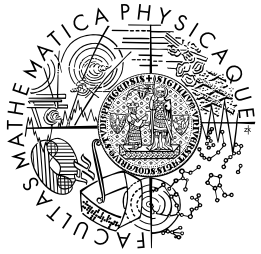

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

**BACHELOR THESIS**

Dominik Smrž

# Feature extraction from Android application packages and its usage in machine learning for malware classification

Computer Science Institute of Charles University

Supervisor of the bachelor thesis: Mgr. Martin Bálek
Study programme: Computer Science
Study branch: General Computer Science

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date .....  signature of the author

Title: Feature extraction from Android application packages and its usage in machine learning for malware classification

Author: Dominik Smrž

Institute: Computer Science Institute of Charles University

Supervisor: Mgr. Martin Bálek, Department of Applied Mathematics

Abstract: In this Thesis, we propose a machine-learning based classification algorithm of applications for a popular mobile phone operating system Android that can distinguish malicious samples from benign ones. Feature extraction for the machine learning is based on static analysis of the application's bytecode with focus on API and method calls. We show various ways to transform the most frequent API and method calls into numeric (histogram-based) features. We further examine the specifics of the extracted features and discuss their importance. The dataset used for experiments in this Thesis contains more than 200,000 samples with approximately half of them malicious and half of them benign. Further, multiple machine learning algorithms are examined and their performance is evaluated. The size of our dataset prevents overfitting and hence provides a reliable basis for training of classification models. The results of the experiments show that the proposed algorithm achieves very low false positive rate under 2.9% while preserving specificity over 93.6%.

Keywords: malware analysis, android, machine learning, feature engineering

I would like to express my gratitude to my supervisor, Mgr. Martin Bálek, for his guidance and insight. I also want to thank my family and friends for their support and encouragement.

# Contents

# Introduction

Technologies have been changing rapidly in last decades. Nowadays almost everybody has a mobile phone. Recently it is more and more common that these phones have highly customizable software capable of an enormous number of various actions. The computational performance of a regular phone has increased so much, and the gap between the old ones and the new ones have become so wide that people started calling modern mobile phones smartphones.

The rapid growth in the usage of smartphones has triggered manufacturers to create several different operating systems customised to fit such devices. There are a couple of notable examples such as Apple's iOS and Windows Phone[1] from Microsoft. The dominant operating system for mobile phones is Android developed by Google [1]. It also has a significant share among tablet users. Given the late growth in the usage of such devices, it has even recently become the most used operating system overall [2].

Smartphones are highly customizable (at least compared to old phones) via millions of application that virtually anybody can create or use. This means that installing new software into our phones has become more than common.

People use smartphones for virtually anything, starting from internet browsing over playing games to managing their personal data and banking accounts.

Its widespread usage combined with the new capability of installing custom software and easy access to various data regrettably attracted the attention of crackers and criminals who started to develop malicious applications for these operating systems. This gives us the motivation to study such applications in the hope of creating an algorithm for distinguishing malicious applications from benign ones.

As the capabilities of mobile phones have become greater, the malware designed for them has become more dangerous. The malicious behavior ranges from mildly annoying such as stealing contact for sending spam to potentially devastating for the user such as taking control over bank accounts.

Even though there are multiple operating systems for mobile phones, in this thesis we focus on the one with the largest market share – Android. Other ones have naturally entirely different architecture, and the algorithm designed specifically for a single OS can focus on its design and thus have better results.

Further on this topic, we shall note that antiviruses designed for usual desktops are not suitable for Android applications. The malicious files for desktops are documents, images or binaries that exploit weaknesses of desktop operating sys-

[1]Microsoft Windows 10 is a universal operating system for desktops, smartphones and tablets and no longer has "Phone" in its name

tems. On the other hand, the applications for Android system (so called Android Package Kit, APK for short) have their structure designed with bytecode that is far easier to reverse engineer in order to understand its functionality.

