DroidPDF: The Obfuscation Resilient Packer Detection Framework for Android Apps

CAIJUN SUN1,2, HUA ZHANG1, (Member, IEEE), SUJUAN QIN1, JIAWEI QIN1, YIJIE SHI1, AND QIAOYAN WEN1
1State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China
2State Key Laboratory of Cryptology, Beijing 100876, China

Corresponding authors: Hua Zhang (zhanghua_288@bupt.edu.cn) and Sujuan Qin (qsujuan@bupt.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61671082, Grant 61672110, Grant 61976024, and Grant 61972048, and in part by the Fundamental Research Funds for the Central Universities under Grant 2019XD-A01.

ABSTRACT The Android packing techniques were originally used to pack and conceal important information of the apps to prevent malicious developers from deconstructing the software logic. However, due to the lack of supervision, packing techniques have become the common methods for Android malwares to harden apps and circumvent virus detection engines in recent years. With obfuscation and encryption techniques, packing engines can alter the code structure of malwares and hide the malicious code to deceive and bypass the detection mechanisms, such as signature matching. For packing techniques, the packers are the agents created by packing engines and used to protect the softwares. Hence the packers pose a major challenge to the automated malware detection when researchers analyze a large collection of Android apps statically. It is necessary to identify packed samples in advance so that researchers can adopt different process procedures. To address this problem, we propose an intelligent Android Packer Detection Framework called DroidPDF. It adopts a concise feature set that is resilient to obfuscation techniques. It also introduces weighted entropy to improve the detection effectiveness and achieves an average F1 Score of 0.9870.

INDEX TERMS Android packer, Android packer detection, Android malware analysis.

I. INTRODUCTION

As the most popular mobile operating system, the current Android market share is as large as 76.2% [1]. In 2018, 75.7 billion downloads were made through Google Play [2]. The great success of Android apps provides attackers with lucrative and profitable targets. In particular, attackers disassemble popular apps with reverse-engineering technique, pack them after inserting modules with malicious behaviors, then upload repackaged apps to various markets to compromise the victims’ smartphones.

According to 2018 Mobile Threat Report of McAfee [3], they noticed a 77% increase in Android banking Trojans. The banking Trojans evolve constantly and have been the main source of revenue for cyber criminals. According to another report of AVL Team [4], they observed a rapid growth of packed Android malwares. To be more specific, the number of packed subjects increase from 54971 to 1897256 from 2013 to 2017. With the growth of popularity, Android also attracts the attention of malware developers who use advanced code packing techniques.

Packing engines perform a series of purposeful operations that modify the original code structure, encrypt the payloads of apps and insert stub programs which serve as unpacking routines to release the payloads at runtime [5]. Most packing engines also adopt a number of obfuscation techniques to make the logic obscure and unreadable [6]–[10], including anti-debug, string encryption, payload encryption, dynamic code loading, control flow transformation, junk code insertion, etc. On the one hand, benign apps employ packers to protect proprietary logic and sensitive algorithms from misusing. On the other hand, malicious apps make use of packers to conceal malicious functionalities to escape from anti-virus engines. Anti-virus engines can easily recognize known malwares by matching the predefined signatures or patterns of malicious apps. However, the signatures or patterns become invisible once the apps are packed. There is a growing trend that malwares are taking advantage of packers to evade anti-virus engine. It has been a common phenomenon that malwares adopting packing...
techniques pretend to be popular apps. The emergence of packed malwares pose a serious challenge to the anti-virus engines during static analysis of a large collection of samples [11]–[13].

In light of this background, we propose a lightweight solution named DroidPDF to detect packed samples. It is able to identify whether an Android Package (APK) has been packed or not by pattern recognition [14]. It can preprocess the samples to automate the identification of malwares. For example, identify packed samples in advance so that researchers can adopt additional operations for packed samples to recover the source code and the anti-virus engine can always have source code to analyze.

In summary, our major contributions include:

- We introduced **weighted entropy** to improve the detection performance. **Weighted entropy** is a technique based on information entropy and customized for Android packer detection. It allocates more weight to subjects with higher entropy to increase the differentiation between the encrypted text and the plain text.
- We trained a precise classification model with highly trusted data and derive a concise and effective feature set of 9 independent features from the raw attributes. Through comparison, DroidPDF outperforms other state-of-the-art frameworks.
- Our feature set is robust and resilient to obfuscation techniques. Although packers adopt a series of obfuscation techniques, DroidPDF is still able to achieve an average F1 Score of 0.9870. When it comes to identify obfuscated samples, DroidPDF does not experience dramatic performance decrease and achieves an average F1 Score of 0.9819. Through statistical test, we find that there is no significant difference between the scores of DroidPDF achieved on normal and obfuscated datasets. In other words, DroidPDF is resilient to obfuscation techniques.

The rest of the paper is organized as follows. Section II introduces related background knowledge regarding the packing mechanism and information entropy. Section III demonstrates the implementation of our framework DroidPDF which is a practical approach to detect the packed Android apps from a massive collection of samples. Section IV presents the method of selecting optimal classification model for DroidPDF and evaluates DroidPDF from various angles. The results show that DroidPDF has a favorable prospect for Android packer detection. Section V discusses our work and brings up the limitations and future work. Section VI cites some related researches and states briefly the advantages and disadvantages of them. Section VII concludes the paper.

### II. BACKGROUND

Before presenting our packer detection framework, we first introduce some prerequisites concerning the packing mechanism and entropy theory.

### A. PACKING MECHANISM

Android packer usually refers to the runtime packer [15]. Runtime packer is a process that decompresses or decrypts the packed code and data during execution. A runtime packer routine, usually uses native libraries to decompress or decrypt the original code dynamically, loads the released code into memory, then hands over the control from packer routine to the released application. In the majority of cases, the packer only contains a DEX file and several native libraries after decompressing. In fact, the DEX file generally serves as a stub program of packer to initialize and launch the packer routine. As for the native libraries, they contain proprietary algorithms used to recover the original payload of the app. The general design is that the packer routine starts up from the stub DEX file which loads the native libraries. These native libraries are the major unpacking agents which perform all the necessary steps to recover the DEX file structure to form a valid DEX file that is homologous with the original one. Then the packer routine loads the recovered DEX file into memory and commences executing it, meanwhile, hands over the control to the newly started application.

As aforementioned, most packing engines also adopt several obfuscation techniques to make the logic difficult to understand. Through our manual analysis and other researches [6]–[10], we find that packers usually adopt following obfuscation techniques to protect apps.

- **Anti-debug.** Packers usually create a sub-process to check if current app is being debugged. Packers will exit the main process once being attached by other processes.
- **String Encryption.** Packers usually encrypt constant strings so that some sensitive values will not leak out easily.
- **Payload Encryption.** Packers usually encrypt the original payload and decrypt it only at the time of execution.
- **Dynamic Class Loading.** Packers usually load classes with dedicated class loaders at runtime.
- **Control Flow transformation.** Packers usually convert Java methods to native methods which are invoked through JNI calls. Therefore the real control flows are concealed.
- **Junk Code Insertion.** Packers usually insert junk code to change the work flow and make it hard to understand.

### B. ENTROPY

Entropy [16], also called information entropy, is measured as the average amount of information contained in received message in information theory. The more random the data is, the greater the entropy will be. Therefore, high entropy is usually deemed as a symbol of uncertainty instead of certainty.

Lyda and Hamrock [17] introduced entropy analysis and put forward *Bintropy* to measure the statistical variation of executable programs. The underlying principle is that packed and encrypted samples usually generate less predictable sequences. This approach enables analysts to effectively identify packed apps.
Robert’s work provides a good inspiration for Android packer detection through entropy analysis. We study 6 major packers (Ali, Baidu, Bangcle, Ijiami, Qihoo, and Tencent [18]–[23]) and find that after packing process the average entropy increases. At first, we randomly collect 80 open source Android apps, next get them packed with above 6 packers. Then we work out the average entropy of them and compare the packed with non-packed apps pairwise. The average entropy changes of Ali, Baidu, Bangcle, Ijiami, Qihoo, and Tencent packers are +2.45%, +2.4%, +2.8%, +3.1%, +2.6%, +4%, respectively.

