CAPIL: Component-API Linkage for Android Malware Detection

Zhaoheng Yang[a], Frank Breitinger[a,*], Ibrahim Baggili[a]

[a] *Cyber Forensics Research and Education Group (UNHcFREG)*
Tagliatela College of Engineering, University of New Haven, West Haven CT, 06516, United States

Abstract

The popularity of Android devices has increased in recent times, leading to an pervasive growth in the amount and diversity of malicious applications. This situation has challenged traditional signature-based malware identification techniques, and therefore machine learning approaches have recently become popular. In this work, we introduce CAPIL (Component-API Linkage), a novel approach for detecting Android malware. First, we describe a new procedure for feature extraction, where we link APIs to their components. Second, we present a unique procedure for feature selection / refinement. In the last step we use the well-known J48 classifier to generate our final model / decision tree. The experimental results show that CAPIL obtains an accuracy of over 98 % in our sample set of 12,958 applications (11,700 benign; 1,258 malicious). Thus, it can reliably detect both malware and benign applications. Furthermore, we illustrate that our method for feature selection is appropriate, and can improve information gain – a widely adopted feature selection method.

Keywords: Android, API-Component Linkage, Feature selection, Feature extraction, Malware detection.

1. Introduction

The proliferation of malware on Android has caused a growing demand for designing effective tools and techniques for the detection and analysis of Android applications. While the research in mobile malware detection started with classical fingerprint-based approaches (e.g., extracting permissions), many current approaches involve data mining and machine learning for more robust malware identification. Generally speaking, machine learning consists of three parts: (1) feature extraction[1], (2) feature refinement and (3) model creation. Existing approaches mainly focus on feature extraction and use data science for selecting / weighting features as well as creating the model.

There are four major contributions / outcomes of our work:

1. *Feature extraction* – we present a novel approach to define features based on APIs and their usage named Component-API Linkage (CAPIL). More precisely, we link the Android Application Programming Interfaces (APIs) with one or more of their application components – activity, content provider, service and broadcast receiver.

2. *Feature refinement* – additionally we introduce a self-devised approach based on the centered cosine similarity to identify differences in the average usage of APIs in various application types (begin and malicious).

3. *CAPIL* – the final outcome of this work is a tool (a python prototype implementation) called CAPIL which utilizes our feature extraction as well as feature refinement method. In case we talk about the sub-function (i.e., feature selection or refinement), we indicate this by using the terms CAPIL-extraction or CAPIL-refinement.

4. *Data set* – while our malware samples are freely available, we will publish the benign samples that may be used in future research.

Note, *feature extraction* is independent from the *feature refinement* and vice versa. In other words, extracted features may be refined by any common feature selection / refinement algorithm like information gain.

The motivation for linking APIs with components is that it enables us to realize where these APIs are being called from, as some APIs are typically less / not called from certain application components in benign applications. This makes our approach unique (explained in more detail in Sec. 2.2). Here are two examples:

- The `startRecording()` API can be used in benign and malicious applications. While most benign applications call it actively (of course they might be a few exceptions), e.g., for sending a voicemail to a known person (activity component), malware applications tend to use it to audio record the surrounding environment without a user's knowledge (service component). In this case the API is linked to an activity component for benign application and a service component in potential malware.

- The `openConnection()` API is used to establish a connection in order to send and receive data over the Internet.

*Corresponding author.
Email addresses: `zhaoheng1988@gmail.com` (Zhaoheng Yang), `fbreitinger@newhaven.edu` (Frank Breitinger), `ibaggili@newhaven.edu` (Ibrahim Baggili)

[1] Literature also often uses the term attribute instead of feature in data mining / machine learning. In this article, we use these terms synonymously.

It is usually used for streaming downloads and uploads in either the front end or back end (activity or service component respectively). However, malware tends to invoke this function in the Short Message Service (SMS) message receiver component aiming at setting up a connection to a remote server in order to gain access to text messages. This way, each time an incoming message triggers the SMS receiver, the message is secretly copied to a remote server. In this case, the API is linked to an activity or service component for benign application and receiver component for potential malware.

Our experiments demonstrate that CAPIL achieves an accuracy of 98.84 % in our sample set of 12,958 Android applications (11,700 benign and 1,258 malicious applications; ratio $\approx 1/10$. Furthermore, we also show that our feature selection procedure (a.k.a. feature refinement) can improve the widely adopted information gain feature selection technique. The prototype including instructions will be freely available after publication on our website.

The rest of the paper is structured as follows: The next section explains Android applications. In Sec. 3, we discuss the related work. The core of this paper is Sec. 4, which presents the methodology. Our experimental results are shown in Sec. 5. The last two sections conclude this paper and discuss the limitations of our work.

2. Android application package (APK) files

This section outlines the general structure and content of Android applications. From a high level perspective, Android applications are basically zip files that contain the actual implementation and application components including programming code, resources, assets, certificates, and a manifest file. The upcoming subsections describe the content of APK files most relevant to our work. Readers familiar with Android applications and their structure may want to skip reading this section.

2.1. Programming code

Android applications are written in Java and compiled into Dalvik bytecode (known as Dalvik EXecutable, `.dex` files) which can then be decompiled into *smali*-code. "smali/baksmali is an assembler/disassembler for the dex format used by Dalvik, Android's Java Virtual Machine (VM) implementation. The syntax is loosely based on Jasmin's/dedexer's syntax, and supports the full functionality of the dex format (annotations, debug info, line info, etc.)"[2].

2.2. AndroidManifest.xml

Every Android application comes with a manifest file which holds essential information about the application the Android system *must* read before it can run its code. More precisely, the system will not allow an application to access / run components or resources that are not declared in the manifest file (Enck et al., 2009b). The manifest file is roughly divided into two sections; *application components* and *uses-permissions*.

