market Fdroid; 10 adware and ransomware families; 4,574 apps collected from Google Play; 23,656 apps from eight third-party markets; and 116,264 apps retrived from VirusTotal stream.

Details of datasets are shown in Table 3.

6 EXPERIMENTAL RESULTS

On our experiments, we aimed to evaluate several aspects of our approach:

(1) Validate the effectiveness of the proposed approach in attributing apps to the corresponding authors.
(2) Investigate the effect of $NoveltyThreshold$ and $GroupThreshold$ on accuracy of attribution.
(3) Assess the ability of the incremental analysis component to attribute unknown apps and generate emergent profiles.
(4) Evaluate our approach capability to attribute a stream of wild data to the existing profiles.

6.1 Validating effectiveness

To validate the proposed approach, we employed our manually labeled *Authors dataset* and performed 5-fold cross-validation in all experiments. The classification was based on authors' feature vectors composed of classes, fields and methods information, data structures employed and array related mnemonics. Since the amount of features containing application structure related values is stable, a major concern is a potentially significant amount of generated n-grams from data structures and array related opcodes. We evaluated the approach accuracy for values of n ranging from 1 to 8, then embedding the results into a fixed length array of 541, 1053, 2077, 4125, 8221, 16413, 32797, 65565 and 131101 features. The results are presented in Table 4.

The best result in terms of accuracy is 98.12%, with 8-gram and 65565 features, and 158 seconds in average to perform the classification. With a large number of features the time to perform training and testing increases. With a large number of n-grams the time to extract and represent the features increases. We decide to choose not the best accuracy, but the best trade-off between accuracy and time based on the n-grams and number of features. The selected parameters for future experiments are 3-grams with 2077 features, which lead to an accuracy of 97.7% and 68 seconds on average to perform classification.

6.2 Tunning thresholds

The proposed incremental analysis approach relies on three different thresholds for attributing binaries and forming

Table 3: Datasets

Name	Apps	Description
Authors dataset	1,444	Collected from eight different markets, this dataset contains information about 37 known authors with more than 20 apps in the markets, based on the assumption that same serial numbers certificates leads to same author (without considering public or leaked certificates)
Malicious authors	222	Collected from koodous system, this dataset contains at least 10 unique and malicious apps from 10 different authors. We use the same assumption that similar certificates lead to same author.
Hacking Team apps	131	Mobile apps offered by surveillance malware vendor Hacking Team. They were collected from koodous system from community ruleset 675.
Drebin partial dataset	3,181	From the original dataset, we only retained 47 malware families with more then 20 apps.
Adware dataset	211	Three related families (kemoge 90, shedun 97, shuanet 24) known to trojanize legitimate apps.
Ransomware dataset	408	Ransomware including the following seven families, fakeDefender 44, koler 74, Pletor 16, RansomBO 100, scarePakage 2, sLocker 72 and svpeng 100
Fdroid dataset	1,395	Open source apps without advertisement libraries.
GooglePlay-2015 dataset	4,574	Apps collected from Google Play in middle 2015 from all categories, top popular and top new.
Markets dataset	23,656	Apps collected from the following third-party markets: 360, 3gyu, anzhi, aptoide, mobomarket, nduoa, tencent, xiaomi.
GooglePlay-2016 dataset	898	Apps collected in January 2016.
VirusTotal stream	116,264	Suspicious apps provided by VirusTotal service.
Total	196,385	

Table 4: Accuracy results over 5-fold cross validation for different n-gram values