In this paper, we introduce weighted entropy which is based on information entropy analysis to measure the uncertainty of a series of bytes of certain decompressed files. We use Eq. (1) to calculate the information entropy of discrete random bytes.

\[
E = - \sum_{i=1}^{M} p_i \cdot \log_2(p_i)
\]  

where \( p_i \in [0, 1] \) is the probability of the \( i \)th character appearing in the stream of characters of the data. \( p_i \) becomes 0 when \( i \)th character does not exist in the data stream and 1 when the data stream only contains the \( i \)th character. The special case is \( p_i = 0 \) because \( \log_2(p_i) \) is an invalid expression. So we make \( p_i \cdot \log_2(p_i) = 0 \) when \( p_i = 0 \). \( M \) is the kind of unique characters. Since there are 256 different characters, the entropy value should be a real number within the range \( 0 \leq E \leq 8 \). The entropy becomes 0 when there is only 1 kind of unique character in the data stream. The entropy becomes 8 when there are 256 kinds of different characters and their frequencies are the same. The corresponding entropy calculator is implemented in Algorithm 1.

Algorithm 1 Calculation of Entropy for Specified File

**Input:** file, the file path of which the entropy is going to be calculated.

**Output:** Entropy of the given file.

1: \( M \leftarrow 256; \)
2: \( BUF\_SIZE \leftarrow 4096; \)
3: \( occurrence \leftarrow \text{Array}[\text{Create an array of size } M \text{ that initialized with } 0]; \) \( \triangleright \) Used to keep the occurrence of each character
4: \( buffer \leftarrow \text{Array}[\text{Create an array of size } BUF\_SIZE]; \) \( \triangleright \) Used to keep the content read from file
5: \( file\_size \leftarrow 0; \)
6: \( fd \leftarrow \text{open(file)}; \)
7: \( \text{while not end of } fd \text{ do } \triangleright \text{Count the occurrence of each character} \)
8: \( \text{size} \leftarrow \text{read(fd, buffer, BUF\_SIZE)}; \)
9: \( \text{for } i \leftarrow 0 \text{ to size by } 1 \text{ do} \)
10: \( \text{char} \leftarrow \text{buffer}[i]; \)
11: \( \text{occurrence[char} \leftarrow \text{occurrence[char] + 1}; \)
12: \( \text{end for} \)
13: \( \text{file}\_size \leftarrow \text{file}\_size + \text{size}; \)
14: \( \text{end while} \)
15: \( \text{entropy} \leftarrow 0; \)
16: \( \text{for } i \leftarrow 0 \text{ to file}\_size \text{ by } 1 \text{ do } \triangleright \text{Calculate the entropy} \)
17: \( \text{if occurrence}[i] \neq 0 \text{ then} \)
18: \( \text{probability} \leftarrow \text{occurrence}[i] / \text{file}\_size; \)
19: \( \text{entropy} \leftarrow \text{entropy} - \text{probability} \cdot \log_2(\text{probability}); \)
20: \( \text{end if} \)
21: \( \text{end for} \)
22: \( \text{return} \text{entropy} \)

are fundamental elements and used to form the final feature set.

1) **SIZE OF APK FILE**
This attribute can be acquired by counting the bytes of the APK directly.

2) **SIZE OF DEX FILE**
To obtain this attribute, we decompress the APK first. Then, count the bytes of the file named classes.dex which is located in the root directory of the decompressed APK.

3) **PACKAGE NAME**
This attribute is stored in the file named AndroidManifest.xml which is located in the root directory of the decompressed APK. It is noteworthy that AndroidManifest.xml is encoded as AXML format. To read the package name after decompressing, it requires an AXML format parser to decode the unreadable text in AndroidManifest.xml.

III. METHODOLOGY
In this section, we describe the procedures of building up DroidPDF. A set of concise and valuable features will be used which are derived by analyzing a series of entries in APK file. By combining with the optimal classification model, DroidPDF is able to recognize both known and unknown packers.

A. ARCHITECTURE
Fig. 1 presents the architecture of DroidPDF. Since the APK file is organized as the zip file format, DroidPDF uses decompression tool unzip [24] to extract compressed files at first. Next, it collects a set of original attributes from the decompressed content. After a series of process on these raw attributes, the original data are transformed to features. Then each sample is simplified into a feature vector which will be sent to the trained classifier and be examined whether packed or not.

B. RAW ATTRIBUTES EXTRACTION
Before calculating the feature vector, we need to extract attributes from APK file. The following 9 types of attributes

| File | Description |
|------|-------------|
| Manifest.xml | Contains the namespace, package, and class information in the AndroidManifest. |
| AndroidManifest.xml | Holds metadata about the app, including permissions, activities, and services. |
| classes.dex | Contains the compiled bytecode of the app. |
| Asset files | Includes images, sounds, and other resources. |
| Resources_apk | Contains the compiled resource files. |
| Primary Dalvik VM | Is the main Dalvik VM that runs the app. |
| Secondary Dalvik VM | Is the Dalvik VM that runs in the background. |
| System libraries | Includes the standard library and native code. |
| Dalvik debug tool | Used for debugging purposes. |
4) APPLICATION NAME
The method of extracting application name is as same as getting Package Name. However, this attribute value may be empty in most circumstances. The reason is that a number of Android apps adopt the default android.app.Application class which does not need to be declared in AndroidManifest.xml. By contrast, if the app implements some customized features by creating a subclass of android.app.Application, it should specify the fully-qualified name of this subclass in AndroidManifest.xml.

5) DECLARED COMPONENTS
In general, there are four typical components which are the primary elements for constructing Android apps, including android.app.Activity, android.app.Service, android.content.BroadcastReceiver, and android.content.ContentProvider. Once the application implements several of them and calls them during execution, these implemented components must be declared in AndroidManifest.xml during development, otherwise it will crash.

6) CLASS DEFINITIONS
This attribute can be extracted from classes.dex. File classes.dex essentially is the Dalvik executable file which is compiled from the Android program. In terms of the DEX format specification, we build up a custom parser to read all the class definitions.

7) CLASS METHODS
This attribute is also extracted from classes.dex. The manner of reading the class methods is as same as getting Class Definitions.

8) LIST OF DECOMPRESSED FILES
To obtain this attribute, it requires recursively searching the root directory of the decompressed APK. Find all the files and count their byte size, respectively.

9) ENTROPY OF DECOMPRESSED FILES
The entropy of all the native libraries (.so extension), JAR files (.jar extension), and DEX files (.dex extension) from last step will be calculated. Particularly, entropy calculation is a compute-intensive task. To achieve high performance, the logic should be written in C language or other compiled language with excellent execution efficiency [25].

C. FEATURE TRANSFORMATION
Since all the attributes are extracted in last phase, we transform these attributes to features for classification models. The feature set contains 9 different kinds of features which are briefly described in Table 1.

1) PROPORTION OF DEX FILE
It is a numeric type which represents the proportion of the DEX file size to the entire APK size. In most situations, the new classes.dex file created by packing engines is a packer stub used to set up packer routine. Most content of the original classes.dex file is encrypted and extracted to other files to avoid being found. In general, the stub file is in smaller size than the original one.

2) NON-EMPTY APPLICATION NAME
It is a boolean type and the value should be set to true if the developer designs a customized Application class. The packer needs to implement its own subclass of the android.app.Application class so that it can maintain the global application state. Moreover, the implemented Application subclass must be declared in AndroidManifest.xml and instantiated before any other classes. After releasing the original content, the packer application switches execution to the original application.