[2] `https://code.google.com/p/smali/` (last accessed April 5th, 2016).

2.2.1. Application components.

The first essential part of the manifest file and each application is the application components. Each component serves a portion of an application's overall functionality. An application can perform user interaction when the user taps the application's icon, which then opens the application's user interface. Therefore, it is critical to note that one application may have multiple components; each aiding in performing part of its various functions. They are described in the following paragraphs:

- **Activities component** provides visualized user interfaces. Each activity component provides a specific screen with which users can perform some actions or request some tasks. For example, an activity can be a screen with an active camera and a button to take a picture in a photo application or a scrolling screen of calendars in a hotel booking application. The purpose of this component is to provide a method for the application to interact with users.

- **Services component** usually performs certain tasks continually without interacting with users in the background. For example, a music player application could run as a service when the user closes the main screen. A malicious application could use the service component to send information to remote servers, receive harmful commands from them or record its environment.

- **Receivers component** (a.k.a. broadcast receivers) enables applications to receive notifications or messages from the system or other applications. Common messages from the system can be time-zone changes and low-battery warnings. Since not all messages are useful to an application on an Android device, a filter may be applied to eliminate messages of no interest by a specific application.

- **Providers component** (a.k.a. content providers) creates a connection from an application's database to another application it intends to share data with. An application that provides shared databases needs to declare a content provider component in its manifest file. A uses-permission for controlling the access to a provider needs to also be defined in the manifest file.

2.2.2. Uses-Permissions

Android systems have two different kinds of permissions called uses-permissions and permissions. The latter are offered permissions where an application itself allows other applications access to its data. The uses-permissions are claimed permissions which the application wants to have from the Android system and are granted by the user at installation time. An Android application must declare all desired uses-permissions inside the `AndroidManifest.xml` file. An example of a uses-permission is `READ_CONTACTS`, which if granted, allows for reading all the contacts in the address book.

3. Related work

In mobile malware detection and analysis, existing approaches are mainly divided into three categories; static anal-

ysis, dynamic analysis and machine learning based approaches. Literature related to each of these approaches are reviewed in the following three subsections.

3.1. Detection using static analysis

Several methods have been proposed that statically examine applications and disassemble their code (Chin et al., 2011; Schmidt et al., 2009; Zheng et al., 2012). For instance, Grace et al. (2012a) statically check vulnerability-specific signatures in an application's source code, then they categorize applications into three danger levels based on the riskiness of the vulnerability specific signatures found in applications being examined.

Enck et al. (2011) studied the 1100 top free applications from the official Android Market, also known as Google Play, and uncovered their misuse of private sensitive information. In particular, they found that applications were leaking data such as phone identifiers and geographic locations.

Felt et al. (2011) analyzed API calls and permissions to detect over-privileged applications. Zhou & Jiang (2012) systematically analyzed Android malware based on installation, activation mechanisms as well as the nature of their carried payload in order to categorize them into malware families. Enck et al. (2009a) examined application manifest files by looking for suspicious action strings used by activities, services or broadcast receivers and dangerous combinations of declared permissions.

Elish et al. (2015) focused on classifying benign Android inter-component communication (ICC) flows from colluding ones by statically constructing ICC Maps to capture pairwise communicating ICC channels of each benign application.

3.2. Detection using dynamic analysis

Dynamic analysis studies focused on the detection of Android malware by monitoring the behavior of applications such as monitoring file changes, network activity, processes, threads and system call tracing (Bläsing et al., 2010; Egele et al., 2008; Portokalidis et al., 2010). There are, however, issues with dynamic analyis - mainly that malware authors tend to apply anti-analysis techniques to evade dynamic analysis in emulated Android environments. An example is malware that is capable of detecting being used in virtual environments thus being able to stop the execution of malicious functions (Petsas et al., 2014; Vidas & Christin, 2014).

Enck et al. (2010) performed dynamic taint analysis to track the flow of private and sensitive data through third party applications and detect any information leakage to remote servers. Yan & Yin (2012) employed two-level virtual machine introspection to rebuild operating system and Java level semantics of virtual Android devices. Thus, their work can be used to monitor the activities of the malware process based on its Java components and native components. Shabtai et al. (2012) developed a method for continuously monitoring various system events to detect suspicious activities by applying anomaly detection techniques. Karbalaie et al. (2012) inferred the system-call graphs from execution traces and applied frequent subgraph mining to find unique graphs that discriminate malware from benign application. Zhang et al. (2014) designed a dynamic analysis system called VetDroid to automatically construct permission use behaviors for Android applications

Finally, research by Guido et al. (2013) applied "traditional digital forensic techniques to remotely monitor and audit Android smartphones". In other words, the phone sends file system data to a remote server where it can be analyzed.

3.3. Detection using machine learning

To save practitioners resources in handcrafting and updating detection patterns, machine learning techniques have been widely applied for building classifiers to automatically detect malware (Sahs & Khan, 2012; Sanz et al., 2012a,b; Zhao et al., 2011). Aafer et al. (2013) utilized frequency analysis to capture the most relevant API calls that malware invoke, and used them as features to generate classifiers. Peiravian & Zhu (2013) extracted permissions from 130 possible system defined permissions and APIs from 1,326 possible system defined API calls, then used three classification methods to build a detection model. Arp et al. (2014) trained a relatively robust classification model based on SVM on computer and use as a security alarm service on android platforms. Aung & Zaw (2013) used permissions as features. Then they used the K-means cluster method to break 130 permissions into several clusters and selected one permission in each cluster as key features. They then reduced the amount of features, and used a decision tree to build a classification model.