Malware for mobile phones is a pressing issue. On the brighter side, the APK files have a structure that can be studied and exploited for analysing the behavior of the application. This led us to our attempt to create a classifying algorithm for APK files. As we have access to over 200,000 samples, we train multiple machine learning algorithm for such purpose. We rely on static analysis (i.e. examining samples without executing the code) because we want to keep the analysis reasonably fast.

# 1. Related work

As we said, the popularity of Android attracts the attention of malware creators. This also means that the topic of characterization malicious APK files is also widely covered in the scientific literature.

For example in [3] the authors thoroughly discuss general characteristics of malware. They manually analyse 1,260 malware samples that were gathered over a year-long period of time. They focus on various aspects of malware, for example, they note that the majority of samples are a legitimate application with injected malicious payload. Given the nature of the APK files, the payload can be added to virtually any existing application without access to the original code. The authors also get a surprising result when they analyse the purpose of malicious applications, finding out that 93% of analysed applications allow the mobile phone to be remotely controlled.

Furthermore, in [4] the authors build machine learning algorithm for detecting malware. In this article, the authors try various approaches to feature extraction. Their best classifier relies on API calls with further refinement and uses k-Nearest Neighbours algorithm. The authors have created machine learning algorithm with accuracy 99% with the low false positive rate of 2.2%. These results were achieved on a dataset consisting of 3,987 malware applications and 16,000 most popular applications on Google Play.

In [5] the authors propose schemes for detecting known malware based on behavioral footprint matching. Since the generation of a behavioral footprint is time-consuming, they integrate pre-filtering based on used permissions into the algorithm. The authors also focus on zero-day malware detection and propose heuristic filtering combined with dynamic analysis to detect such malicious samples.

Finally, in [6] the authors build machine learning algorithm based on more advanced features such as permissions, control flow graphs and kernels. The authors proposed machine learning algorithm based on Support Vector Machines.

# 2. APK structure

Every application for Android system is distributed in a form of an APK file. Such files have a defined format and required components. In order to understand the behavior of APK files and provide a basis for static analysis, we discuss its structure in this chapter.

## 2.1 APK overview

The APK file is a regular ZIP archive with some specific files included. In this archive, we can find all the components which are required to run the application. The executable code for Android application needs to be written in Java [1] and we can find it in the compiled version as so called Dalvik bytecode inside the APK. Besides the bytecode, there are also other necessary resources.

Required files that need to be the content of any APK file are as follows:

- `META-INF` directory – this folder contains a certificate of the APK and related files. Android does not allow installation of the application without a proper certificate. Since any slight change in the content of any file inside the APK results in a change of a signature, nobody can modify the APK without having access to the private key of the original publisher unless they re-release the application with their own key.

  For a shortened example of a certificate see Figure 2.1. We can find additional information there such as the issuer of the certificate. To trust the information one still needs to verify that the signing key actually belongs to the mentioned issuer.

  This makes a certificate potentially helpful part of the APK file for malware classification. For example, if we had a list of trusted certificates we would be able to easily classify at least some APK files as clean without further analysis (it is also possible to have a list of leaked certificates and classify corresponding APKs as malicious). Unfortunately, we do not possess any such database; furthermore, this approach can be applied to a very small fraction of samples anyway.

- `classes.dex, classes2.dex, classes3.dex...` – these files hold entire bytecode (this means in the majority of cases the whole executable code). As they contain information about the actual behavior of the application,

[1] Note that Google recently introduced a new language for developing on Android – Kotlin. This does not pose any problem for us because it has no effect on generated APK

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            e5:ad:48:3c:01:17:93:ef
    Signature Algorithm: sha512WithRSAEncryption
        Issuer: C=CZ, ST=Prague, L=Prague, O=AVAST Software a.s., CN=
            avast! Android/emailAddress=android@avast.com
        Validity
            Not Before: Nov 29 12:46:17 2011 GMT
            Not After : Dec 23 12:46:17 2052 GMT
        Subject: C=CZ, ST=Prague, L=Prague, O=AVAST Software a.s., CN=
            avast! Android/emailAddress=android@avast.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:de:95:cf:00:31:7d:52:05:1d:38:d9:ea:37:69:
                    c5:b2:43:92:cc:27:40:a6:5d:89:8f:d4:fd:5a:7e:
                    ...
                    f4:45:0d
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                91:E0:C1:65:CD:A7:F8:01:7F:37:24:92:5C:D9:D0:39:26:DA:09:
                    F0
            X509v3 Authority Key Identifier:
                keyid:91:E0:C1:65:CD:A7:F8:01:7F:37:24:92:5C:D9:D0:39:26:
                    DA:09:F0