N-grams	# features	Performance (sec)	Accuracy
8	65565	158.61	98.12 %
5	32797	102.67	97.85 %
4	131101	92.49	97.84 %
7	65565	140.46	97.83 %
8	131101	171.01	97.77 %
6	65565	122.47	97.77 %
4	16413	84.83	97.76 %
5	16413	102.14	97.76 %
3	**2077**	**68.69**	**97.71** %
6	131101	128.21	97.71 %
5	131101	111.39	97.70 %
4	65565	86.27	97.70 %
5	4125	102.64	97.69 %
7	131101	149.31	97.69 %
5	8221	102.75	97.63 %

new profiles: *NoveltyThreshold* , *outliersThreshold* and *GroupThreshold*. We conducted a series of experiments to determine the impact of these thresholds on attributing apps using *Authors' dataset*. Our classifier was trained on 75% of the authors (i.e., 25 authors and their corresponding apps) and tested on the remaining 8 authors. Since traditional metrics for evaluation of classification accuracy did not reveal enough information about how the apps were attributed, we used several other metrics to help us assess the behavior of thresholds. Specifically, we evaluated thresholds in terms of percentage of attributed apps AA, percentage of correctly attributed apps among all considered apps TCA, and a percentage of correctly attributed apps among those that were attributed to some profiles (CA).

$$AA = \frac{Apps\_attributed}{Total\_apps\_reviewed} \tag{4}$$

$$TCA = \frac{Apps\_correctly\_attributed}{Total\_apps\_reviewed} \tag{5}$$

$$CA = \frac{TCA}{AA} = \frac{Apps\_correctly\_attributed}{Apps\_attributed} \tag{6}$$

For this analysis, we also experimented with several bucket sizes: 10, 15, 20, 30 and three scenarios for filtering outliers: no outliers filtering (No), maximum (MAX) or minimum (MIN) between values defined to detect outliers. To avoid situations when 2-3 apps form a new profile, the minimum number of apps required to establish a profile was enforced as *GroupThreshold*, we experimented with 20, 15, 10, 7, 5 apps as minimum (20 is the least amount of apps in the dataset). The best results are showed in Table 5.

While we can achieve over 96% in correctly attribute apps (CA), AA and TCA values are below 80%. The highest

Table 5: Thresholds tuning using Authors' dataset

Bucket size	Min num of apps allowed	Outliers	Total apps	AA	TCA	CA
10	7	MIN	361	78.77 %	76.39 %	**96.97** %
10	10	MIN	396	74.82 %	71.61 %	95.71 %
10	5	MIN	387	83.87 %	80.64 %	96.15 %
10	5	NO	325	85.99 %	80.30 %	93.38 %
15	10	MIN	377	78.41 %	75.52 %	96.31 %
15	5	MAX	415	88.09 %	80.47 %	91.35 %
15	5	NO	374	87.10 %	**82.26** %	94.45 %
20	5	MIN	402	87.62 %	81.67 %	93.21 %
30	5	MIN	361	**88.43** %	79.48 %	89.89 %
30	10	NO	420	86.48 %	75.02 %	86.74 %

amount of attributed apps (AA=88.43%) was achieved with a bucket size of 30 and 5 apps as a minimum. Similar results were received with a bucket size of 15 and 5 minimum apps, that is 87% with no outlier detection, and 88.09% with maximum outlier detection. Since the difference between these results is insignificant, other factors have to be taken into consideration. One of these is the presence of outliers detection. Generally, additional processing of candidate profiles incurs an overhead on the system, so with equal performance, it is desirable to avoid this overhead. It is interesting to note though that these threshold settings (bucket size of 15, 5 minimum apps, and no outlier detection) also provide the highest amount of correctly attributed apps among these options (94.45%). These are the parameters we will employ throughout the rest of the experiments.

We repeat the same experiment, with Malicious authors data. The best 10 results based on TCA value are shown in Table 6. The performance under different settings is similar. The highest amount of apps is attributed with bucket size of 20, 5 minimum apps and no outlier detection (AA=80.47% with CA=70.70%). However, the threshold parameters selected in the previous experiments reached very similar result (AA=78% and CA=71%).

6.3 Incremental analysis

Given the optimal thresholds obtained in the previous round of experiments, we turned our attention to incremental analysis component.