Gascon et al. (2013) employed an explicit feature map inspired by the neighborhood hash graph kernel to represent applications based on their function call graphs. Zhou et al. (2012a) applied a fuzzy hashing technique to detect repackaged applications in third-party android market.

Lastly, Wang et al. (2014) evaluated the risk of collaborative permissions and used the risk value as a feature to build classification models.

4. Methodology

From a high-level perspective, we use machine learning techniques to classify applications into benign or malicious applications. Our procedure is divided into the following four steps:

1. **Decompile:** First, we prepare the APK file for processing which means we unpack it and decompile the bytecode. Both steps are executed using APKTool[3]. This step will not be discussed in any further detail as it is straightforward – the tool needs to be run on every APK file.

2. **Feature extraction:** After decompiling, the features / attributes are extracted from the smali code (decompiled bytecode). As mentioned in the introduction, our idea is to relate the APIs to the four application components (we also call this step 'API marking' or 'API linking'). All related details are outlined in Sec. 4.1.

[3] https://code.google.com/p/android-apktool/ (last accessed April 5th, 2016).

3. **Feature refinement:** After generating all features (marked APIs), we perform a refinement in order to identify the most relevant ones which helps improve the accuracy of the model and prevent overfitting. CAPIL uses two refinement steps in order to build a ranked list of features. The exact procedure is defined in Sec. 4.2

4. **Final model creation:** Once the features are refined, we have to find the best (final) and most accurate prediction model, i.e., how many features are needed for the strongest predictive model. Therefore, we utilize the J48 algorithm to create the classifier which is our final model. A brief description is provided in Sec. 4.3.

Note, CAPIL focuses on documented / Android framework APIs only, and does not consider third party APIs. In total, we identified 671 unique APIs based on previous research (more details are given in *Eliminate unused APIs (features)* in Sec. 4.2).

4.1. Feature extraction

Using API calls for classification is not a new technique. While traditional approaches analyze if a specific API is used or not, or rely on API sequences to build malware detection models, our goal is to **link the APIs to their application component(s)** and analyze the differences in usage. The devised procedure is as follows:

We mark each API according to the component(s) that uses it. In other words, we assign one or more of the four component (activity, service, provider or receiver) to each API call.

First, we assign one or more of the application components to each class[4]. This is accomplished by starting with those classes that are directly inherited from one of the main Android classes. Next, we follow their invocations to mark other classes. Each invoked class will have the same label. Note, a class has at least one label and may have up to four. Finally, all APIs in a class receive the label(s) of the class itself. If an API is called by two different classes, the API receives the label for both classes. An example is shown in Fig. 1.

Our example has two main classes `MainActivity` and `NetworkService`[5] which are directly inherited from the Android application components `activity` and `service`, respectively. As indicated by the arrow, `MainActivity` invokes `Login` which then invokes `Utilities`. Thus, both classes receive the `activity` label (blue bar). The second class, `NetworkService`, invokes `OpenVPNSettings` and `Utilities` which are both labeled as `service` (red bar). Note, since `Utilities` is used by two different classes, it receives two labels. Finally, all API calls inside this class receive the same label.

[4]When an application is decompiled by APKTools into smali code, the decompiler creates one file per class.

[5]Since no arrow is pointing to them, we know that these are main classes.

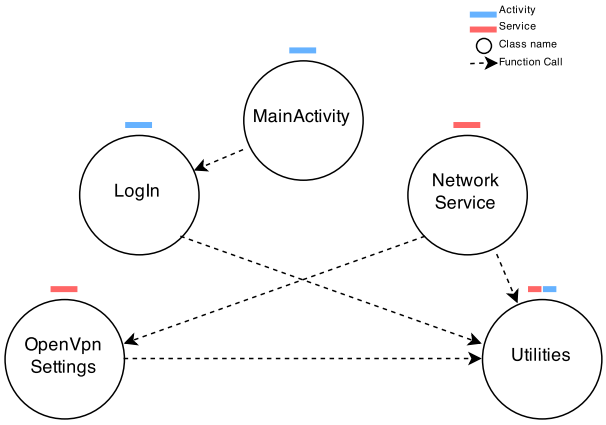

Figure 1: Example for labeling classes according to their application components.

4.2. Feature refinement

The purpose of feature refinement is to predict and evaluate the effectiveness / quality of the features for the classification model with the goal of identifying the minimum number of features. In other words, we attempt to select the most promising APIs.

CAPIL uses two steps for feature refinement starting with filtering out unused APIs. We then devised a new method to rank the API features and yield a classification model with a high accuracy rate. More details are given in the upcoming two subsections.

4.2.1. Eliminate unused APIs (features)

As aforementioned, we only use the **protected** Android framework APIs / documented APIs; third party APIs are ignored and filtered out. In this case **protected** means that we only consider APIs that require permissions as they tend to be more critical. In order to identify the documented APIs, we used previous work by Au et al. (2012) which showed that there are different APIs for different Android versions. We used their findings, combined all identified framework APIs and eliminated duplicates. This resulted in 671 unique APIs.

Note, naturally, API features are constantly changing due to Android system updates. It thus requires maintenance from time to time in order to adapt newly developed applications. In our case, the number of unique API will increase each time we add new documented APIs to it.

On the other hand, we would like to point out that some of the APIs are now deprecated but might still be found in test applications (especially when working with an older sample set). Therefore, CAPIL is able to consider those older APIs. When generating the decision tree model, we only focus on those APIs that are actually found in the sample applications and hence, we assume that this number (671 unique APIs) will drop. We denote this by $|API|$.