3) PACKAGE NAME IN APPLICATION NAME
It is a boolean type and the value should be set to true if package name exists in the fully-qualified application name.
TABLE 1. Feature set of DroidPDF.

| ID  | Feature                                      | Abbreviation          | Type       | Memory Overload | Computational Complexity |
|-----|----------------------------------------------|-----------------------|------------|-----------------|-------------------------|
| 1   | Proportion of DEX File                       | dex_file_proportion   | numeric    | O(n)            | O(n)                    |
| 2   | Non-empty Application Name                   | nonempty_app_name     | boolean    | O(n)            | O(n)                    |
| 3   | Package Name in Application Name             | package_name_in_app_name | boolean    | O(n)            | O(n)                    |
| 4   | Number of Native Libraries                   | num_native_lib        | numeric    | O(n)            | O(n)                    |
| 5   | Number of Visible Classes                    | num_visible_class     | numeric    | O(n)            | O(n)                    |
| 6   | Number of Native Methods                     | num_native_method     | numeric    | O(n)            | O(n)                    |
| 7   | Proportion of Native Methods                 | native_method_proportion | numeric    | O(n)            | O(n)                    |
| 8   | Proportion of Visible Components             | visible_component_proportion | numeric    | O(n)            | O(n)                    |
| 9   | Weighted Entropy                             | weighted_entropy      | numeric    | O(n)            | O(n)                    |

For the original apps that implement their own Application classes, the fully-qualified application names usually are prefixed with their package names. By contrast, the packers' counterparts do not carry the package names of the original apps' in most cases.

4) NUMBER OF NATIVE LIBRARIES
It is a numeric type which means the number of .so extension files. Native libraries are in the form of .so extension files in Android which are executable and linkable. Most packers contain native libraries which incorporate algorithms of releasing Dalvik byte code. There are two advantages to implement packer routines based on native code. On the one hand, the native code is stored as binary code which is resilient to static analysis and makes the software not easy to be deconstructed. On the other hand, the native code is able to access and manipulate the memory storing the byte code directly.

5) NUMBER OF VISIBLE CLASSES
It is a numeric type which denotes the number of classes defined in classes.dex. Android packing engines will encrypt most of the DEX classes and keep them into other files. Usually, there will be very few classes left. In fact, a normal app without being processed by packing engines has at least 1500 classes.

6) NUMBER OF NATIVE METHODS
It is a numeric type which represents the number of native methods defined in classes.dex. More and more sophisticated packing engines use native method technique to translate Java methods to native ones. The native methods should be invoked by JNI calls and executed by dedicated runtime. Through investigation, we find that almost all the popular packer vendors support native method technique, including Ali, Baidu, Bangcle, Ijiami, Qihoo, Tencent, etc.

7) PROPORTION OF NATIVE METHODS
It is a numeric type which means the proportion of native methods to total methods. Some packers adopting native method technique contain more native methods of which the proportion of native methods is higher.

8) PROPORTION OF VISIBLE COMPONENTS
It is a numeric type which means the proportion of components implemented in DEX file to those declared in manifest file. We all know that Android components should be declared in AndroidManifest.xml before using them. Since packing engines always conceal classes of classes.dex, it is a high probability event that only a few components left in DEX file. The app may be packed if components are registered in AndroidManifest.xml, but not present in classes.dex.

9) WEIGHTED ENTROPY
It is a numeric type of which the value is in the range from 0 to 8. Although the higher entropy follows the higher suspicion of packer, simply calculating entropy on files does not produce sufficient information. It is reasonable to emphasize the weight of high entropy subject since content of high entropy is more likely to carry encrypted sequences. Here we propose the weighted entropy method. Weighted entropy takes in the entropy of these subjects and yields a uniform representation of the entropy.

Before the weighted entropy, we explain how to derive the weight factors acting on each subject. The factors should be expressed as Eq. (2).

\[
w_j = \begin{cases} 
0 & N = 0 \\
1 & N = 1 \\
\frac{1}{N} & N > 1, i < N \\
\frac{2}{N-1} & N > 1, j = N 
\end{cases} 
\]  (2)

where \(N\) represents the total number of data flows, and \(j\) represents the \(j\)th data flow. The data flows are sorted in descending order by the entropy score, which means the first data flow carries the highest entropy and in turn the \(N\)th carries the lowest. The overall summation of weight factors should be 1.
Since we get acquainted with the weight factors, we can derive the weighted entropy as Eq. (3),

$$WE = -\sum_{j=1}^{N} (w_j \cdot \sum_{i=1}^{M} p_i \cdot \log_2(p_i))$$  \(\text{(3)}\)

where \(i\), \(p_i\) and \(M\) are already described in Eq. (1); \(N\) and \(j\) are already described in Eq. (2). The formula of weighted entropy can be summarized as Algorithm 2.

**Algorithm 2 Calculation of Weighted Entropy for Specified Sample**

**Input:** \path, the file path of the sample of which the weighted entropy is going to be calculated.

**Output:** Weighted entropy of the given sample.

1: \texttt{decompressed\_files} \leftarrow \texttt{Decompress path}; \triangleright Use unzip to decompress the file.
2: \texttt{file\_entropy\_map} \leftarrow \texttt{Map}\{\texttt{Create a map}\}
3: \texttt{for file} \in \texttt{decompressed\_files} \texttt{do}
4: \quad \texttt{if file\_extension(file) \notin \{dex, so, jar\} then}
5: \quad \quad \texttt{continue;}
6: \quad \texttt{end if}
7: \quad \texttt{file\_num} \leftarrow \texttt{file\_num} + 1;
8: \quad \texttt{file\_entropy\_map[file]} \leftarrow \texttt{Invoke Algorithm 1 to get the entropy of file;}
9: \texttt{end for}
10: \texttt{Sort file\_entropy\_map in descending order by entropy value}
11:
12: \texttt{file\_num} \leftarrow \texttt{size\_of(file\_entropy\_map)};
13: \texttt{weights} \leftarrow \texttt{Array}\{\texttt{Create an array of size file\_num}\};
14: \texttt{for} \texttt{i} \leftarrow 1 \texttt{to file\_num} \texttt{by 1 do} \triangleright \texttt{Calculate the weight factors, see Eq. (2)}
15: \quad \texttt{weights[i - 1]} \leftarrow 1.0 / \texttt{power(2, i)};
16: \quad \texttt{if} \texttt{i} = \texttt{file\_num} \texttt{then}
17: \quad \quad \texttt{weights[i - 1]} \leftarrow 1 / \texttt{power(2, i - 1)};
18: \quad \texttt{end if}
19: \texttt{end for}
20:
21: \texttt{id} \leftarrow 0
22: \texttt{weighted\_entropy} \leftarrow 0
23: \texttt{for key, value} \in \texttt{file\_entropy\_map do} \triangleright \texttt{Calculate the weighted entropy, see Eq. (3)}
24: \quad \texttt{weighted\_entropy} \leftarrow \texttt{weights[id]} \cdot \texttt{value + weighted\_entropy};
25: \quad \texttt{id} \leftarrow \texttt{id + 1};
26: \texttt{end for}
27:
28: \textbf{return} \texttt{weighted\_entropy};

**D. MODELING**

In the modeling procedure, each app is converted to vector based representation to form the dataset. With the dataset, we can perform hyper-parameter tuning to work out the optimal combination of hyper-parameters for the candidate models. Besides, model selection is also as important as hyper-parameter tuning. It is recommended that researchers compare the performance of different models on the given dataset to determine the finally model. We detail this process in the following paragraphs.

1) FEATURES TO DATASET

Suppose each sample is defined by \(m\) features and the number of our samples is \(n\), then the whole sample set can be represented by a \(m \times n\) matrix \(M\).

\[
M = \begin{bmatrix}
  x_{11} & \ldots & x_{1m} & y_1 \\
  x_{21} & \ldots & x_{2m} & y_2 \\
  x_{31} & \ldots & x_{3m} & y_3 \\
  \vdots & \vdots & \vdots & \vdots \\
  x_{n1} & \ldots & x_{nm} & y_n
\end{bmatrix}
\]

where \(x_{ij}\) means the \(i\)th feature of \(r\)th sample. Each feature and its corresponding ID is listed in Table 1. In our implementation of DroidPDF, the size of feature set is \(m = 9\). The possible values of \(y_i\) are either integer 0 (means non-packed) or 1 (means packed). Given a new \(X_i\), the classifier predicts the corresponding \(y_i\).

