# ANDRANA: Quick and Accurate Malware Detection for Android

Andrew Bedford[1], Sébastien Garvin[1], Josée Desharnais[1], Nadia Tawbi[1], Hana Ajakan[1], Frédéric Audet[2], and Bernard Lebel[2]

[1] Laval University
[2] Thales Research & Technology Canada

**Abstract.** In order to protect Android users and their information, we have developed a lightweight malware detection tool for Android called ANDRANA. It leverages machine learning techniques and static analysis to determine, with an accuracy of 94.90%, if an application is malicious. Its analysis can be performed directly on a mobile device in less than a second and using only 12 MB of memory.

**Keywords:** malware detection, android, static analysis, machine learning

## 1 Introduction

Android's domination of the mobile operating system market [30] has attracted the attention of malware authors and researchers alike. In addition to its large user base, what makes Android attractive to malware authors is that, contrarily to iOS users, Android users can install applications from a wide variety of sources such as first and third-party application markets (e.g., Google Play Store, Samsung Apps), torrents and direct downloads. Malware on mobile devices can be damaging due to the large amounts of sensitive information that they contain (e.g., emails, photos, banking information, location).

In order to protect users and their information, researchers have begun to develop malware detection tools specifically for Android. Traditional approaches, such as the signature-based and heuristics-based detection of antiviruses can only detect previously known attacks and hence suffer from a low detection rate. One possible solution is to use Machine Learning algorithms to determine which combinations of features (i.e. characteristics and properties of an application) are typically present in malware. These algorithms learn to detect malware by analyzing datasets of applications known to be malicious or benign.

The features used in Machine Learning are typically dynamically detected by executing the application in a sandbox (an isolated environment where applications can be safely monitored) where events are simulated [7,20,21]. This approach has two major problems, the first being the time needed. Analyzing each malware takes between 10 and 15 minutes (depending on the number of events sent to the simulator). The infrastructure required to keep such a tool up-to-date needs to be of considerable size, as more than 60 000 applications are

added to Google's Play Store each month [4]. The second problem is that this approach cannot take into account all possible executions of the application, only those that happen in the time allocated. Furthermore, sophisticated malwares can exploit this fact by stopping their malicious behavior when they detect that the current execution is in a sandbox.

To address these issues, we built a new malware detection tool for Android called Andrana. It uses static analysis to detect features, and Machine Learning algorithms to determine if these features are sufficient to classify an application as a malware. Static analysis can be performed quickly and directly on a mobile device. This means that no sandbox and no external infrastructure is required. Also, because static analysis considers all possible executions, it can detect attempts to evade analysis by the application. Andrana analyzes applications in three steps. First, the application is disassembled to obtain its code. Then, using static analysis, the application's features are extracted. Finally, a classification algorithm decides from the set of present features if the application is malicious.

One of the most important obstacle to static analysis is *obfuscation*. A code is obfuscated to make it hard to understand and analyze while retaining its original semantics. Although obfuscation has its legitimate uses (e.g., protection of intellectual property), it is often used by malware authors in an attempt to hide the malicious behaviors of their applications. Andrana can identify a number of obfuscation techniques and takes advantage of this information to improve the precision of its analysis.

In summary, our contributions in this paper are:

- We introduce Andrana, a malware detection tool able to quickly and accurately determine if an application is malicious (Section 3).
- We present the set of features that Andrana uses to classify applications. It includes the obfuscation techniques used by the application (Section 4).
- We have trained and tested classifiers using different machine learning algorithms on a dataset of approximately 5 000 applications. Our best classifier has an accuracy of 94.90% and a false negative rate of 1.59%. (Section 5).
- We report on two of our experiments to improve the overall accuracy and usability of Andrana: (1) using string analysis tools to improve the detection rate of API calls, (2) executing Andrana on a mobile device (Section 6).

## 2 Android

Before presenting Andrana, we must first introduce a few Android-related concepts and terminology, namely, the components of Android applications, Android's permission system and the structure of application packages.

### 2.1 Components

Android applications are composed of four types of components:

- **Activities**: An activity is a single, focused task that the user can do (e.g., send an email, take a photo). Applications always have one main activity

(i.e., the one that is presented to the user when the application starts). An application can only do one activity at a time.
- **Services**: A service is an application component that can perform operations in the background (e.g., play music). Services that are started will continue to run in the background, even if the user switches to another application.
- **Intents**: An intent is a message that can be transmitted to another component or application. They are usually used to start an activity or a service.
- **Content Providers**: Content providers manage access to data. They provide a standard interface that allows data to be shared between processes. Android comes with built-in content providers that manage data such as images, videos and contacts.