4.2.2. API (feature) refinement based on dissimilarity and sample coverage

In order to refine the APIs further and find the most appropriate ones, we build a new ranker mechanism, that allows us to predict the effectiveness of a single API / feature. In general, our approach aims to select those features that are widely but distinctly used by malware and benign applications. Therefore, we created two values to measure the degree of popularity (sample coverage) and distinctiveness (dissimilarity) which are explained in the following two paragraphs:

Sample coverage. As indicated by its name, sample coverage C describes how commonly an API is used throughout the set. That is, we count the number of samples a specific API is used within the set. Next, the value is normalized; we divide it by the total number of samples N:

$$C_{norm} = {}^{C}/_{N}$$

Dissimilarity. With the dissimilarity value D we want to identify those features that are the most different between benign and malicious applications. This requires the following two steps:

- **Step 1: Build summed vectors.** As explained in Sec. 4.1, we create one vector per API and per application that reflects the usage of the API in different components in the format `<activity, service, receivers, providers>`. In this step, we only accumulate all vectors that belong to the same API for benign and malicious, respectively, and receive two summed vectors for each API – a summed benign and a summed malicious vector. For instance, assuming that we have two benign applications and the API `ABC`, where in App_1 the API is marked as `<1,1,0,0>` and in App_2 as `<1,0,0,1>`. This means the combined benign vector for `ABC` is `<2,1,0,1>`.

- **Step 2: Calculate the dissimilarity value.** Next, we analyze the differences for a certain API and calculate the dissimilarity score D between the benign and malicious summed vectors. Therefore, we use the centered cosine similarity[6]:

$$D = \frac{\sum_{i=1}^{4}(B_i - \overline{B}) \times (M_i - \overline{M})}{\sqrt{\sum_{i=1}^{4}(B_i - \overline{B})^2} \cdot \sqrt{\sum_{i=1}^{4}(M_i - \overline{M})^2}}$$

  where 4 is the number of items in the vector, B_i is the i-th element in the benign vector and $\overline{B}$ is the mean value of the vector[7]. Correspondingly, M and $\overline{M}$ represent the malicious vectors and it's mean value. Since $-1 \leq D \leq 1$, we normalize it by

$$D_{norm} = \frac{(D+1)}{2} .$$

  Now D is in the range from 0 to 1 where 1 means two vectors hold the highest dissimilarity.

Combining the values. Here, we explain how to combine the two values C_{norm} and D_{norm} in order to generate the evaluation score E. The calculation of E follows the standard procedure for weighting values but set $W_1 = W_2 = 1$ [8]:

$$E = W_1 \cdot C_{norm} + W_2 \cdot D_{norm} = C_{norm} + D_{norm} ,$$

where $0 \leq E \leq 2$. The evaluation score is used to rank all the features. The higher the score, the more important a feature is for the classification model.

4.3. Final model creation

Before focusing on the model creation, we have to convert the feature vectors into a single value since classification algorithms (like J48) cannot handle vectors as features (attributes). In other words, our vector in the format `<activity, service, receivers, providers>` is converted into a single distinguishable numeric value ranging from 0 to 15 according to a binary table, i.e., `<0,0,0,1> = 1`, `<0,0,1,0> = 2`, `<0,0,1,1> = 3`, ... and `<1,1,1,1> = 15`. Accordingly, each feature becomes a number between 0 and 15.

In the final step, we have to identify the minimum amount of features that are necessary to build the best classifier. Precisely, we want to maximize the accuracy AC of the J48 decision tree using the least amount of features. The process for selecting the best classifier is the following:

$$x_{max} = \max_{x \in X}(\text{J48}(x)) \quad \text{where} \quad X = \{10, 20, 30, \ldots, |API|\}$$

where x represents the collection of top X features and the function J48(x) returns the accuracy AC of the classification model for the corresponding set. The outcome is x_{max} which is the number of needed features obtaining the best accuracy.

5. Experimental results

Our experimental results are mainly based on WEKA (Waikato Environment for Knowledge Analysis). WEKA is a popular suite of machine learning software written in Java and developed at the University of Waikato, New Zealand. It

[6] `http://en.wikipedia.org/wiki/Cosine_similarity` (last accessed April 5th, 2016).

[7] For instance, the mean value $\overline{B}ofavector$`<2,1,0,1>` = (2 + 1 + 0 + 1)/4 = 1. The resulting vector after the subtraction is `<2-1,1-1,0-1,1-1 =<1,0,-1,0>`.

[8] This needs further research. There might be weightings that obtain a better accuracy.

supports several standard machine learning tasks, more specifically, data preprocessing, clustering, classification, regression, visualization, and feature selection. We utilized their API and implemented python scripts for feature selection, classification, model building and evaluation.

We first describe our data set in Sec. 5.1. In Sec. 5.2 we analyze the feature refinement and discuss the final model. The overall performance is shown in Sec. 5.3. In Sec. 5.4 and Sec. 5.5 we compare CAPIL to other approaches and other refinement techniques, respectively, followed by a demonstration that shows how to combine refinement approaches in Sec. 5.6. The last section briefly discusses the runtime efficiency of our approach.

5.1. Data set

In our work, we utilized 12,958 sample applications which can be divided into 11,700 benign applications that were downloaded from the Google Play store (these were popular applications and we assume they contain no malicious code) and 1,258 malware samples which were acquired from the Android Malware Genome Project[9] (Zhou & Jiang, 2012). Thus, we have an imbalanced set with a ratio of $\approx 1/10$.

This total set is split into two subsets in the ratio $2/3$ (training) to $1/3$ (testing):

- a *training set* containing 8,638 samples (7,800 benign and 838 malware) and
- a *testing set* containing 4,320 samples (3,900 benign and 420 malware).