2) HYPER-PARAMETER TUNING

In machine learning, hyper-parameter tuning is the process of selecting a set of optimal hyper-parameters for a classification model [26]. It is important to optimize the hyper-parameters before the model training process as the appropriate hyper-parameters may influence the performance greatly. Researchers usually prepare several possible values for each hyper-parameter, then run randomized or grid search to seek for the combination that yields the highest score.

It is a common practice to apply \(K\)-fold Cross Validation (\(K\)-fold CV) to evaluate different hyper-parameter settings [27], [28]. The procedure of performing \(K\)-fold CV is summarized in Algorithm 3. It randomly divides original sample data into \(K\) disjoint subsets of the same size. One of these subsets is selected as testing set and the other \(K-1\) are merged to form the training set. We repeat above process \(K\) times and in each round every subset takes turn in serving as testing set and the remainder being the training set.

We summarize this evaluation method of hyper-parameter settings in Algorithm 4. For a specific model, the optimal hyper-parameter set can be determined by the mean of \(K\)-fold CV scores after searching all the possible combinations. To boost the efficiency of hyper-parameter tuning, it is suggested to apply some dedicated hyper-parameter optimization frameworks, such as Irace [27], scikit-learn [29], and Talos [30].

3) MODEL SELECTION

Model selection is the process of selecting the optimal one from a collection of candidate classification models. In this work, 9 types of powerful models are evaluated.
Algorithm 3 K-Fold Cross Validation

**Input:** $M$, $D$, $K$, scoring, where $M$ is the model to be evaluated; $D$ is the dataset; $K$ is the number of folds; scoring represents the scoring method.

**Output:** Performance scores of model $M$ trained on $K$ different training data sets.

1. $scores ← \text{Array}[\text{Create an array of size } K]$; $\triangleright$ Used to keep the scores of the model
2. $\triangleright$
3. $\text{for } i ← 0 \text{ to } K - 1 \text{ by } 1 \text{ do}$
4. $D_i^{\text{train}}, D_i^{\text{test}} ← \text{Do } i\text{th split on } D$;
5. $M' ← \text{Train } M \text{ on dataset } D_i^{\text{train}}$;
6. $\triangleright$
7. $y_{\text{pred}} ← \text{Predict } D_i^{\text{test}} \times \text{X with model } M'$;
8. $y_{\text{true}} ← D_i^{\text{test}}.y$; $\triangleright$ The ground truth targets
9. $\triangleright$
10. $score ← \text{Calculate metrics with } y_{\text{pred}} \text{ and } y_{\text{true}} \text{ according to } \text{scoring}$;
11. $scores[i] ← score$; $\triangleright$ Keep the score of current iteration
12. $\triangleright$
13. $\triangleright$
14. $\text{return } scores$

Algorithm 4 Scoring the Hyper-Parameter Settings Using K-Fold Cross Validation

**Input:** $M$, $D$, $K$, $P_{\text{sets}}$, scoring, where $M$ is the model to be evaluated; $D$ is the dataset; $K$ is the number of folds; $P_{\text{sets}}$ are the hyper-parameter settings to be evaluated; scoring represents the scoring method.

**Output:** Performance scores of different combinations of hyper-parameter settings acting on model $M$.

1. $ps\_score\_map ← \text{Map}[\text{Create a map}]$; $\triangleright$ Used to keep the performance score of each hyper-parameter setting in $P_{\text{sets}}$.
2. $\triangleright$
3. $\text{for each } P_{\text{set}} ∈ P_{\text{sets}} \text{ do}$
4. $M' ← \text{Initialize } M \text{ with } P_{\text{set}}$
5. $\triangleright$
6. $scores ← \text{Invoke Algorithm 3 by passing in the initialized model } M', \text{ dataset } D, \text{ number of folds } K, \text{ and scoring method } \text{scoring}$;
7. $\triangleright$
8. $ps\_score\_map[P_{\text{set}}] ← \text{Calculate the mean of the returned scores}$;
9. $\triangleright$
10. $\triangleright$
11. $\text{return } ps\_score\_map$

They are AdaBoost, Bagging, DecisionTree, GaussianNaive-Bayes, GradientBoosting, RandomForest, SVM, Xgboost, and LSTM [28], [31]–[33], respectively. All of them are frequently used models and many previous studies have applied them to recognize the patterns of observed subjects.

It is a common method to select the best model based on the $K$-fold CV results. However, this method has the problem that the training sets overlap with each other and information may thus leak into the model which will result in the over-fitting problem [34]–[36]. To address this challenge, we adopt Nested Cross Validation (Nested CV) [35] to evaluate these models. According to research [37], the Nested CV yields a nearly unbiased estimate of the true error. Nested CV is the extension of cross validation which consists of two nested loops of cross validation. The inner loop is used to perform the hyper-parameters tuning and the outer loop is used to compute an estimator of the error. This method is summarized in Algorithm 5.

IV. EVALUATION

In this section, we evaluate the performance of DroidPDF. The development of DroidPDF is done on Thinkpad s540 laptop, carrying 8G RAM and 4 physical processors. Each processor is Intel(R) Core(TM) i7-4500U CPU with 2 cores up to 1.80GHz. The operating system is Ubuntu 16.04 LST. All the Machine Learning models are backed by scikit-learn [29] and the version is 0.21.2. The Deep Learning models are backed by TensorFlow [38] of which the versions is 2.2.0.

A. DATASET

To evaluate the predictive ability of DroidPDF, we collect two sample sets, $S_1$ and $S_2$. $S_1$ consists of 1400 packed Android apps which are collected from VirusShare [39] (600 samples), AppChina [40] (500 samples) and our previous research (300 samples). Samples from our previous research are packed through invoking the online packing services of Ali, Bangcle, Baidu, Ijiami, Qihoo, and Tencent. For the rest 1100 packed samples, 1043 of them are identified by signature based method [9], [41]. The rest 57 samples are detected manually which is similar to research [17]. To be specific, we use the unpacker DexX [7] to recover their source code when running these samples on the physical devices. Runtime packers encrypt and conceal the original source code and only release it when the app is running. So we compare the recovered source code with the decompressed source code and mark the apps as packed if the source code does not match. Fig. 2 shows the packer distribution of all the packed apps. The majority of them are from the mainstream packer vendors, such as Ali, Bangcle, Baidu, Ijiami, Qihoo, and Tencent. $S_2$ is composed of 1400 non-packed Android apps which are obtained from F-Droid [42]. F-Droid is a website that provides source code of Android apps and it is often mentioned by other researches [9], [17] when seeking for non-packed samples. To ensure that all the samples from F-Droid are non-packed, we download all the source code from F-Droid which is used to compare with the corresponding samples. It turns out that all the samples are non-packed.
We prepare several datasets in advance which will be used in the following experiments. Note that we will follow a stratified fashion when partitioning the datasets. In other words, the proportion of samples of different labels (packed, non-packed) will be the same in each dataset. We split S1 into two datasets D_{pk1000} and D_{pk400} after shuffling. They consist of 1000 and 400 packed samples, respectively; we repeat this process and split S2 into D_{npk1000} and D_{npk400}. They consist of 1000 and 400 non-packed samples, respectively. Dataset D_{pk1000} and D_{npk1000} are used for model selection and parameter tuning in Sections IV-C and IV-E.

**Algorithm 5 Model Comparison Based on Nested Cross Validation**

**Input:** \{M_1, M_2, \ldots, M_n\}, D, K_1, K_2, where \{M_1, M_2, \ldots, M_n\} are the models to be evaluated; D is the dataset; K_1 is the number of folds of outer loop; K_2 is the number of folds of inner loop.