### 2.2 Permissions

Android uses a permission system to restrict the operations that applications can perform. Android permissions are divided into two categories:

- **Normal**: Normal permissions are ones that cannot really harm the user, system or other applications (e.g., change the wallpaper) and are automatically granted by the system [2].
- **Dangerous**: Dangerous permissions are ones that involve the user's private information or that can affect the operation of other applications [24]. For example, the ability to access the user's contacts, internet or SMS are all considered to be dangerous permissions. These permissions have to be explicitly granted by the user.

### 2.3 Application Packages

Android applications are packaged into a single *.apk* file which contains:

- **Executable**: Android applications are written in Java and compiled to Java bytecode (*.class* files). The *.class* files are then translated to Dalvik bytecode and combined into a single Dalvik executable file named *classes.dex*.
- **Manifest**: Every Android application is accompanied by a manifest file, named *AndroidManifest.xml*, whose role is to specify the metadata of the application (e.g., title, package name, icon, minimal API version required) as well as its components and requested permissions.
- **Certificate**: Android applications must be signed with a certificate whose private key is known only to its developers. The purpose of this certificate is to identify (and distinguish) application authors.
- **Assets**: The assets used by the application (e.g., images, videos, libraries).
- **Resources**: They are additional content that the code uses such as user interface strings, layouts and animation instructions.

## 3 Overview of Andrana

Andrana analyzes applications in three steps: disassembly, feature extraction and classification (see Figure 1).

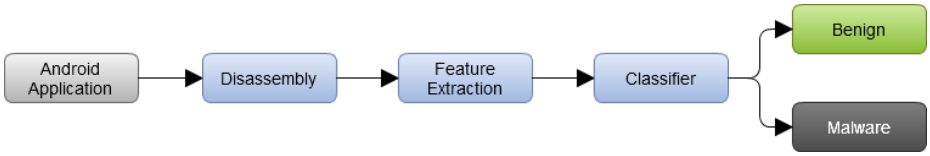


**Fig. 1.** General flow of ANDRANA

**Step 1: Disassembly** To analyze the code of the application and extract its features, we must first disassemble it. Fortunately, Android applications are based on Java, which is easy to disassemble and decompile. Moreover, Java forces multiple constraints on the structure of the code, which prevents manipulations that could make static analysis less effective (e.g., explicit pointer manipulations).

To disassemble the application, we use a tool called Apktool [3]. It converts Dalvik bytecode into Smali [29], a more readable form of the bytecode.

**Step 2: Feature Extraction** Once the application has been disassembled, its features are extracted using static analysis. These features, presented in Section 4, are characteristics and properties that the classifier will use in Step 3 to distinguish malicious from benign applications. It is the most computationally intensive step of the analysis.

**Step 3: Classification** Finally, the detected features are fed to a binary classifier that classifies the application as either "benign" or "malware". To generate the most accurate classifier possible, we have tried a variety of Machine Learning algorithms (see Section 5). They were trained and tested on a dataset of approximately 5 000 applications.

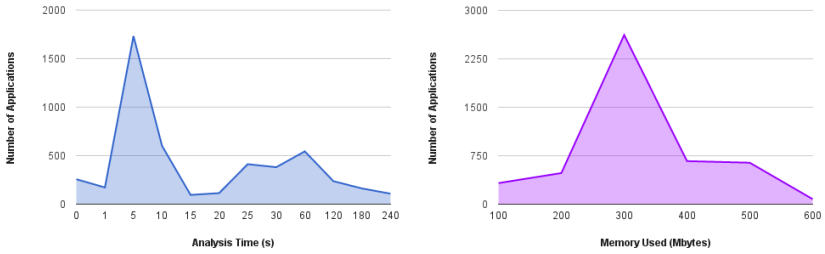


**Fig. 2.** Analysis time and memory usage distributions.

The whole process takes on average 30 seconds and 280 MB of memory (see Figure 2) on a desktop computer (Intel Core i5-4200U with 4 GB of RAM). We were able to produce an optimized Android version which utilizes a reduced set of significant features and whose analysis takes on average less than a second and 12 MB of memory (see Section 6).

## 4 Feature Extraction

In this section, we present the features extracted by ANDRANA. These features characterize the behavior of an application and are used by the classifier to determine if an application is malicious or benign. In addition to the features that are typically extracted in similar tools (see Section 7), such as requested permissions and API calls, ANDRANA also detects a number of obfuscation techniques and tools used by the application.