For clarity: the training set is used to train and identify the best model. This is performed using a 10-fold approach (details see next section). The actual evaluation is executed with the testing set. Thus, we never train and evaluate with the same samples. Subsets are always divided randomly.

Imbalanced set. We are aware that we utilize an imbalanced set by a factor of 1 to 10 towards benign samples. The domination of benign samples affects the classifier to adjust itself to detect more benign samples in order to achieve an optimized overall accuracy (this can be seen in our results).

Possible solutions are to decrease the number of benign samples in the training set or adjust a larger punishment on the classifier when it makes a wrong detection on the minority class (malware samples) (He & Garcia, 2009). However, we decided to have this imbalanced set for two reasons. First, our malware set is rather small and thus we wanted to use all of the samples. Second, this reflects real world circumstances – there are significantly more benign than malicious applications.

Finally, we would like to point out that many research groups do not publish their data sets making it difficult to gain access to large malware data sets.

[9] `http://www.malgenomeproject.org` (last accessed April 5th, 2016).

5.2. CAPIL-Feature refinement and model creation

In order to identify the best model, we have to extract and refine features and then test for the most accurate model (what is the best amount of quality features) which actually requires a third set. We solved this by using the training set in a 10-fold manner[10]. That is, the training set is split randomly into 10 equal-size subsets: 9 subsets are used for extraction and validation while the final subset is used for the model evaluation. This procedure is repeated 10 times where the subsets change. In the following we share the average results:

1. Our feature refinement method eliminates unused APIs which reduced the features from 671 to 373. Note, this is a result of some rarely used protected APIs in applications. For instance, the `MockApplication` is a collection of APIs dedicated to help developers test their application during the development stage – they almost never appear in published applications. Another example are `Inputmethodservice` APIs where even the developer documentnation says "not for use by normal applications"[11].

2. Next, we rank these remaining 373 features in descending order using the evaluation score E discussed in Sec. 4.2.

3. Finally, we create the model using the top 10, top 20, top 30, ... features and validate the model with the last subset.

Our final result is shown in Fig. 2. As indicated by the graph, the top 10 features already yield an accuracy of 96.40 %. The highest accuracy is 98.70 % which occurred when using the top 90 APIs. Note, this is the accuracy on the training set for the refinement only. The overall performance on the test set is assessed in the next section.

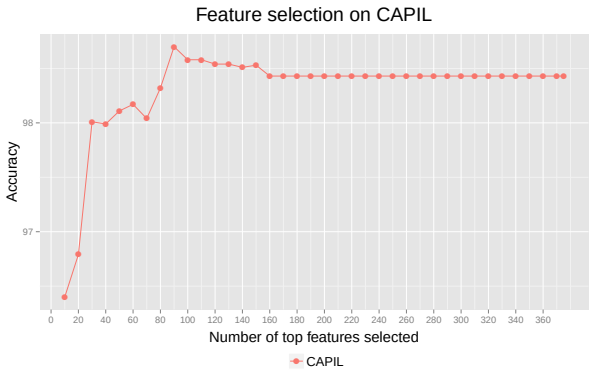

Figure 2: How a different number of features impact the accuracy.

[10] `http://en.wikipedia.org/wiki/Cross-validation_(statistics)` (last accessed April 5th, 2016).

[11] `http://developer.android.com/reference/android/inputmethodservice/package-summary.html` (last accessed April 5th, 2016).

5.3. Overall performance

Knowing the best model, this section shows the overall performance of CAPIL. Let TP (true positives) be the number of malware applications correctly identified, TN (true negatives) be the number of benign applications correctly identified, FP (false positives) be the number of benign applications identified as malware, and FN (false negatives) be the number of malware applications classified as benign, then we use the following three commonly adopted conventions[12]:

True positive rate (TPR)

$$TPR = \frac{TP}{TP + FN}$$

True negative rate (TNR)

$$TNR = \frac{TN}{TN + FP}$$

Accuracy (AC)

$$AC = \frac{TP + TN}{TP + TN + FP + FN}$$

That means, for the overall performance we measure the model's accuracy AC, i.e., the total number of benign and malware instances correctly classified divided by the total number of the data set instances.

A ROC (Receiver Operating Characteristic curve[13]) correlates the true positive rate (TPR, Y-axis) against the false positive rate (FPR, X-axis) at various thresholds. Unlike the three common criteria which examine the performance of a classifier with a chosen threshold, the Area Under the ROC curve (AUROC) quantifies the overall ability of the classifier separating the two classes across all possible thresholds. The higher the AUROC the smaller the area of overlapped distribution of the two classes, thus the better performance of the classifier. The results for CAPIL are shown in Table 1.

Table 1: CAPIL overall performance summary.

AC	TPR	TNR	AUROC
98.84 %	92.40 %	99.60 %	96.98 %

As can be seen, the malware detection rate (TPR) is 92.40% which is much lower than benign application detection rate which is 99.60%. The difference is clearly due to the imbalanced training set.

One should keep in mind that a balanced set reduces the TNR but increases the TPR (should approximately be identical). We would like to point out (depending on the scenario) that in a real world scenario and dealing with millions of applications, we might target a high true negative rate (TNR) as a low benign detection rate increases the number of malware warnings. For instance, let us assume there are 5 malicious and 100 benign applications. If the TPR (ability to detect malware) is very high (100%) but TNR (ability to detect benign apps) is very low (80%), there will be 20 false alarms. Upscaling this to real world numbers will be even worse.