Starting with benchmark profiles, we sequentially feed our system with the following datasets in this order: Hacking team, Drebin, Adware, Ransomware, Fdroid, GooglePlay-2015 and Markets datasets. This experiment was conducted in two phases: malicious data first, followed by the last 3 datasets. The results from the first phase are given in Table 7. The overall attribution rate for each of the datasets is fairly high, reaching 100% in case of apps coming from the Hacking Team. Out of attributed apps, the majority were assigned to labels (emergent profiles) generated for a given set (90% of attributed Hacking Team apps were part of 3 new profiles). For the malicious data, it is safe to assume that apps attributed to benchmark profiles are labeled incorrectly. Intuitively, this makes sense as original benchmark profiles were built on benign data and thus attribution of malware to legitimate profile is likely to be an error. The overall percentage of these miss attributed apps across datasets in this phase ranged between 9-0%.

The second phase focused on the datasets thought to be primarily benign. As a result of this experiment, Fdroid dataset produced 124 emergent profiles, Googleplay2015 dataset produced 733 profiles and Markets data produced 94 new profiles. Even though we did not expect any apps to be attributed to profiles generated in the first phase, less than 10 apps overall were labeled as part of these malicious emergent profiles. Manual check of these apps revealed some suspicious behaviour and presence of adware. The percentage of apps attributed to emergent profiles also dropped compared to what we saw in case of malicious data. This is likely to be the result of authors' diversity, i.e., the number of different authors contributing to GooglePlay market is significantly larger than that in Hacking team, as an example.

6.4 Attributing a stream of apps

In practical setting, an analyst might be interested to identify apps in the wild that are potentially written by a given set of authors. Using multiple certificates and claiming different identities is not uncommon on the markets. However, figuring out whether or not apps belong to the same author is challenging. Having an easy way to attribute unknown apps to already established and well-known profiles is beneficial in practice. As such in this experiment we focus on a given scenario.

We evaluated a stream of over 160 000 wild apps obtained from GooglePlay and VirusTotal using defined thresholds, benchmark and emergent profiles generated by all other datasets. In this experiment we were not concerned with generation of new emergent profiles, but were rather interested to see if any apps captured in the wild can be attributed to already established profiles. The results are given in Table 8.

With the size of our datasets and the variety of potential authors, it is not surprising that much smaller percentage of apps were attributed. 6% of apps in GooglePlay data were attributed to overall 59 existing profiles and 9% of VirusTotal set to 104 profiles. Among them, 20% of GooglePlay apps and 25% of VirusTotal apps were attributed to original known

Table 6: Thresholds tuning for emerging profiles, attributing Malicious authors data

Bucket size	Min num of apps allowed	Outliers	*AA*	*TCA*	*CA*
10	5	MIN	75.34 %	**58.36** %	**77.46** %
10	5	MAX	76.96 %	56.76 %	73.75 %
10	5	NO	77.05 %	57.50 %	74.63 %
10	7	MIN	72.32 %	54.91 %	75.93 %
15	5	MIN	78.20 %	56.42 %	72.15 %
15	5	MAX	78.18 %	57.64 %	73.73 %
15	5	NO	78.29 %	55.81 %	71.29 %
15	7	MIN	75.56 %	54.86 %	72.61 %
20	5	MIN	78.99 %	56.73 %	71.83 %
20	5	MAX	80.29 %	54.53 %	67.91 %
20	5	NO	80.47 %	56.89 %	70.70 %
20	7	MIN	77.77 %	54.37 %	69.91 %
30	5	MIN	80.18 %	55.54 %	69.27 %
30	5	MAX	80.36 %	54.44 %	67.74 %
30	5	NO	**80.61** %	53.94 %	66.92 %