### 4.1 Features Extracted from the Manifest and Certificate

**Requested Permissions** We extract the permissions requested by the application from the manifest file. Certain combinations of requested permissions can be indicative of a malicious intent. For example, an application that requests permissions to access the microphone and start a service could be covertly listening in on conversations.

**Components** From the manifest file, we extract the application's components and determine if the application executes code in the background, which intents it listens to and which content providers it accesses.

**Invalid Certificate** We verify the validity of certificates using a utility called *jarsigner*. An invalid certificate indicates that the application has been tampered with.

### 4.2 Features Extracted from the Code

**API Calls** We extract API calls from the code and, when possible, we also extract the value of their parameters. The latter are useful, for example, when trying to detect the attempt to send an email. This is done by looking for the function call `Activity.startActivity("act=android.intent.action.sendto dat=mailto:")`. ANDRANA considers that this feature is present if there is a call to this function and a string containing the value `"act=android.intent.action.sendto dat=mailto:"` somewhere in the code.

**Necessary Permissions** By analyzing the API functions used in the code, we extract the permissions that are actually necessary to run the application. This allows to detect incongruities between the permissions requested by the application and those needed. Missing permissions could indicate that the application uses a root exploit to elevate its privileges during execution. To extract this information, ANDRANA uses an exhaustive mapping between the API calls and their required permissions. This mapping, which Google does not offer, is generated using PScout [25]. Note that since PScout's mapping is only an approximation, it may lead to false positives (i.e., reporting that there are missing permissions when, in fact, it is not true).

**Obfuscation Techniques Used** We identify the obfuscation techniques possibly used by the application. The common techniques are: renaming, reflection, encryption and dynamic loading [23]. Note that their presence does not necessarily mean that the application uses obfuscation, only that it *may* have. It is the role of the learning algorithm to consider this feature as important or not.

*Renaming* A simple way to obfuscate a code is to rename its packages, classes, methods and variables. For example, a class "Car" could be renamed "diZx9cA" or "ಟೀಪರೀಕ್ಷಾ" (Java supports unicode characters). This technique is particularly effective against human analysts as the purpose of a class or method has to be guessed from its content. It also makes it harder to recognize the method elsewhere in the code.

To detect the use of renaming, we exploit the fact that class names usually contain common names (e.g., File, Car, User) and methods contain verbs (e.g., getInstance, setColor). Knowing this, the first strategy of ANDRANA is to look for classes that have single-letter names (e.g., b.class). If there are many of them, then we assume that renaming has been used. If none are found, then we use an n-gram-based language detection library [18] to detect the language used to name the classes and functions of the application. If the result varies widely across the application, then we assume that renaming has been used.

*Reflection* Reflection refers to the ability of the code to inspect itself at runtime. It can be used to get information on available classes, methods, fields, etc. More importantly, it can also be used to instantiate objects, invoke methods and access fields at runtime, without knowing their names at compile time. For example, using reflection, an instance of `ConnectivityManager` is created and method `getActiveNetworkInfo` is invoked in Listing 1.1.

```
Class c = Class.forName("android.net.ConnectivityManager");
Object o = c.newInstance()

Method m = c.getDeclaredMethod("getActiveNetworkInfo", ...);
method.invoke(o, null);
```

**Listing 1.1.** Instantiating an object and calling a function using reflection

The use of reflection itself can be detected easily, by looking for standard reflection API calls.

*Encryption* Encryption can be used to obfuscate the strings of the code. For instance, it could be used to statically hide the names of classes instanced using reflection, as in the following listing.

```
String className = decrypt(encryptedClassName);
Class c = Class.forName(className);
```

**Listing 1.2.** Instanciating an object of a statically unknown class using reflection

To detect the possible use of encryption, ANDRANA looks for standard cryptography API calls.

*Dynamic Loading* Java's reflection API also allows developers to dynamically load code (.apk, .dex, .jar or .class files). This code can be hidden in encrypted/compressed assets or data arrays. However, to load this code, applications must use Android's API `getClassLoader` function. Once loaded, the reflection API must be used to access the classes, methods and fields of the dynamically loaded code. Android applications can also dynamically load native libraries through the *Java Native Interface* (JNI). This not only allows Java code to invoke native functions, but also native code to invoke Java functions. According to Zhou et al. [34], approximately 5% of Android applications invoke native code.