5.4. Comparison with other existing approaches

It is quite difficult to compare CAPIL with existing approaches since we do not possess the test sets nor the implementations from prior work. However, this section exemplifies a more general comparison. The idea is to compare CAPIL with two other common approaches:

- The permission based approach takes commonly used permission (see Sec. 2.2.2) as features. In this experiment, we select all the documented permissions which is a total of 151[14].
- The API based approach uses possibly invoked APIs as features. As the amount of features can be quite large, a refinement is required in order to reduce the model's complexity and prevent it from over-fitting. The most common procedure is information gain.

The selected features are then fed into three different and wide spread classification algorithms: J48 (Decision Tree), Nearest Neighbor and Decision Table. The result is shown in Fig. 3 and outlines the models' accuracy as well as the AUROC.

As indicated by the graphs, it is a head-to-head race between CAPIL and the API based approach. The permission based approach lags behind, especially using the J48 decision tree, our approach performs very well.

Permission-based detection focuses on searching for certain combinations of critical permissions (Sanz et al., 2012b; Aung & Zaw, 2013). Some risk scoring method can also be used to give user warnings (Peng et al., 2012). However, researchers have shown that over 40% of the applications 'over declare' their permissions, that is, permissions are declared but no action is performed in the code (Pearce et al., 2012; Zhou et al., 2012b). Besides, some malware performs malicious behavior without the use of permissions making permission-based approaches fruitless in those instances (Grace et al., 2012b).

5.5. Comparison with existing feature refinement methods

Underpinning our approach is a novel method for feature refinement. In this section we compare our method of feature refinement with two existing and well-known approaches. The results are shown in Fig. 4 and will be discussed in the two subsections.

[12] https://en.wikipedia.org/wiki/Precision_and_recall (last accessed April 5th, 2016).

[13] https://en.wikipedia.org/wiki/Receiver_operating_characteristic (last accessed April 5th, 2016).

[14] http://developer.android.com/reference/android/Manifest.permission.html (last accessed April 5th, 2016).

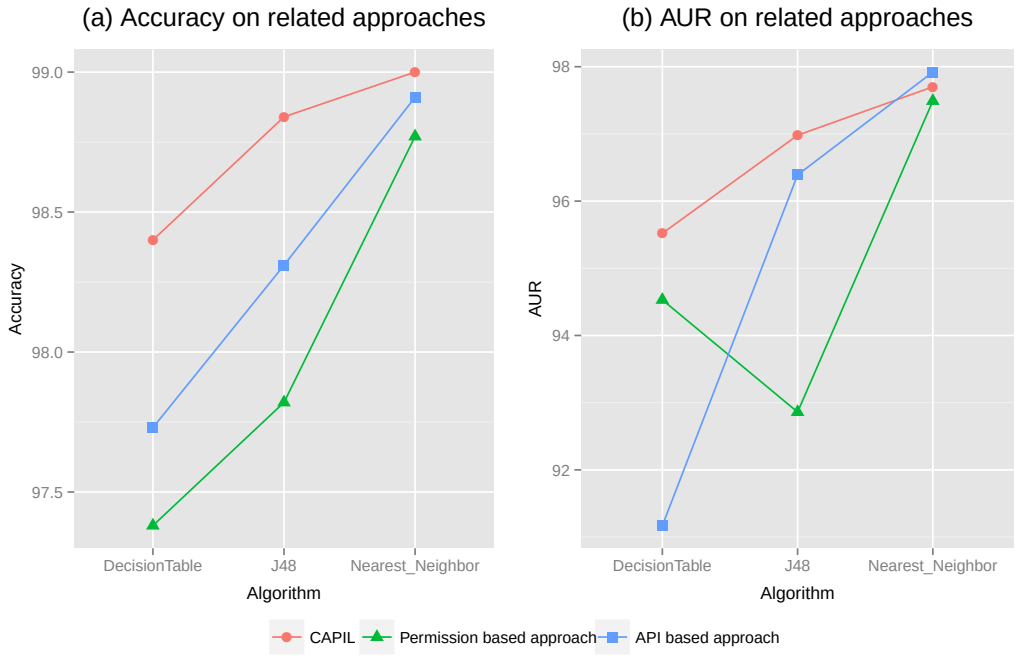

Figure 3: Detection rates for different approaches where (a) shows the accuracy and (b) AUR of the approaches.

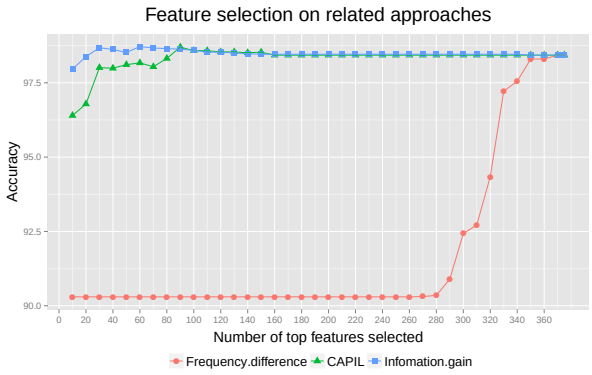

Figure 4: Comparison of different feature refinement algorithms.

5.5.1. *Comparison with frequency differences*

A commonly used technique to refine features is ranking them by their frequency differences (Aafer et al., 2013). As indicated by the results, our method is much more effective for feature selection: while the traditional approach needs around 370 features to obtain an accuracy of 98.51%, CAPIL can yield a better accuracy with 90 features selected.

5.5.2. *Comparison with information gain*

Information gain is one of the most powerful methods for feature refinement and is well documented and studied. As shown in Fig. 4, both algorithms refine the features well with a slightly advantage for information gain which yielded a 98.72% accuracy rate on 60 selected features while our approach needed approximately 90.