**Output:** Performance scores of \{M_1, M_2, \ldots, M_n\} of K_1 iterations.

\[
\begin{align*}
1: & \quad \text{model_scores_map} \leftarrow \text{Map}[\text{Create a map}]; \quad \triangleright \ 	ext{Used to keep the performance scores of models in } \{M_1, M_2, \ldots, M_n\}. \\
2: & \quad \text{for } i \leftarrow 0 \text{ to } K_1 - 1 \text{ by } 1 \text{ do} \\
3: & \quad \quad D_i^{\text{train}}, D_i^{\text{test}} \leftarrow \text{Do ith split on } D; \\
4: & \quad \text{for each } M_j \in \{M_1, M_2, \ldots, M_n\} \text{ do} \\
5: & \quad \quad \text{Let } P_{\text{sets}} \text{ denotes the set of hyper-parameters with different values}; \quad \triangleright \ 	ext{Hyper-parameter settings for model } M_j \\
6: & \quad \quad \text{ps_score_map} \leftarrow \text{Invoke Algorithm 4 by passing} \\
7: & \quad \quad \quad \text{in the model } M_j, \text{ dataset } D_i^{\text{train}}, \text{ hyper-parameter settings} \\
8: & \quad \quad \quad \quad P_{\text{sets}}, \text{ and scoring method } f_i; \quad \triangleright \ 	ext{Score each hyper-parameter setting} \\
9: & \quad \quad \quad \text{best_P_set} \leftarrow \text{Select the best hyper-parameter setting} \ 	ext{from } \text{ps_score_map} \ 	ext{by score value}; \\
10: & \quad \quad \quad M_j' \leftarrow \text{Initialize } M_j \ 	ext{with best_P_set}; \\
11: & \quad \quad \quad M_j'' \leftarrow \text{Train } M_j' \ 	ext{on dataset } D_i^{\text{train}}; \\
12: & \quad \quad \quad y_{\text{pred}} \leftarrow \text{Predict } D_i^{\text{test}} X \ 	ext{with model } M_j''; \\
13: & \quad \quad \quad y_{\text{true}} \leftarrow D_i^{\text{test}} y; \quad \triangleright \ 	ext{The ground truth targets} \\
14: & \quad \quad \quad F_1 \leftarrow \text{Calculate F1 score with } y_{\text{pred}} \ 	ext{and } y_{\text{true}}; \quad \triangleright \ 	ext{See Eq. (5), (6), (4)} \\
15: & \quad \quad \text{model_scores_map}[M_j][i] \leftarrow F_1; \quad \triangleright \ 	ext{Keep the performance score of model } M_j \ 	ext{of ith iteration} \\
16: & \quad \text{end for} \\
17: & \quad \text{end for} \\
18: & \quad \text{return model_scores_map} \\
\end{align*}
\]

**B. EVALUATION METRICS**

In this work, we use F1 score [43] as the metrics to evaluate the performance of different models. In statistical analysis of binary classification, the F1 score is a measure of accuracy of the given model. The F1 score can be deemed as a harmonic mean of metrics Precision and Recall. The formula of F1 score can be expressed as:

\[
F1Score = 2 \cdot \frac{\text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}}
\]

where Recall and Precision can be calculated as follows:

\[
\text{Precision} = \frac{TP}{TP + FP}
\]

\[
\text{Recall} = \frac{TP}{TP + FN}
\]

where TP, TN, FP, and FN represent True Positive, True Negative, False Positive, and False Negative, respectively. Positive indicates packed and Negative indicates non-packed. True means correct prediction and False means incorrect prediction.

**C. MODEL SELECTION FOR DroidPDF**

In this section, we detail the process of searching the optimal hyper-parameter setting and the method of choosing the optimal model for DroidPDF. Before the experiment, we combine the dataset D_{npk1000} and D_{pk1000} together to form the dataset D_{2000} for the follow-up evaluation.

In this experiment, we use 5-fold CV to tune up the hyper-parameters of the models. It is necessary to point out that in K-fold CV experiments, the choice of K is a trade-off between bias and variance. Generally speaking, K is an empirical value and researchers usually choose K=5 or K=10. The larger K is, the more data is added to the training set, the smaller the bias of the model will be. However, with larger K, there will be greater correlation among training sets (considering the most extreme example, when K is the size of the data set, also known as Leave One Out Cross Validation [44], every time the training sets are almost the same). The large correlation of training sets will result in a
large variance on the final result. For each model, we work out the $K$-fold CV scores of all the possible combinations of hyper-parameters by invoking Algorithm 3. The optimal hyper-parameter setting is determined by the mean of $K$-fold CV scores. The evaluation results are presented in Table 2. The second column presents the best hyper-parameter setting of each model; the third column presents the scores achieved by corresponding hyper-parameter settings.

As explained in Section III-D, it is recommended that researchers adopt Nested CV to evaluate the performance of different models rather than Non-nested CV. We adopt $5 \times 2cv$ [45] method which is one of the implementation of Nested CV and recommended by MLxtend [46] for practical usage. For $5 \times 2cv$, the outer loop is based on 5-fold CV and the inner loop is based on 2-fold CV. Table 3 presents the performance of 9 different models after performing $5 \times 2cv$. As shown in Table 3, Xgboost outperforms other models and achieves an average F1 Score of 0.9870. This fact leads us to choose Xgboost as the classification model of DroidPDF. It is a common practice to fit the final model to the whole dataset after hyper-parameter optimization [29]. Therefore, with the optimal hyper-parameter setting in Table 2, we train Xgboost on $D_{2000}$ and name the trained model as $M_\alpha$ for the further use.

### D. COMPARISON OF EXISTING RESEARCHES

In this section, we prove that our DroidPDF outperforms some typical state-of-the-art researches. APKiD [41] and Liao et al. [47] are compared in this experiment. APKiD is a signature-based solution of packer detection which identifies many types of packers, compilers, and obfuscators; Liao et al. presented a framework based on the combination of static analysis and dynamic analysis. Firstly, it extracts all the classes names from the APK file through static analysis. Secondly, it modifies the class loading function `Dalvik_dalvik_system_DexFile_defineClassNative` and dump all the class names through dynamic analysis. Finally, it compares the class names from static and dynamic analysis. If class names only exist in the result of dynamic analysis, this means the app contains packed code.

To evaluate each packer detection framework, we adopt datasets $D_{pk400}$ and $D_{npk400}$ which consist of 400 packed and 400 non-packed samples, respectively. The results are listed in Table 4. It is obvious that our model outperforms

---

**TABLE 2. Hyper-parameter Tuning Based on 5-fold CV.**

| Model            | Best Hyper-parameters                        | Best Score |
|------------------|----------------------------------------------|------------|
| AdaBoost         | {'n_estimators': 1, 'base_estimator__criterion': 'entropy', 'base_estimator__splitter': 'random'} | 0.975606   |
| Bagging          | {'max_samples': 0.5, 'base_estimator__max_depth': 5} | 0.975627   |
| DecisionTree     | {'criterion': 'entropy', 'max_depth': 11}     | 0.975691   |
| GaussianNaiveBayes | {'None'}                                          | 0.924789   |
| GradientBoosting | {'min_samples_split': 1000, 'min_samples_leaf': 70} | 0.977537   |
| RandomForest     | {'max_features': 4, 'n_estimators': 30}        | 0.986895   |
| SVM              | {'penalty': 'l1', 'dual': False}               | 0.961272   |
| Xgboost          | {'subsample': 0.6, 'colsample_bytree': 0.8, 'max_depth': 6, 'gamma': 0.1, 'min_child_weight': 1} | 0.986284   |
| LSTM             | {'activation': 'relu', 'batch_size': 30, 'dropout': 0, 'epochs': 100, 'first_neuron': 10, 'hidden_layers': 0, 'kernel_initializer': 'normal', 'last_activation': 'sigmoid', 'losses': 'binary_crossentropy', 'optimizer': 'Nadam'} | 0.9347     |