To detect the use of dynamic loading, ANDRANA looks for instances of classes `DexClassLoader` and `ClassLoader`. To detect the use of native libraries, we look for calls to the API `System.loadLibrary`. Note that, for the moment, ANDRANA only detects the use of dynamic loading and native libraries: the libraries are not analyzed.

**Commercial Obfuscation Tools** While developers can manually obfuscate the code themselves, most of them use commercially available obfuscation tools. ANDRANA is able to detect the use of these tools using the techniques described by Apvrille and Nigam [5]. The obfuscation tools that are currently detected are ProGuard, DexGuard and APKProtect.

- ProGuard renames packages, classes, methods and variables using either the alphabet (default behavior) or a dictionnary of words. ProGuard comes with the Android SDK and runs automatically when building an application in release mode. As such, it is the most popular obfuscation tool. ANDRANA can detect the use of ProGuard by looking for strings such as "a/a/a->a" in smali code.
- DexGuard is the commercial version of ProGuard. It also renames the packages, classes, methods and variables, but uses by default non-ASCII characters which reduces even more the readability of the code. It also encrypts the strings present in the code. ANDRANA detects the use of DexGuard by looking for names that contain non-ASCII characters.
- APKProtect can be detected by searching for the string `"apkrotect"` in the *.dex* file.

**Sandbox Detection** Certain malwares have the ability to deactivate their malicious behaviors when they detect that they are in a sandbox. This may indicate a malicious intent, as it could invalidate the results of a dynamic analysis. It does not affect static analyses, of course.

To detect the use of sandbox detection, we look for strings whose values are typically present in Android sandboxes. Vidas and Christin [31] enumerate some of the most common ones.

**Disassembly Failure** While disassembly works in most cases, it can sometimes fail. Disassembly failure clearly indicates an attempt to thwart analysis. For this reason, it is part of our feature set.

## 5 Classification and Evaluation

In this section, we evaluate the performance of multiple Machine Learning algorithms.

### 5.1 Dataset

To train and test our algorithms, we have collected and analyzed a dataset of approximately 5 000 applications, 80% of which were malwares. To avoid overfitting, the malware samples were randomly selected from two repositories: Contagio [12] and Virus Share [32]. The benign samples came from Google's Play Store various "Top 25". We noted that 47% of the samples used some kind of obfuscation.

Our dataset contains more malware samples than benign samples for two reasons. The main reason is that it is hard to obtain benign applications. Indeed, while there are many repositories of malicious Android applications, we found none that contained certified benign applications. Had we taken a larger number of applications from the Play Store, we would have risked introducing malicious samples into our dataset of benign samples. Another reason for using more malware samples is that it has a desirable side effect on the learning algorithm: it will lead the algorithm to try to make fewer bad classifications on this class. Hence, the number of false negatives (i.e., applications classified as "benign" when they are in fact malicious) will be naturally lower than the number of false positive.

### 5.2 Learning Algorithms

In order to obtain the best classifier possible, we have experimented with different learning algorithms: Support Vector Machines (SVM), k-Nearest Neighbors (KNN), Decision Trees (DT), Adaboost and Random Forest (RF).

***Support Vector Machines (SVM) [13]*** is a learning algorithm that finds a maximal margin hyperplane in the vector space induced by the examples. The SVM can also take into account a *kernel function*, which encodes a notion of similarity between examples. Instead of producing a linear classifier in the *input space*, the SVM can produce a linear classifier in the space induced by the chosen kernel function. In our experiments, we use the *Radial Basis Function* (*RBF*) kernel $k(x, x') = e^{-\gamma||x-x'||_2^2}$, where $\gamma$ is a parameter of the kernel function. The SVM also considers a hyperparameter $C$ that controls the trade-off between maximizing the margin and permitting misclassification of training examples.

***$k$-Nearest Neighbors (KNN) [14]*** is a learning algorithm that classifies a new data point by considering the $k$ most similar training examples and by choosing the most frequent label among these examples. Here, $k$ is a hyperparameter of the algorithm: different values of $k$ might give different results. The most similar examples are computed using any similarity function, such as the Euclidean distance.

***Decision Tree (DT) [9]*** is a learning algorithm that classifies examples by applying a decision rule at each internal node. The label of the example is decided at a leaf of the tree. Decision trees are learned by considering a measure of quality for a split such as the Gini impurity or the entropy for the information gain. In our experiment, we use the Gini impurity.

***Adaboost [27]*** is an *ensemble classifier*, that considers many *base classifiers* and learns a weighted combination of these classifiers. At each iteration, a new base classifier is chosen (or generated) to focus on examples that are incorrectly classified by the current weighted combination. The algorithm usually stops after a fixed number of iterations, or when the maximum number of base classifiers is attained. This maximum number of base classifiers is a hyperparameter of the algorithm.