5.6. *Combining CAPIL and information gain.*

In this experiment we show that combining both methods can improve the final refinement result. Fig. 5 demonstrates that information gain and CAPIL need 60 and 90 features to obtain the best accuracy, respectively, while combing these approaches (running information gain followed by CAPIL) decreases the number of features to 50 with a higher accuracy of 98.76%.

This combination of the approaches needs further study. It is eye-catching that there is an intersection between the top two curves while we were expecting 'combined' to be constantly superior.

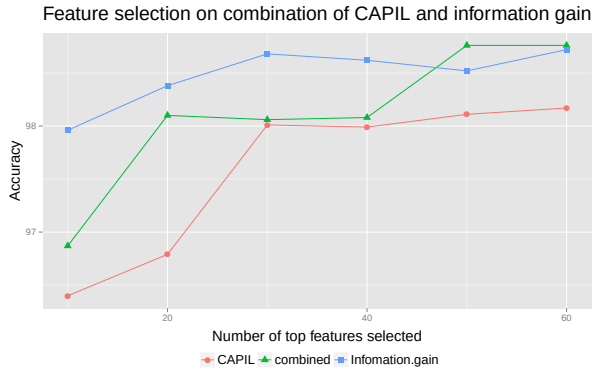

Figure 5: Combining information gain and CAPIL's feature refinement.

5.7. *Run-time efficiency*

The runtime of CAPIL is based on two aspects. First, finding the actual APIs in the decompiled code which encompasses

parsing the file line-by-line. Second, the marking / linkage procedure. Given that these two aspects may increase or decrease CAPIL's runtime, we decided to provide an average summary.

To evaluate the run-time efficiency, we randomly selected 1000 Android application samples and processed them with a desktop computer (2.6 GHz Intel Core i7 with 8 GB 1600 MHz DDR3). The average size (we only consider smali code files) was 4.2 MB and the average execution time was 9.95 seconds per application (this includes decompiling which was 4.74 seconds on average). As depicted in Fig. 6 and indicated by the regression line (blue line), one can say the execution time increases (in seconds) as the source code size increases (in MBs).

However, the marking impacts the execution time as we can see from the outliers. For instance, there is one application that needs 174 s (upper right corner of the figure) which results from a large amount of classes (many classes but only view lines of code per class). Nonetheless,it is possible to parallelize our approach via multi-threading and therefore these issues can be mitigated in a large-scale implementation.

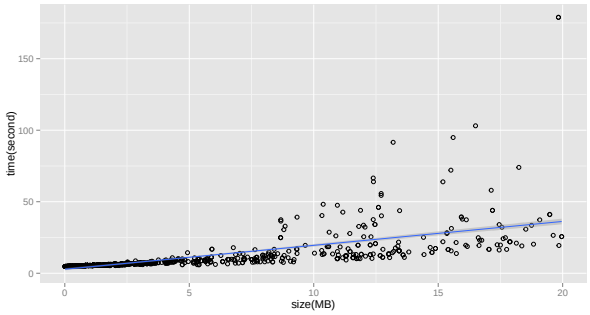

Figure 6: Run-time efficiency with respect to file size for 1000 samples.

6. Limitations

Currently there are several limitations for CAPIL. The most obvious limitation is its dependency to APKTool; if it is not able to decompile the application, CAPIL will not be successful. A reason for APKTool's failure may be due to 'code protection techniques' like code obfuscation or encryption. Furthermore, developers have the possibility to outsource code (including API calls) into library files named shared objects or `SO`-files. These files are mostly written in C/C++ and thus are harder to decompile (and APKTool does not support this).

We are also aware of the limitation given by the rather small and imbalanced data set which was discussed in Sec. 5.1. Again, we posit that these malware data sets are difficult to acquire as many researchers do not share their resources. However, we are constantly increasing our samples in an attempt to devise more comprehensive and balanced tests in the future. We will also examine possibilities for dealing with imbalanced sets.

7. Conclusion

In this work we described a technique for feature extraction as well as for feature refinement which we combined in our prototype implementation called CAPIL. The overall idea is to link APIs to their components for Android malware identification. To the best of our knowledge, this is a novel approach.

The component-API linkage is reasonable as it considers the intention behind a specific API usage. On the other hand, our method of feature refinement shows promising results considering the fact that it is new – the refinement is almost as good as information gain (when tested on our sample Android applications). The fact that our refinement method allowed us to improve the information gain results (reducing the amount of features from 60 to 50) is a clear indication that it is a reasonable approach. In total, CAPIL obtains an accuracy of 98.70 % requiring only 90 features within our data set of 12,958 applications.

We note that a problem we faced is the comparison of CAPIL to existing approaches since Android (application/malware) data sets are barely available. Therefore, we will publish all of our data to counteract this issue.

Before addressing the issues mentioned in the limitation section, we want to analyze the 'evaluation score' further. Currently this ranking function is based on equal weights, however, changing the weights may impact the final accuracy. Furthermore, future work should explore all potential APIs (not only Android framework APIs) and discern if this has any impact on the detection rates and the number of features.

8. References

Aafer, Y., Du, W., & Yin, H. (2013). Droidapiminer: Mining api-level features for robust malware detection in android. In *Security and Privacy in Communication Networks* (pp. 86–103). Springer.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., & Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket. *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), San Diego, CA*, .

Au, K. W. Y., Zhou, Y. F., Huang, Z., & Lie, D. (2012). Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* CCS '12 (pp. 217–228). New York, NY, USA: ACM.

Aung, Z., & Zaw, W. (2013). Permission-based android malware detection. International Journal Of Scientific and Technology Research Volume 2, Issue 3.