**TABLE 3. Model comparison based on 5 $\times$ 2cv.**

| Model            | Fold 1  | Fold 2  | Fold 3  | Fold 4  | Fold 5  | Mean    | Stdev   |
|------------------|---------|---------|---------|---------|---------|---------|---------|
| AdaBoost         | 0.9752  | 0.9849  | 0.9576  | 0.9802  | 0.9751  | 0.9746  | 0.0092  |
| Bagging          | 0.9825  | 0.9899  | 0.9676  | 0.9751  | 0.9778  | 0.9786  | 0.0075  |
| DecisionTree     | 0.9703  | 0.9849  | 0.9674  | 0.9851  | 0.9850  | 0.9785  | 0.0080  |
| GaussianNaiveBayes | 0.9330  | 0.9295  | 0.9333  | 0.9394  | 0.9044  | 0.9279  | 0.0122  |
| GradientBoosting | 0.9774  | 0.9850  | 0.9824  | 0.9826  | 0.9726  | 0.9800  | 0.0045  |
| RandomForest     | 0.9874  | 0.9925  | 0.9799  | 0.9874  | 0.9875  | 0.9869  | 0.0040  |
| SVM              | 0.9622  | 0.9751  | 0.9846  | 0.9576  | 0.9596  | 0.9638  | 0.0061  |
| Xgboost          | 0.9850  | 0.9925  | 0.9825  | 0.9900  | 0.9851  | **0.9870** | 0.0037  |
| LSTM             | 0.9284  | 0.9418  | 0.9406  | 0.9563  | 0.9590  | 0.9452  | 0.0112  |
TABLE 4. Performance comparison with other two state-of-the-art packer detection frameworks.

| Framework   | TP  | FN  | TN  | FP  | F1     |
|-------------|-----|-----|-----|-----|--------|
| APKiD       | 360 | 40  | 400 | 0   | 0.9474 |
| Liao’s      | 369 | 31  | 400 | 0   | 0.9597 |
| DroidPDF    | 398 | 2   | 398 | 2   | 0.9950 |

other 2 models. Notably, DroidPDF achieves an average score of 0.9950, meanwhile APKiD only achieves 0.9474 and Liao’s only achieves 0.9597.

Here we analyze the reasons of the failures of APKiD and Liao’s framework, respectively. (1) For APKiD, although fingerprint detection methods can detect packed apps, they are less versatile in the real world environment. The limitation of fingerprint based tools is that they can not recognize never seen features. Through our research, we find that APKiD fails to detect most Tencent packers since it has no signatures of related packers. Due to the fact that packers have been evolving all the time and more and more custom designed packers are devised, the traits of packers will change over time and the fingerprint based detection methods become less effective. (2) For Liao’s framework, we find that many apps protected by Ali and Ijiami packers conceal method code instead of the whole classes. According to Liao’s method, apps of which the class names from static and dynamic analysis being the same should be classified as non-packed. But the fact is that apps of same class names are still packed for Ali and Ijiami packers.

E. ENTROPY ANALYSIS

In this section, we validate the effectiveness of weighted entropy. As we researched in Section II-B, the entropy of apps will more or less increase after being packed by the mainstream packers. This phenomenon leads us to devise weighted entropy as a trait to help to distinguish the packers. We conduct a controlled experiment including statistical test to measure whether it is statistically significant to have weighted entropy in the feature set. In other words, we examine whether there is significant difference with or without weighted entropy.

In this experiment, we repeat the process in Section IV-C to work out the optimal model on dataset $D_{2000}$ without weighted entropy. To be more specific, we remove the feature column of weighted entropy from the dataset, then evaluate the models as previous process. Table 5 shows the F1 scores of 5 iterations. It is evident that Xgboost outperforms other models. Consequently, in the controlled experiment, we employ Xgboost as the classifier for dataset without weighted entropy.

Comparing the best mean score in Table 3 (achieved by Xgboost, 0.9870) with that in Table 5 (achieved by Xgboost, 0.9779), we can see performance decline when the feature of weighted entropy is removed. However, we need to determine whether this happens by chance or not. Here we apply independent two-sample t-test [48] to compare the performance of these two models in a more comprehensive manner. The statistical significance of the t-test indicates whether the difference between two models is significant or not. We set a significance threshold of $\alpha = 0.05$ for rejecting the null hypothesis $H_0$ that both models perform equally well.

In our scenario, statistical significance suggests the performance difference between these two models trained on datasets with and without weight entropy. Let $x$ and $y$ denote these two models to compare, The formula of $t$-value is presented in Eq. (7).

$$t-value = \frac{\bar{x} - \bar{y}}{\sqrt{\sigma^2 \cdot \left( \frac{1}{n_x} + \frac{1}{n_y} \right)}}$$  \hspace{1cm} (7)

where $\bar{x}$ and $\bar{y}$ represent the means of performance scores of model $x$ and $y$, respectively; $n_x$ and $n_y$ represent the number of performance scores of model $x$ and $y$, respectively; $\sigma^2$ is the pooled variance of the given samples. It is expressed as follows:

$$\sigma^2 = \frac{(n_x - 1) \cdot \sum_{i=1}^{n_x} (x_i - \bar{x})^2 + (n_y - 1) \cdot \sum_{j=1}^{n_y} (y_j - \bar{y})^2}{n_x + n_y - 2}$$  \hspace{1cm} (8)

TABLE 5. Model comparison on dataset without weighted entropy based on 5 x 2cv.

| Model            | Fold 1 | Fold 2 | Fold 3 | Fold 4 | Fold 5 | Mean    | Stdev   |
|------------------|--------|--------|--------|--------|--------|---------|---------|
| AdaBoost         | 0.9531 | 0.9630 | 0.9550 | 0.9655 | 0.9578 | 0.9589  | 0.0047  |
| Bagging          | 0.9681 | 0.9825 | 0.9730 | 0.9679 | 0.9754 | 0.9734  | 0.0054  |
| DecisionTree     | 0.9633 | 0.9676 | 0.9731 | 0.9723 | 0.9550 | 0.9663  | 0.0066  |
| GaussianNaiveBayes | 0.9418 | 0.9247 | 0.9373 | 0.9502 | 0.9273 | 0.9363  | 0.0094  |
| GradientBoosting | 0.9570 | 0.9750 | 0.9697 | 0.9703 | 0.9476 | 0.9639  | 0.0101  |
| RandomForest     | 0.9652 | 0.9824 | 0.9719 | 0.9849 | 0.9698 | 0.9748  | 0.0076  |
| SVM              | 0.9426 | 0.9676 | 0.9455 | 0.9383 | 0.9450 | 0.9478  | 0.0102  |
| Xgboost          | 0.9700 | 0.9799 | 0.9799 | 0.9875 | 0.9723 | 0.9779  | 0.0062  |
| LSTM             | 0.9319 | 0.9390 | 0.9243 | 0.9449 | 0.9111 | 0.9302  | 0.0118  |
where $x_i$ and $y_i$ represent the performance scores of model $x$ and $y$, respectively; $n_x + n_y - 2$ is the degrees of freedom (df) which is $5 + 5 - 2 = 8$ in this experiment.

According to Eq. (7), we work out the $t$-value of 2.52. Since $t$-value and the degrees of freedom $df$ are determined, the $p$-value turns out to be 0.036. The fact that $p$-value = 0.036 is less than $\alpha = 0.05$ indicates that the performance difference between these two models is significant. In other words, the enhancement of performance is statistically significant after including weighted entropy to the feature set.

### F. OBfuscATION ANALYSIS

In this section, we validate the model’s resilience to the obfuscation techniques. Through investigation, we find that identifier obfuscation performed on apps’ code would cause much more drastic change (increase or decrease) to the entropy than other obfuscation techniques. However, apps simply adopting identifier obfuscation does not mean they are packed since the source code is not concealed. Since our framework is partially based on entropy analysis, the change of entropy may be an adverse impact to our framework. We can see the difference between the original code and the identifier obfuscated code in Fig. 3.

There are two major challenges for DroidPDF when facing obfuscation technique. For packed apps, there are chances that some packers may use identifier obfuscation techniques to obfuscate the source code which may result in the entropy decrease; for the non-packed apps, the identifier obfuscation acting on normal apps may result in entropy increase. To prove that DroidPDF is obfuscation resilient, the most straightforward approach is to use model $M_{\alpha}$ to identify obfuscated datasets (including packed and non-packed samples). Then demonstrate that there is no significant difference between scores achieved on obfuscated and non-obfuscated datasets.