***Random Forest (RF) [8]*** is, similarly to adaboost, an ensemble classifier. It builds a majority vote of decision tree classifiers, by considering sub-samples of the data and by controlling the correlation between the trees. The number of trees or tree construction parameters such as the maximal depth are hyperparameters of the algorithm.

### 5.3 Performance Metrics

To evaluate the performance of the resulting classifiers, we measured their True Positive Ratio (TPR). It represents the proportion of malware applications that are correctly classified:

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

where TP is the number of malware applications that are correctly classified and FN is the number of malware applications that are classified as "benign". Similarly, we measured their True Negative Ratio (TNR), which represents the proportion of benign application that are correctly classified:

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

where TN is the number of benign applications that are correctly classified and FP is the number of benign applications that are classified as "malware". Finally, we measured their overall accuracy, which represents the proportion of applications that are correctly classified:

$$\mathrm{Accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}.$$

### 5.4 Evaluation

According to Hoeffding's bound [16], with at least 600 test samples, the real risk is almost equal to the risk on test with 95% confidence. Hence, we chose to use

the following splitting scheme in our experiments: 2/3 ($\sim$ 3300 samples) for the training set and 1/3 ($\sim$ 1700 samples) for the testing set. For each algorithm, we chose the hyperparameters using a 5-folds cross-validation on the training set and chose the hyperparameter values that optimized the accuracy. Table 1 shows the hyperparameter values on which the cross-validation was performed for each algorithm. Finally, we trained the algorithm using the whole training set, and predicted the examples of the testing set. Note that all reported values are metrics calculated on the testing set, containing examples that have not been seen during training time. Table 2 shows the resulting accuracies for each algorithm.

| Learning Algorithm | Hyperparameter | Values |
|---|---|---|
| SVM | $C$ | $\{0.001, 0.01, 0.1, 1, 10, 100, 1000\}$ |
| | $\gamma$ | $\{100, 10, 1.0, 0.1, 0.01, 0.001, 0.0001\}$ |
| KNN | $k$ | $\{1, 2, 3, 4, 5, 10, 15, 20, 25, 50, 100\}$ |
| Decision Trees (DT) | $max\_leaf\_nodes$ | $\{5, 10, 15, 20, 25, 30, 40, 50\}$ |
| | $min\_samples\_leaf$ | $\{1, 2, 3, 5, 10, 20\}$ |
| AdaBoost | `n_estimators` | $\{5, 10, 25, 50, 100, 250, 500, 1000\}$ |
| RandomForest (RF) | `n_estimators` | $\{2, 5, 10, 25, 50, 100, 500, 1000, 2000, 3000\}$ |

**Table 1.** Considered values for each hyperparameter, for each algorithm.

| Learning Algorithm | Accuracy% | TPR% | TNR% |
|---|---|---|---|
| SVM | 94.72 | **98.64** | 78.43 |
| KNN | 94.11 | 97.74 | 79.06 |
| Decision Trees (DT) | 93.20 | 97.43 | 75.62 |
| AdaBoost | 94.11 | 98.26 | 76.87 |
| RandomForest (RF) | **94.90** | 98.41 | **80.31** |

**Table 2.** A comparison of the classifiers' metrics, Accuracy, True Positive Ratio and True Negative Ratio, using different machine learning algorithms.

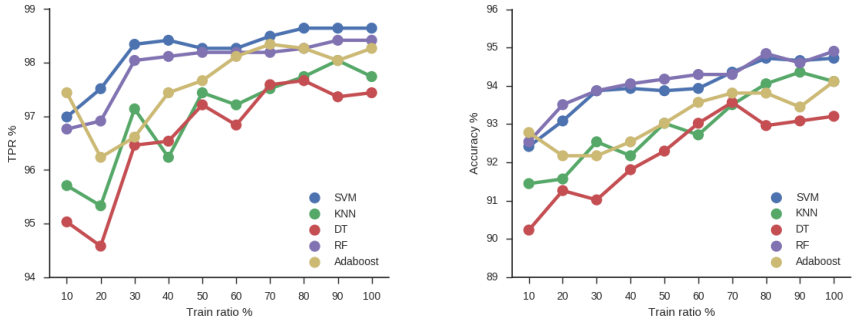


**Fig. 3.** The progression of true positives ratio and accuracy on the test set for different ratios of training set and for each learning algorithm. It is calculated using the best configuration of hyperparameters outputted by a 5-fold cross-validation.