            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: sha512WithRSAEncryption
         33:45:2e:8e:64:ea:33:d7:39:94:74:05:41:3d:35:79:92:d9:
         70:e6:75:7b:c5:fb:c0:c0:56:a1:41:2c:2d:c3:88:10:3b:e4:
         ...
         3e:70:28:d6:56:e8:5f:f4
```

Figure 2.1: Example of certificate

we take them as main input for our research. We discuss the files further in Section 2.2.

- `AndroidManifest.xml` – this file is in the form of binary XML (e.g. the file still has the structure of the XML file but with different formatting). A lot of interesting information can be found here, such as the list of permissions the application asks for, the list of events that starts the execution of a part of the application code, and what methods handle such event.

- `lib, res, assets, resources.arsc...` – additional files and folders contained in an APK file. They hold other assets such as images, texts, binary libraries and so on. In general, these files and folders can have any content whatsoever. This means any suspicious code can be hidden in these files.

  Such wrongful behavior is extremely hard to detect. Initial code is still executed from `classes.dex` though. Thus we can still recognize if there is some extra code executed (there will be API calls responsible for loading data from external files). However, without further techniques (dynamic analysis, evaluating binary code) we cannot recognize whether this behavior is indeed malicious or benign and we will not cover such approach in this thesis. Overall these hard-to-detect ways of hiding malicious code are seen only in minority of cases.

## 2.2 DEX and Dalvik bytecode

The main source of information for our research is so called Dalvik bytecode that is contained inside `classes.dex` file, as this file includes the executable code of the application. Sometimes we can see that there are more such files called `classes2.dex`, `classes3.dex` and so on. There is a reason for that. Methods inside the DEX file is indexed by two-byte integer. Thus there is a hard limit on the number of methods in single DEX file (65,535 methods). Some of the newest applications find this limit too restricting. Android has a native solution to this problem. If there is a need for a high number of methods compiler splits the code into several DEX files. Overall this is not an issue for us. When parsing APK with multiple DEX files, we just need to parse each one of them individually and at the end concatenate their contents (e.g. a list of methods).

There is a lot of information inside the DEX files. The file itself starts with some header information including magic flag and consistency checksums (security checksums is not included anymore since this part is covered in `META-INF` directory of APK). Besides these checks, the header contains offsets to other parts of the DEX file, such as the string table, the list of methods and the list of classes.

The first core part of the DEX file is the string table, which has to contain every string the application uses including hardcoded strings that are used inside original Java code. Furthermore, the descriptor of classes (and even primitive types) are stored here. Thus if there is a usage of basic Java `Object` anywhere inside the DEX (which most likely is, because custom classes usually inherit directly from this class), there is the string `LJava/lang/Object;` inside the string table. This means we can see if the application uses any classes (e.g. `android/telephony/gsm/SmsManager`) without even reading the bytecode. The strings are encoded in MUTF-8 encoding, which is very similar to standard UTF-8. Further description can be found in [7].