Bläsing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S., & Albayrak, S. (2010). An android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on* (pp. 55–62).

Chin, E., Felt, A. P., Greenwood, K., & Wagner, D. (2011). Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (pp. 239–252). ACM.

Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2008). A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, *44*, 6:1–6:42.

Elish, K. O., Yao, D. D., & Ryder, B. G. (2015). On the need of precise inter-app icc classification for detecting android malware collusions. *Proceedings of IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, .

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., & Sheth, A. N. (2010). Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones. In *Proceedings of the 9th USENIX*

Conference on Operating Systems Design and Implementation OSDI'10 (pp. 1–6). Berkeley, CA, USA: USENIX Association.

Enck, W., Octeau, D., McDaniel, P., & Chaudhuri, S. (2011). A study of android application security. In *USENIX security symposium* (p. 2). volume 2.

Enck, W., Ongtang, M., & McDaniel, P. (2009a). On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* CCS '09 (pp. 235–245). New York, NY, USA: ACM.

Enck, W., Ongtang, M., & McDaniel, P. (2009b). Understanding android security. *IEEE Security and Privacy*, *7*, 50–57.

Felt, A. P., Chin, E., Hanna, S., Song, D., & Wagner, D. (2011). Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* CCS '11 (pp. 627–638). New York, NY, USA: ACM.

Gascon, H., Yamaguchi, F., Arp, D., & Rieck, K. (2013). Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security* AISec '13 (pp. 45–54). New York, NY, USA: ACM.

Grace, M., Zhou, Y., Zhang, Q., Zou, S., & Jiang, X. (2012a). Riskranker: Scalable and accurate zero-day android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* MobiSys '12 (pp. 281–294). New York, NY, USA: ACM.

Grace, M. C., Zhou, Y., Wang, Z., & Jiang, X. (2012b). Systematic detection of capability leaks in stock android smartphones. In *NDSS*.

Guido, M., Ondricek, J., Grover, J., Wilburn, D., Nguyen, T., & Hunt, A. (2013). Automated identification of installed malicious android applications. *Digital Investigation*, *10, Supplement*, S96 – S104. The Proceedings of the Thirteenth Annual {DFRWS} Conference13th Annual Digital Forensics Research Conference.

He, H., & Garcia, E. (2009). Learning from imbalanced data. *Knowledge and Data Engineering, IEEE Transactions on*, *21*, 1263–1284.

Karbalaie, F., Sami, A., & Ahmadi, M. (2012). Semantic malware detection by deploying graph mining. *Int. J. of Computer Science Issues (IJCSI 2012)*, *9*, 373–379.

Pearce, P., Felt, A. P., Nunez, G., & Wagner, D. (2012). Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* ASIACCS '12 (pp. 71–72). New York, NY, USA: ACM.

Peiravian, N., & Zhu, X. (2013). Machine learning for android malware detection using permission and api calls. In *Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence* ICTAI '13 (pp. 300–305). Washington, DC, USA: IEEE Computer Society.

Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., & Molloy, I. (2012). Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* CCS '12 (pp. 241–252). New York, NY, USA: ACM.

Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., & Ioannidis, S. (2014). Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security* EuroSec '14 (pp. 5:1–5:6). New York, NY, USA: ACM.

Portokalidis, G., Homburg, P., Anagnostakis, K., & Bos, H. (2010). Paranoid android: Versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference* ACSAC '10 (pp. 347–356). New York, NY, USA: ACM.

Sahs, J., & Khan, L. (2012). A machine learning approach to android malware detection. In *Intelligence and Security Informatics Conference (EISIC), 2012 European* (pp. 141–147).

Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., & Bringas, P. (2012a). On the automatic categorisation of android applications. In *Consumer Communications and Networking Conference (CCNC), 2012 IEEE* (pp. 149–153).

Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P. G., & Alvarez, G. (2012b). Puma: Permission usage to detect malware in android. In *CISIS/ICEUTE/SOCO Special Sessions'12* (pp. 289–298).

Schmidt, A.-D., Clausen, J. H., Camtepe, A., & Albayrak, S. (2009). Detecting symbian os malware through static function call analysis. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on* (pp. 15–22). IEEE.

Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., & Weiss, Y. (2012). "andromaly": A behavioral malware detection framework for android devices. *J. Intell. Inf. Syst.*, *38*, 161–190.

Vidas, T., & Christin, N. (2014). Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* ASIA CCS '14 (pp. 447–458). New York, NY, USA: ACM.

Wang, W., Wang, X., Feng, D., Liu, J., Han, Z., & Zhang, X. (2014). Exploring permission-induced risk in android applications for malicious application detection. *Information Forensics and Security, IEEE Transactions on*, *9*, 1869–1882.

Yan, L. K., & Yin, H. (2012). Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (pp. 569–584). Bellevue, WA: USENIX.

Zhang, Y., Yang, M., Yang, Z., Gu, G., Ning, P., & Zang, B. (2014). Permission use analysis for vetting undesirable behaviors in android apps. *Information Forensics and Security, IEEE Transactions on*, *9*, 1828–1842.

Zhao, M., Ge, F., Zhang, T., & Yuan, Z. (2011). Antimaldroid: An efficient svm-based malware detection framework for android. In *ICICA (1)'11* (pp. 158–166).

Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., & Zou, W. (2012). Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices* (pp. 93–104). ACM.

Zhou, W., Zhou, Y., Jiang, X., & Ning, P. (2012a). Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy* CODASPY '12 (pp. 317–326). New York, NY, USA: ACM.

Zhou, Y., & Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (pp. 95–109).

Zhou, Y., Wang, Z., Zhou, W., & Jiang, X. (2012b). Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*.