We now discuss on whether an increase in the size of the training dataset can possibly improve the learning algorithms' performance. For this experiment, we first split the dataset into a training set (2/3) and a test set (1/3). Then, we followed the same procedure as above, but applied exclusively to a ratio of the training set and without altering the test set. Figure 3 shows that an increase of the training ratio leads to a fluctuating improvement of the accuracy. The non-monotonous behavior of the accuracy is a common occurrence in statistical learning and is mainly caused by noise in the dataset. Still, one can see that the accuracy tends to increase when the training ratio increases. So, we can expect a higher performance by using a larger dataset.

## 6 Additional Experiments

This section presents the various experiments that we did in order to improve the overall accuracy and usability of ANDRANA.

### 6.1 E1: Using String Analysis Tools

As previously mentionned, API calls can be invoked through reflection. To detect those calls, we look for their class and method names in the strings of the code. Of course, strings are not necessarily hard coded, they can also be dynamically built. For instance, in Figure 1.3, the class instantiated could be either `"java.lang.String"` or `"java.lang.Integer"`.

```
String a = "java.lang.";
String b;
if (random) { b = "String"; } else { b = "Integer"; }
String className = a + b;
Class c = Class.forName(className);
Object o = c.newInstance();
```

**Listing 1.3.** Dynamically built class name

To take into account cases where the class and/or function names are dynamically created, we have experimented with a tool called *Java String Analyzer* (JSA) [11,17]. JSA performs a static analysis of Java programs to predict the possible values of string variables. This allows us to determine that the possible values for the string variable `className` in Listing 1.3 are {`"java.lang.String"`, `"java.lang.Integer"`}.

However, JSA is not able to analyze entire Android applications in a reasonable time or without running out of memory. Li et al. [19] encountered similar problems with JSA and hypothesize that this problem is due to the fact that it uses a variable-pair-based method to do the global inter-procedural aliasing analysis. This method has an $O(n^2)$ memory complexity, where $n$ is the number of variables in the application.

We have also experimented with another string analysis tool called *Violist* [19]. While it is considerably faster than JSA and can actually be used to

analyze Android applications, it still requires too much time (around 4 minutes) and resources (up to 2.4 GB of memory) for our purpose: the analysis has to be executable on a mobile device. Furthermore, in our test on 10 applications that used reflection, it did not lead to the detection of additional features. For these reasons, we chose to not use them in ANDRANA. Besides, as seen in the previous section, it turns out that a precise string analysis is not required to accurately classify applications. This is because ANDRANA uses a wide variety of features to classify applications, some of which are not affected by obfuscation techniques (e.g., permissions, certificate, disassembly failure).

### 6.2 E2: Executing ANDRANA on a Mobile Device

Mobile devices generally have low computing power and memory compared to desktop computers. Consequently, if ANDRANA is to run directly on such devices, it must be very efficient. To evaluate ANDRANA's runtime performance on mobile devices, we have implemented a version of it for Android (see Figure 4). In order

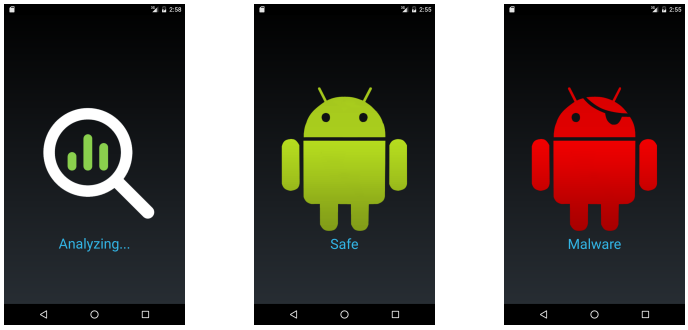


**Fig. 4.** ANDRANA's interface on Android.

to minimize ANDRANA's analysis time on Android, we chose to use the decision tree as the classifier. As previously shown, it is accurate (93.20%) and requires only a small subset of our features to classify applications (between 3 and 9 features). We also optimized ANDRANA's Android version so that it uses as little memory as possible. We analyzed 150 randomly selected applications from our dataset on a Nexus 5X and, on average, the analysis took only 814 milliseconds and 12 MB of memory, much quicker than our desktop version which extracts *all* features. Besides its performance, another advantage of using the decision tree classifier is that it is simple to understand and interpret.

## 7 Related Work

Research on malware detection tools for Android has been very active in recent years. We present in this section the approaches that are most similar to ours.