Another important part of the DEX file is a list of classes. By simply reading this part we can obtain all information about every class. Definitions of all classes are stored here (besides those loaded dynamically and those hardcoded in Android core). We can find there the name of the class as well as the name of its superclass and the list of its methods (both virtual and direct). For every method, there is its name, the number of its parameters and their types, type of the return value, and its bytecode.

For describing a method we define string that uniquely identify given method.[2] The string has the format ⟨*class*⟩`->`⟨*methodName*⟩`(`⟨*paramTypes*⟩`)`⟨*returnType*⟩. The ⟨*class*⟩ is the description of the class that contain given method and consists of `L`⟨*fullClassName*⟩`;`. Such description contains full class name including namespaces and may look like this: `Lcom/example/MainActivity;`. This identifying string for classes is also mentioned in the documentation [7] and used for describing classes inside the DEX file (i.e. inside every class definition there is a reference to the corresponding string in the string table). The ⟨*returnType*⟩ is description of the return type. It can either be a class, in which case it would be described in the same way as ⟨*class*⟩, or one of the primitive types described in Table 2.1. Finally, the ⟨*paramTypes*⟩ is just concatenated description of types of parameters (if any). The description of each parameter is the same as in the case of return types with a small difference: the `V` for void can not be used and any type can be preceded by (several) "`[`" indicating the parameter is (multidimensional) array of given type. For example the description of a method can be: `Lcom/example/MainActivity;->setTheme(I)V`.

### 2.2.1 Dalvik bytecode

The Dalvik bytecode is very similar to the Java bytecode. The main difference is that all classes are encapsulated in a single file (in the case of multidex the classes are spread across several files, but still one file holds many classes). Besides

[2] The same unique strings are also used by smali format which we use further in the thesis.

| Descriptor | Explanation |
|---|---|
| `V` | void |
| `Z` | boolean |
| `B` | byte |
| `S` | short |
| `C` | char |
| `I` | int |
| `J` | long |
| `F` | float |
| `D` | double |

Table 2.1: List of primitive types and their descriptors

that, the Dalvik uses register-based architecture, and the instruction set has some minor differences. Overall the similarity is still very significant, and one of the common and well-functional methods of reverse engineering is to convert the DEX file into `.jar` file. Then we can use any techniques for analysing such files, such as decompile it with any Java decompiler and manually analyse the Java source code that is easier to understand compared to the bytecode.

The Dalvik bytecode uses various instructions. As in a case of any other bytecode, this includes basic instructions for elementary numeric operations such as adding, multiplying, computing xor and conditional jumps.

Besides that the Dalvik uses high-level instructions. For example the instruction for loading the string directly from the string table. Most importantly for us – there are the instructions for invoking other methods and storing their results. We mainly focus on this kind of instructions when extracting the features as they give us the ability to follow the flow of the code easily. Furthermore, since every function call including basic API calls generates this instruction, we can easily track usage of suspicious API calls.

This is very important since the majority of malicious acts is done via such API calls as this is the only way how to conduct actions such as sending premium SMS. For this reasons those API calls (and instruction that invokes other methods in general) is the main focus of our research.

As an example we can take a look at a function with a simple body:

```
public emailCheck(String email) {
    return email.contains("@");
}
```

The whole method translates into the following sequence of bytes:

```
001a 0215 206e 4e1c 0002 000a 000f
```

| Bytecode | Explanation |
|---|---|
| `0x001a` | `0x1a` is code for `const-string` instruction and `0x00` is index of register to store the string in. |
| `0x0215` | Index to string table to string "`@`". |
| `0x206e` | `0x6e` is code for `invoke-virtual` instruction, `0x2` is number of arguments and `0x0` is just space-filler |
| `0x4e1c` | index to method table that resolves to method `contains` of class `String` |
| `0x0002` | `0x0` and `0x2` are registers of arguments of methods and leading `0x00` are just space-fillers |
| `0x000a` | `0a` is code for `move-result` instruction (instruction that stores the result of previous call of method) and `0x00` is index of register to store the result to |
| `0x000f` | `0x0f` is code for `return` instruction and `0x00` is the register that holds the returning value |

Table 2.2: Detailed explanation of function that checks if there is "`@`" inside its first parameter

```
const-string v0, "@"
invoke-virtual { p1, v0 },
LJava/lang/String;->contains(LJava/lang/CharSequence;)Z
move-result v0
return v0
```

Figure 2.2: Function that checks presence of "`@`" inside its first parameter in so called smali format

Please note that the Dalvik bytecode has little-endian architecture and is parsed by 16-bit numbers. For this reason, the order of reading bytes may be counterintuitive. For the description of each word see Table 2.2 and for a whole bytecode in human readable form see Figure 2.2. This format called smali is frequently used when reversing the Dalvik bytecode. The format is quite self-explanatory: There is one instruction per line. Standard names are used for the opcodes. If there is a reference to the string table, the string itself is used instead, and if there is a reference to the method list the unique string of method as described in the previous section is used. We use this formatting in this thesis, as it is the standard way to write the Dalvik bytecode.

# 3. Dataset

We have a list of APK files that appeared on the Internet from January to April 2017. From this dataset, we select over 100,000 clean samples and 100,000 malicious samples randomly with uniform distribution. We take the classification of Avast Software anti-virus as our ground truth to decide whether given sample is malicious or not.

Taking decision of an anti-virus as the gold standard has, of course, significant disadvantages. Most notably we can have some samples in our sample set wrongly classified. This could be quite a problem for machine learning. However, a decision of commercial anti-virus program seems accurate enough [8] and overall false classification rate is reasonably low.

We have decided to choose decision of single anti-virus (and not for example majority vote over all available anti-viruses on VirusTotal) for the consistency. Different anti-viruses can have different policies what we should classify as malware (e.g. whether the application that gathers personal data and sends them to a server is malicious or still considered clean). Considering a combination of anti-viruses could very well confuse our machine learning algorithm.

In general, we believe that the advantages of this approach can easily outweigh the disadvantages. The most significant benefit of this method is that we can take large dataset for training our machine learning algorithm. Classifying every single sample by hand could take a lot of time, taking an automated decision allows us to work with hundreds of thousands of samples.

Furthermore, we do not filter out input samples, and the distribution of our dataset resembles the real distribution of applications on the Internet (separately on clean set and malware set). For example, we may tend to choose samples which all anti-viruses classified into the same category to avoid incorrectly classified input. However, by this method, we would restrict ourselves to samples that are easy to classify and hence the mesured result would not reflect the result we would get on a real-world dataset.

We have gathered 118,107 clean samples and 116,608 malware samples. To get an even better simulation of use of our classifier we have decided to take samples collected in first two months as the training set and the rest of them as the test set. The split after two months was chosen in order to keep both training and test set sufficiently large. After this division, we end up with the training set of size 145,030 samples with ratio 91,968 to 53,062 (clean to malware) and test set consisting of 89,685 samples 26,139 out of them are clean and remaining 63,546 are malware.

We manually ensured we get (roughly) 1:1 ratio of clean and malware samples which of course does not reflect the real distribution. We did this for a couple of

reasons. One of them is that we are not quite sure what the actual distribution is (though we assume that clean samples heavily outweigh malicious ones). Also if we had samples with the real distribution only small fraction of them would be malicious, and hence our machine learning algorithm would have only a small set of malware samples to learn from. This is a serious problem because the classifier would probably end up classify everything as clean or heavily overfit due to the small number of malware samples in the training set.

# 4. Feature extraction

As we said earlier, regarding feature extraction we mainly focus at method calls and API calls in particular. The reason for this choice is that the searching for method calls is reasonably easy and thus fast (a quick search in a DEX file and lookup in related tables is sufficient). Furthermore, API calls inside any given DEX file give reasonable overview of capabilities of the APK.

## 4.1 Retrieving API calls

We saw that retrieving method calls is straightforward. Binary parsing and couple of lookups can be done reasonably fast. Basically, we are interested in every single `invoke-*` instruction (actually there are couple different `invoke-` instruction depending on how is the method called – for example virtual methods have their own `invoke-vitaul[/range]` instruction, for our needs it is perfectly fine to ignore the differences and handle every method call the same).