To generate obfuscated datasets, we use two obfuscators, Proguard [49] and Obfuscapk [50], to perform the obfuscation on $D_{pk400}$ and $D_{npk400}$. The process of performing obfuscation is detailed as follows. We randomly select 400 samples from $D_{pk400}$ and $D_{npk400}$ (each dataset 200 samples) and use Proguard to obfuscate them; the rest 400 samples of from $D_{pk400}$ and $D_{npk400}$ are obfuscated by Obfuscapk. Finally we produce two obfuscated datasets $D_{obfpk400}$ (packed) and $D_{obfpk400}$ (non-packed). It turns out that the entropy of 80% samples decreased after being obfuscated by Proguard and 92.2% samples’ entropy increased after being obfuscated by Obfuscapk. According to our study, ProGuard uses a great deal of special characters. This fact will result in relatively lower entropy for samples due to the high occurrence of same character; Obfuscapk is different from Proguard as it uses a serials of more random sequences which can result in relatively higher entropy for samples.

We conduct an experiment that iterates 10 times. In each iteration, we regulate the proportion of obfuscated samples to study the influence of obfuscated apps to our framework. For example, if we want to form a dataset of which the proportion of obfuscated samples is 10% (80 obfuscated samples), we can randomly select 40, 40, 360, and 360 samples from $D_{obfpk400}$, $D_{obfpk400}$, $D_{pk400}$, and $D_{npk400}$, respectively. Note that samples in $D_{obfpk400}$ are from $D_{pk400}$ and samples in $D_{obfpk400}$ are from $D_{npk400}$. To avoid duplication, the same sample can not be chosen twice in each iteration. For convenience, we name these test datasets as $D_{mix 10}$, $D_{mix 20}$, $D_{mix 30}$, and $D_{mix 100}$, respectively. The subscript in the dataset name indicates the proportion of obfuscated samples in percentage.

We use model $M_{\alpha}$ to predict on these test datasets. Table 6 shows the experiment results. To evaluate the performance of $M_{\alpha}$ on obfuscated samples in a more comprehensive way, we use paired t-test to measure the statistical significance. According to research [35], the Nested CV method reduces the bias and produces an estimator of the test error which is highly close to that gained on the independent testing set. We reuse the scores of Xgboost in Table 3 as a surrogate of performance estimation of $M_{\alpha}$ to compare with the performance in Table 6. By applying Eq. (7), the degrees of freedom $df$, $t$-value, and $p$-value turns out to be 13, 1.97, and 0.07, respectively. Since $p$-value=0.07 is greater than $\alpha = 0.05$, this outcome indicates that there is no significant difference between the scores of $M_{\alpha}$ achieved on normal and obfuscated datasets. Through above statistical analysis,
TABLE 6. Model performance on datasets of different proportions of obfuscated samples.

| Model | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% | 100% | Mean  | Stdev |
|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|-------|-------|
| $\mathcal{M}_d$ | 0.9912 | 0.9849 | 0.9862 | 0.9837 | 0.9812 | 0.9825 | 0.9812 | 0.9774 | 0.9761 | 0.9748 | **0.9819** | 0.0047 |

---

we can conclude that DroidPDF is resilient to the obfuscation technique.

G. EFFICIENCY ANALYSIS

It is obvious that the processing time is also considered as another performance indicator for a large scale classification system. We combine $S_1$ and $S_2$ together to form the test set. For each sample, DroidPDF performs the feature extraction operation for 10 times and take the average as the actual processing time. Fig. 4 shows the processing performance of DroidPDF.

Fig. 4a shows the APK decompressing time against the APK file size. The average APK decompressing rate is around 60MB/s. We notice a linear increase of decompression time in Fig. 4a which means that the APK decompression time converges linearly with the subject size.

Fig. 4b shows the DEX file parsing time against the DEX file size. The average DEX file parsing rate is around 15MB/s. In Fig. 4b the DEX processing time decreases unusually for some DEX files larger than 8MB. Through analysis, we find that these DEX files are packed by Tencent packers. We also observe that these Tencent packers insert invalid code chunks to pad the DEX files, which makes these DEX files look normal. Since it only contains a few of valid classes, it will not take too much time to process it.

Fig. 4c shows the entropy calculating time against the subject size. The average entropy calculating rate is around 284MB/s. We can see a linear increase of entropy calculating
time which means that the entropy calculating time converges linearly with the size of subjects. Fig. 4d shows the overall processing time for a whole APK including decompressing, DEX file parsing, and entropy calculating. The overall processing rate is around 14MB/s. In Fig. 4d the processing time does not converge linearly for many samples greater than 70MB. After investigation, it turns out to be the problem of the resource files. For most samples larger than 70MB, the size of DEX file and native libraries has seen little growth. As a matter of fact, the actual growth comes from the resource files. Since the resource files are treated as uncorrelated, it explains why the processing time does not increase along the sample size.

V. DISCUSSION
In this section, we discuss some practical methods to boost the speed of DroidPDF.

The programming language that implements the project matters tremendously for the processing time. It is strongly recommended that researchers adopt C programming language to speed up the entropy calculation since it is a computation-intensive task. According to our test, C language implementation is almost 40 times faster than the counterpart written in Python.

To improve the processing efficiency of DroidPDF in real life production, we encourage researchers to incorporate the fingerprint technique, which operates as a filter to pick out the already known packers before sending to the classification engine. In turn, for the never seen packers detected by DroidPDF we can extract the fingerprints which can be utilized to enrich the fingerprint repository.

VI. RELATED WORK
In large scale analysis of Android malicious apps, it is necessary to identify packed apps in advance so that we can adopt different analysis workflows to facilitate the process of automatic identification. To address this challenge, there are several prior contributions with respect to packer detection that can be referred to.

APKiD [41] is an Android packer recognition tool based on fingerprint identification. Although fingerprint detection methods can detect packed apps, they are less versatile in the real world environment. The limitation of fingerprint based tools is that they can not recognize never seen features. In other words, APKiD always lags behind when the new packers come out.

Bintropy [17] is the earliest attempt to use entropy to identify packed apps of Windows Portable Executable (PE) format. This is a good inspiration for packer analysis. But we should be aware of the fundamental difference between PE and APK. The difference is that APK is a compressed file and calculating entropy on it will cause high entropy issue.

Chau et al. [51] also proposed an entropy analysis solution. It selects several typical resource files and calculates their entropy to form feature vector. This method is very innovative except one limitation. That is the entropy analysis is not resilient to obfuscation techniques, especially the identifier obfuscation. Obfuscated identifier may carry lower or higher entropy sequences, which will have a negative impact on the results.

Liao et al. [47] presented a framework based on the combination of static analysis and dynamic analysis. According to Liao’s method, apps of which the class names from static and dynamic analysis being the same should be classified as non-packed. As we analyzed in Section IV-D, they failed to detect some samples packed with Ali and Ijiami packers since these two packers encrypt the method instructions instead of the whole classes.

The reason we use multi-dimensional features is that recognizing packer by the entropy threshold is similar to fingerprint in principle. The entropy value of a packer can be changed, and other non-packed apps’ entropy values may also locate within this range due to the change of SDK [52], obfuscation techniques, or other factors. Hence, we introduce other entropy irrelevant features to make the classification system more robust.

VII. CONCLUSION
In this study, we proposed a packer detection framework which is based on a set of heuristic features to recognize the packed and non-packed apps. We introduce weighted entropy to enhance the performance and achieved a good score. According to the experiment outcomes, Xgboost classifier outperforms other state-of-the-art frameworks. Our DroidPDF shows a promising prospect as a practical framework to detect not only the known packers but also the unknown ones.