**Static Malware Detection Tools for Android** Sato et al. [26] present a method to calculate the malignancy score of an application based entirely on the information found in its manifest file. Namely, the permissions requested, intent filters (their action, category and priority), number of permissions defined and application name. They trained their classifier on a dataset of 365 samples and report an accuracy of 90%.

Aafer et al. [1] present a classifier, named DroidAPIMiner, that uses the API calls present in the code of the application to determine whether an application is benign or malicious. To determine the most relevant API calls for malware detection, they statically analyzed a large corpus of malware and looked at the most frequent API calls. They report a maximum accuracy of 99% using a KNN classifier.

Arp et al. [6] present another classifier, named Drebin, which uses statically detected features. Namely, they extract the hardware components (e.g., GPS, camera, microphone) used by the application by looking at the permissions requested in the manifest file, the requested permissions, the API calls present in the code, IP addresses and URLs found in the code. They use the SVM machine learning algorithm to produce a classifier. It has an accuracy of approximately 94%. Their Android implementation requires, on average, 10 seconds to return a result.

Since the datasets used in these approaches are not actually available for analysis, we cannot directly compare their performance with ANDRANA's. We also do not know if their samples were as heavily obfuscated as ours. All we can say is that ANDRANA seems to equal them in terms of accuracy and surpass them in terms of speed. We expect that by using a larger dataset of applications, like the 20 000 used by DroidAPIMiner, we could improve even more our accuracy. So that others may compare their results with ours, our dataset is available online [22].

There are also various antiviruses available on Google's Play Store (e.g., AVG, Norton, Avira). Antiviruses mostly use pattern matching algorithms to identify known malware (i.e., they look for specific sequences of instructions). This means that different patterns must be used to detect variations of the same malware. ANDRANA's main advantage over antiviruses is that it can not only detect known malware and their variations, but also unknown malware.

**Dynamic and Hybrid Malware Detection Tools for Android** Crowdroid [10], Andromaly [28] and MADAM [15] detect malware infections by looking for anomalous behavior. To detect anomalies, they monitor system metrics such as CPU consumption, number of running processes, number of packets sent through WiFi and/or the API calls performed at runtime by an application. Machine learning techniques are then used to distinguish standard behaviors from those of an infected device.

DroidRanger [34] use both static and dynamic analysis to perform a large-scale study of several application markets. Instead of using machine learning techniques to automatically learn to classify applications, they use a variety of

heuristics. Using their tool, they were able to identify 211 malicious applications present on the markets, 32 of which were on Google's Play Store.

DroidDolphin [33] inserts a monitor into applications to log their API calls and then executes them. The authors generate a classifier using this information and a dataset of 34 000 applications. They report an accuracy of 86.1%.

Andrubis [21] and its successor Marvin [20] uses approximately 500 000 features, detected using a combination of static and dynamic analyses, to train and test their classifier on a dataset of over 135 000 applications. They report an accuracy of 98.24%.

ANDRANA's main advantages over these approaches are that it introduces no runtime overhead and that its analysis can be performed on the user's mobile device, very quickly.

## 8 Conclusion

In this paper, we have presented ANDRANA, a lightweight malware detection tool for Android. It uses static analysis to extract an application's features and then uses a classifier to determine if it is benign or malicious. We have trained and tested multiple classifiers using a variety of Machine Learning algorithms and a dataset of approximately 5 000 applications, 4 000 of which were malwares. The dataset is available online [22]. Its samples came from multiple sources to avoid overfitting. Our best classifier has an accuracy of 94.90% and a false negative rate of 1.59%, which is comparable to other similar tools. As indicated by the upward trends of Figure 3, the use of larger datasets should lead to even higher accuracies.

As almost half of our dataset used reflection, we considered using two string analysis tools, JSA and Violist, to improve the detection rate of our features, but their use turned out to be too computationally expensive for our purpose.

We have implemented a version of ANDRANA for Android and our tests reveal that, on average, it can analyze applications in less than a second using only 12 MB of memory, faster and more efficiently than any similar tools. Since our implementation uses a decision tree as its classifier, users can easily understand what lead the application to be classified as malware/benign.

**Future Work** Benign applications may also compromise the security of a user's information, generally by accident. For this reason, we are working on a way to enforce information-flow policies by inlining a monitor in Android applications. In theory, this type of mechanism could allow users to execute applications safely, that is, without compromising the confidentiality of their information or the integrity of their system.

We are also working on our own string analysis tool. Our goal is to make it as lightweight as possible so that it can be executed on a mobile device.