Table 7: Evaluation of incremental analysis

Dataset	Num of apps	Num of emerg. profiles	*AA*	Apps attrib. to emerg. profiles
Hacking team	131	3	100%	90% (118)
Drebin	3181	117	50% (1588)	91%(1442)
Adware	211	3	54% (113)	94%(106)
Ransomware	408	12	86% (349)	97%(338)
Fdroid	1395	124	18% (254)	100%(254)
Googleplay2015	4574	733	31% (1442)	20%(289)
Markets datasets	23 656	94	10% (2478)	68%(1681)

Table 8: Attributing stream of wild apps

Dataset	Num of apps	Num of attrib apps	Num of profiles	Num of apps attrib to bench profiles
Googleplay2016	44,915	6%(2921)	59	20%(598)
VirusTotal	116,264	9%(10 717)	104	25%(2721)

and verified benchmark profiles. The remaining attributed apps were labeled as parts of emergent profiles. As such 9 GooglePlay apps were attributed to a profile dominated by Hamob malware family. This profile was manually checked and all apps were verified through VirusTotal service. Our approach was not designed for malware detection, however this finding indicates that the same author was involved in development of samples of this malware family, or at least in part. This is common in practice and only confirms our results.

Similarly, among apps from VirusTotal repository 544 apps were attributed to 31 emergent profiles that were deemed suspicious. Checking these apps through VirusTotal service also confirmed their correct labeling. Among 544 apps, 96.51% (525) where correctly attributed to various malicious families, and only 16 apps were detected as benign. Their further manual analysis showed that all 16 apps had the same structure and used the same set of advertisement libraries in exactly same way, as a result we assume that they probably belong to the same source/author and if we were generating emergent profiles, they would likely form one profile.

6.5 New labels for a known dataset

Using defined thresholds, benchmark and emergent profiles generated by all previous datasets, we attribute the whole Drebin dataset, not just the part that we use before to assign new labels based on possible author profiles.

From 5,555 apps 3,643 remain without known author. The rest of apps where attributed to 28 probable authors, with a close match between families and low mixture between families and profiles. The whole list of apps and labels could be downloaded from **github.com/hugo-glez/author-labels**.

7 CONCLUSION

In this work we presented an approach to attribute unlabeled apps to an established set of author profiles and generate Android developers' profiles for known apps. Our analysis is based on author's development style extracted from various features related to authors decisions about development process, number of employed methods, classes and data structures, and associated opcodes extracted from Android binary code. We design and present an incremental Mini batch classify and cluster algorithm, that employs Random Forest predictor to determine the probability of an app belonging to an existing profile. We analyzed almost 35,000 apps and created 259 profiles. From these profiles, 93 where related to malicious apps and authors.

We also evaluated over 160,000 apps to further assess the effectiveness of our approach in attributing apps. Since ground truth is not available for some employed data, we can not objectively evaluate accuracy, and thus measure the number of attributed apps and verify whether these apps are attributed to suspicious or legitimate profiles.

To the best of our knowledge there are no similar approaches to compare our work with. Previous works only work on close-source world where they identify author previously known. However, our evaluation over a set of known authors showed similar or better performance.

Our model is completely data driven and is able to create new author profiles from real Android apps and attribute unknown binaries to these newly established profiles. Our results are encouraging and offer an incentive to extend and continue the development in this area. To facilitate following research in this area, we release our created datasets to a broader research community.

REFERENCES

[1] Saed Alrabaee, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. 2014. OBA2: an Onion approach to binary code authorship attribution. *Digital Investigation* 11 (2014), S94–S103.

[2] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the 21th Annual NDSS*.

[3] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[4] Steven Burrows, Alexandra L Uitdenbogerd, and Andrew Turpin. 2014. Comparing techniques for authorship attribution of source code. *Software: Practice and Experience* 44, August 2012 (2014), 1–32. https://doi.org/10.1002/spe

[5] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing Programmers via Code Stylometry. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 255–270.