REFERENCES
[1] StatCounter. Mobile Operating System Market Share Worldwide—Sep. 2019. Accessed: Sep. 15, 2019. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide
[2] B. O. Apps, App Download and Usage Statistics (2019). Staines, U.K., Mansoor Iqbal, Accessed: Sep. 15, 2019 [Online]. Available: https://www.businessofapps.com/data/app-statistics/#1
[3] McAfee. McAfee Mobile Threat Report. Accessed: Sep. 15, 2019. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/mobile-threat-report-2019.pdf
[4] AVL.Sec. 2017 Annual Report. Accessed: Sep. 15, 2019. [Online]. Available: http://blog.avlsec.com/2018/03/5150/2017-annual-report/
[5] Y. Duan. “Things you may not know about Android (Un) packers: A systematic study based on whole-system emulation,” in Proc. Netw. Distrib. Syst. Secur. (NDSS) Symp., San Diego, CA, USA, Feb. 2018, pp. –15.
[6] P. Graux, J.-F. Lalande, and V. T. Tong. “Obfuscated Android application development,” in Proc. 3rd Eur. Cybersecurity Conf., Munich, Germany, Nov 2019, pp. 1–6.
[7] C. Sun, H. Zhang, S. Qin, N. He, J. Qin, and H. Pan. “DexX: A double layer unpacking framework for Android,” IEEE Access, vol. 6, p. 61267, 2018.
[8] Yu, Rowland. “Android packers: Facing the challenges, building solutions,” in Proc. 24th Virus Bull. Int. Conf., Seattle, WA, USA. Sep. 2014, pp. 1–4.
[9] Zhang, Yueqian, Xiapu Luo, and Haoyang Yin. “DexHunter: Toward extracting hidden code from packed Android applications,” in Proc. 3rd Eur. Symp. Res. Comput. Secur. (ESORICS), Sep. 2015, pp. 293–311.
[10] L. Xue, L. Yuan, L. Yu, S. Wang, and D. Wu. “Adaptive unpacking of Android apps,” in Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE), May 2017, pp. 358–369.
Y. Feng, S. Anand, I. Dillig, and A. Aiken, “Appscoppy: Semantics-based detection of Android malware through static analysis,” in Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. - FSE, Hong Kong, 2014, pp. 576–587.

Y. Du, J. Wang, and Q. Li, “An Android malware detection approach using community structures of weighted function call graphs,” IEEE Access, vol. 29, pp. 17478–17486, 2017.

W. Wang, Z. Gao, M. Zhao, Y. Li, J. Liu, and X. Zhang, “Droid Ensemble: Detecting Android malicious applications with ensemble of string and structural static features,” IEEE Access, vol. 6, pp. 31798–31807, 2018.

B. CM., “Introduction.” in Pattern Recognition Machine Learning, 1st ed. New York, NY, USA: Springer, 2006, pp. 1–11.

X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, “SoK: Deep packer inspection: A longitudinal study of the complexity of runtime packers,” in Proc. IEEE Symp. Secure. Privacy, San Jose, CA, USA, May 2015, pp. 659–673.

Wikipedia. Entropy. Accessed: Sep. 15, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Entropy

R. Lyda and J. Hamrock, “Using entropy analysis to find encrypted and packed malware,” IEEE Secur. Privacy Mag., vol. 5, no. 2, pp. 40–45, Mar. 2007.

Alibaba Corporation. Accessed: Jul. 1, 2018. [Online]. Available: http://jaq.alibaba.com/

Baidu Corporation. Accessed: Jul. 1, 2018. [Online]. Available: http://app.baidu.com/jiaju

Bingele Corporation. Accessed: Jul. 1, 2018. [Online]. Available: http://www.bingele.com/

Tijiami Corporation. Accessed: Jul. 1, 2018. [Online]. Available: http://www.tijiami.cn/

Tencent Corporation. Accessed: Jul. 1, 2018. [Online]. Available: http://ya.qq.com/

S. H. Smith. Unzip—Unix. Linux Command. Accessed: Sep. 17, 2019. [Online]. Available: https://www.tutorials��点com/unix_commands/unzip.htm

Debian. The Computer Language Benchmarks Game. Accessed: Apr. 22, 2020. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/regexredux.html

Wikipedia. Hyperparameter Optimization. Accessed: Apr. 15, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Hyperparameter_optimization

M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, M. Birattari, and T. Stützle, “The irace package: Iterated racing for automatic algorithm configuration,” Oper. Res. Perspect., vol. 3, pp. 43–58, Sep. 2016.

S. Raschka. Python Machine Learning. Birmingham, U.K.: Packt, 2015.

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 É. Duchesnay, “Scikit-learn: Machine learning in Python,” in Proc. JMLR, vol. 11, pp. 2079–2107, Jul. 2010.

T. Dietterich. “Approximate statistical tests for comparing supervised classification learning algorithms,” Neural Comput., vol. 10, no. 7, pp. 1895–1923, Oct. 1998.

G. C. Cawley and N. L. Talbot, “On over-fitting in model selection and subsequent selection bias in performance evaluation,” J. Mach. Learn. Res., vol. 11, pp. 2079–2107, Jul. 2010.

F. Domingos, “A few useful things to know about machine learning,” Commun. ACM, vol. 55, no. 10, pp. 78–87, Oct. 2012.

Varma, Sudhir, and Richard Simon, “Bias in error estimation when using cross-validation for model selection,” BMC Bioinf., vol. 7, no. 91, Feb. 2006.

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, and M. Kudlur, “Tensorflow: A system for large-scale machine learning,” in Proc. 12th Symp. Oper. Syst. Des. Implement. (OSDI), Savannah, GA, USA, Nov. 2016, pp. 265–283.

VOLUME 8, 2020

CAIJUN SUN was born in Yuyao, Zhejiang, China, in 1990. He received the B.S. degree in software engineering from Hangzhou Normal University, China, in 2013. He is currently pursuing the Ph.D. degree in computer science with the Beijing University of Posts and Telecommunications, China. His primary current research interests include malware analysis, and security of mobile systems and apps.

HUA ZHANG (Member, IEEE) received the B.S. degree in telecommunications engineering and the M.S. degree in cryptography from Xidian University, in 1998 and 2005, respectively, and the Ph.D. degree in cryptography from the Beijing University of Posts and Telecommunications, in 2008. She is currently an Associate Professor with the Beijing University of Posts and Telecommunications. Her research interests include cryptography, information security, and network security.

SUJUAN QIN was born in Shijiazhuang, Hebei, China, in 1979. She received the B.S. degree in computer science and technology from the Shijiazhuang Railway Institute, in 2003, and the Ph.D. degree in cryptography from the Beijing University of Posts and Telecommunications (BUP), in 2008. From 2008 to 2010, she was a Postdoctoral Researcher in information and communications engineering with BUP. Since 2011, she has been an Assistant Professor with the Institute of Network Technology, BUP. She is the author of one book and more than 70 articles. Her research interests include information security, network security, cryptography, and quantum cryptography.
JIAWEI QIN was born in Benxi, Liaoning, China, in 1993. He received the B.S. degree in computer science from Shenyang Aerospace University, China, in 2015. He is currently pursuing the Ph.D. degree in computer science with the Beijing University of Posts and Telecommunications, China. His research interests include malware analysis and security of Android systems.

YIJIE SHI received the B.S. degree in radio technology from the Dalian University of Technology, Dalian, Liaoning, China, in 1997, and the M.S. degree in cryptography and the Ph.D. degree in computer science from the Beijing University of Posts and Telecommunications, Beijing, China, in 2000 and 2014, respectively. She is currently a Lecturer with the Beijing University of Posts and Telecommunications. Her major research interests include security of industrial control systems, security of cloud computing, and cryptography.

QIAOYAN WEN received the B.S. and M.S. degrees in mathematics from Shaanxi Normal University, Xi’an, China, in 1981 and 1984, respectively, and the Ph.D. degree in cryptography from Xidian University, Xi’an, in 1997. She is currently a Professor with the Beijing University of Posts and Telecommunications. Her current research interests include coding theory, cryptography, information security, Internet security, and applied mathematics.

***