**Acknowledgments** We would like to thank François Laviolette for his suggestions and Souad El Hatib for her help with the string analysis tools. This project was funded by Thales and the NSERC.

## References

1. Aafer, Y., Du, W., Yin, H.: Droidapiminer: Mining api-level features for robust malware detection in android. In: Sec. and Priv. in Comm. Networks, pp. 86–103. Springer (2013)
2. Android operating system security, http://developer.android.com/guide/topics/security/permissions.html, accessed July 5, 2016
3. Apktool, https://ibotpeaches.github.io/Apktool/, accessed July 5, 2016
4. Appbrain, http://www.appbrain.com/stats/number-of-android-apps, accessed July 18, 2016
5. Apvrille, A., Nigam, R.: Obfuscation in android malware, and how to fight back. Virus Bull pp. 1–10 (2014)
6. Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and explainable detection of android malware in your pocket. In: Proc. of the Annual Symposium on Network and Distributed System Security (NDSS) (2014)
7. Atzeni, A., Su, T., Baltatu, M., D'Alessandro, R., Pessiva, G.: How dangerous is your android app?: an evaluation methodology. In: Proceedings of the 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services. pp. 130–139. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering) (2014)
8. Breiman, L.: Random forests. Machine learning 45(1), 5–32 (2001)
9. Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A.: Classification and regression trees. CRC press (1984)
10. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. pp. 15–26. ACM (2011)
11. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. Springer (2003)
12. Contagio, http://contagiominidump.blogspot.ca/, accessed July 16, 2016
13. Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning 20(3), 273–297 (1995), http://dx.doi.org/10.1007/BF00994018
14. Cunningham, P., Delany, S.J.: k-nearest neighbour classifiers. Multiple Classifier Systems pp. 1–17 (2007)
15. Dini, G., Martinelli, F., Saracino, A., Sgandurra, D.: Madam: a multi-level anomaly detector for android malware. In: International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security. pp. 240–253. Springer (2012)
16. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American statistical association 58(301), 13–30 (1963)
17. Java string analyzer (jsa), http://www.brics.dk/JSA/, accessed July 5, 2016
18. Language detection library, https://github.com/shuyo/language-detection, accessed July 5, 2016
19. Li, D., Lyu, Y., Wan, M., Halfond, W.G.: String analysis for java and android applications. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 661–672. ACM (2015)

20. Lindorfer, M., Neugschwandtner, M., Platzer, C.: Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In: Computer Software and Applications Conference (COMPSAC), 39th Annual. vol. 2, pp. 422–433. IEEE (2015)
21. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Van Der Veen, V., Platzer, C.: Andrubis–1,000,000 apps later: A view on current android malware behaviors. In: 2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS). pp. 3–17. IEEE (2014)
22. Lsfm, http://lsfm.ift.ulaval.ca/recherche/andrana/, accessed September 30, 2016
23. Maiorca, D., Ariu, D., Corona, I., Aresu, M., Giacinto, G.: Stealth attacks: An extended insight into the obfuscation effects on android malware. Computers & Security 51, 16–31 (2015)
24. Permissions classified as dangerous, http://developer.android.com/guide/topics/security/permissions.html#normal-dangerous, accessed July 5, 2016
25. Pscout, https://github.com/dweinstein/pscout, accessed July 5, 2016
26. Sato, R., Chiba, D., Goto, S.: Detecting android malware by analyzing manifest files. Proc. of the Asia-Pacific Advanced Network 36, 23–31 (2013)
27. Schapire, R.E., Singer, Y.: Improved boosting using confidence-rated predictions. Machine Learning 37(3), 297–336 (1999)
28. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y.: "andromaly": a behavioral malware detection framework for android devices. Journal of Intelligent Information Systems 38(1), 161–190 (2012)
29. Smali/baksmali, https://github.com/JesusFreke/smali, accessed July 20, 2016
30. Smartphone os market share, q1 2015 (2015), http://www.idc.com/prodserv/smartphone-os-market-share.jsp, accessed July 7, 2016
31. Vidas, T., Christin, N.: Evading android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM symposium on Information, computer and communications security. pp. 447–458. ACM (2014)
32. Virus share, https://virusshare.com/, accessed July 14, 2016
33. Wu, W.C., Hung, S.H.: Droiddolphin: a dynamic android malware detection framework using big data and machine learning. In: Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems. pp. 247–252. ACM (2014)
34. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In: NDSS. vol. 25, pp. 50–52 (2012)