[6] Dong-Kyu Chae, Sang-Wook Kim, Jiwoon Ha, Sang-Chul Lee, and Gyun Woo. 2013. Software plagiarism detection via the static API call frequency birthmark. *Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13* (2013), 1639. https://doi.org/10.1145/2480362.2480668

[7] Radhouane Chouchane, Natalia Stakhanova, Andrew Walenstein, and Arun Lakhotia. 2013. Detecting machine-morphed malware variants via engine attribution. *Journal of Computer Virology and Hacking Techniques* (2013). https://doi.org/10.1007/s11416-013-0183-6

[8] E. R. de Faria, I. R. Gonalves, J. Gama, and A. C. P. d. L. F. Carvalho. 2015. Evaluation of Multiclass Novelty Detection Algorithms for Data Streams. *IEEE Transactions on Knowledge and Data Engineering* 27, 11 (Nov 2015), 2961–2973. https://doi.org/10.1109/TKDE.2015.2441713

[9] Hugo Gonzalez, Andi A. Kadir, Natalia Stakhanova, Abdullah J. Alzahrani, and Ali A. Ghorbani. 2015. Exploring Reverse Engineering Symptoms in Android Apps. In *Proceedings of the Eighth European Workshop on System Security (EuroSec '15)*. ACM, New York, NY, USA, Article 7, 7 pages.

[10] H. Gonzalez, N. Stakhanova, and A. A. Ghorbani. 2016. Measuring code reuse in Android apps. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*. 187–195. https://doi.org/10.1109/PST.2016.7906925

[11] Jane Huffman Hayes and Jeff Offutt. 2010. Recognizing authors: an examination of the consistent programmer hypothesis. *Softw. Test. Verif. Reliab.* 20, 4 (Dec. 2010), 329–356. https://doi.org/10.1002/stvr.412

[12] Yoon-Chan Jhi, Xinran Wang, Xiaoqi Jia, Sencun Zhu, Peng Liu, and Dinghao Wu. 2011. Value-based program characterization and its application to software plagiarism detection. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 756–765.

[13] Koodous. [n. d.]. Collaborative platform for APK analysis. https://koodous.com. ([n. d.]).

[14] Ivan Krsul and Eugene H. Spafford. 1996. Authorship Analysis: Identifying The Author of a Program. *Computers and Security* (1996).

[15] Kaspersky labs. 2017. MOBILE MALWARE EVOLUTION 2016. https://press.kaspersky.com/files/2017/02/Mobile_report_with-Interpol_2016_27-Feb-20172.pdf. (February 2017). Accessed May 27, 2017.

[16] Christian Lueg. 2017. 8,400 new Android malware samples every day. https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day. (2017). Accessed May 27, 2017.

[17] Konrad Rieck, Christian Wressnegger, and Alexander Bikadorov. 2012. Sally: A tool for embedding strings in vector spaces. *Journal of Machine Learning Research* 13, Nov (2012), 3247–3251.

[18] N Rosenblum, X Zhu, and B Miller. 2011. Who wrote this code? Identifying the authors of program binaries. *Computer SecurityESORICS 2011* (2011). http://www.springerlink.com/index/C422JT41687VU470.pdf

[19] Nathan E. Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2010. Extracting compiler provenance from program binaries. *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '10* (2010), 21. https://doi.org/10.1145/1806672.1806678

[20] Eugene H Spafford. 1994. Computer viruses as artificial life. *Artificial life* 1, 3 (1994), 249–265.

[21] Eugene H. Spafford and Stephen A. Weeber. 1992. Software Forensics: Can We Track Code to its Authors? (1992).

[22] Efstathios Stamatatos. 2009. A Survey of Modern Authorship Attribution Methods. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE AND TECHNOLOGY* (2009), 538556.

[23] Benno Stein, Nedim Lipka, and Peter Prettenhofer. 2011. Intrinsic plagiarism analysis. *Lang. Resour. Eval.* 45, 1 (March 2011), 63–82.

[24] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. 2009. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 545–554.