Next step is to obtain API calls from those method calls. There are a couple of possible approaches to get the required data. One way to do it is to filter just those instructions that invoke some of the standard Android functions. For example, we can store every call of any method of `Landroid/telephony/SmsMessage;` class. As this list can be quite long, and more importantly it can vary over different versions of Android, we use a simple heuristic: we restrict ourselves to calls of any method belonging to Android or Java namespace (i.e. any class starting with either `Landroid/` or `LJava/)`.

This way of filtering API calls has some disadvantages. For example, anybody can create a class in one of these namespaces making extracted list of API calls less clear. There is also a more important problem. There may be a custom class (e.g. `MainActivity`) that extends a base class (e.g. `Landroid/.../AppActivity;`). Then if there is a call of a method of the custom class no "standard" API call is generated. Only something like `Lcom/example/MainActivity;->setTheme(I)V;` can be found in the bytecode in this case and no invocation of the method `Landroid/.../AppActivity;->setTheme(I)V` as we would like to. Thus there is a need for further pre-processing.

For this reason, we tried to improve the extraction of the features. There is the easily readable name of the superclass of given function in the DEX file. So before storing the method call, we can recursively replace every class in the description of the method by its superclass. We will repeat this until we reach the class that is not defined in the DEX file at all. After we resolve each class in the description of the method (method description contains the name of the class it belongs to, the

types of its parameters, and the return type) we get another pseudo-method call. As there is no definition of the class inside the DEX file the definition of obtained method is stored inside the Android core. We can consider this call an API call.

Furthermore, each `invoke-*` instruction contains the list of registers that hold the arguments for called function. We can take as features the number of occurrences of each `invoke-*` instruction including the whole list of registers (i.e. we would consider the number of occurrences of `LJava/util/List;->isEmpty()Z v0` as a feature). While the register numbers by itself do not seem quite as significant nor important, they are dependent on the structure of the code of the method where given API call is used hence useful information can be gained from them.

Finally, as mentioned in [4] it may be worth to filter out API calls that are executed inside advert packages. The content of these packages directly extends the code of actual application; thus we cannot easily distinguish the code of the package from the code that was written by the author of the application. The reasoning behind this filtering is that such packages often use some seemingly malicious API calls without actually being harmful. For this reason, we filter out 40 most common advert packages by name heuristic.

## 4.2 Limitations

As we mentioned earlier, the chosen approach for extraction of features has certain limitations. Most notably, regardless of final machine learning approach, any learning algorithm built upon these features cannot catch any malware that has no suspicious signs inside its DEX files.

This includes malware that has its payload inside binary libraries packed within the APK archive. By reading the DEX file, we can at least detect whether any binary library is used. In such case there is corresponding API call. This API call is not sufficient malicious indicator on its own (even benign application loads binary libraries in some cases) but can be certainly seen as a sign of suspicious behavior.

Furthermore, calling functions inside the DEX using reflection is particularly troublesome. When using reflection, we are not able to see the call of given method directly. However, we are at least able to see the call of reflection itself. Thus even without any advanced techniques the invocation of reflection can be easily seen and can serve as an indicator of obfuscation and potentially malicious behavior. Actual method call can sometimes be retrieved via dynamic analysis or at least more advanced static scan, but we do not use such techniques in order to preserve low processing time.

Other problematic samples are those that uses DEX file stored in non-standard location. While the main `classes.dex` needs to be inside every APK it can

dynamically loads additional DEX file. This additional DEX file can be either downloaded from the Internet or hidden among other resources. With the usage of dynamic analysis, we could retrieve this additional DEX files and analyse them, too. Nevertheless, even without any further investigation, we can see the usage of `DexClassLoader` inside the main DEX file, which can be seen as somewhat alarming.

Due to these reasons our algorithm would work best combined with some other classifiers that focus on these other types of malware. Nonetheless, most of the information is still contained inside the DEX, and we assume that this retrievable data are enough to classify the majority of samples successfully.

# 5. Initial machine learning

As we discussed in Chapter 3 the ratio of clean and malware samples of our dataset do not reflect the real distribution. For this reason, we describe the quality of the algorithm by recall (accuracy on clean set) and specificity (accuracy on malware set), rather than the overall accuracy. We can encounter two algorithms such that one of them has better recall and the other one higher specificity. In such case, we try to find a reasonable balance, but overall we consider recall as more important as we expect the number of clean samples in real usage being several times higher than the number of malicious ones. In other words, we prefer to have as few false positives (clean samples classified as malicious) as possible while the ratio of false negatives (malicious samples classified as clean) may be little higher.

From each gathered sample we extract every single API call by methods we described in Section 4.1. Thus for each sample we get lists of API calls that were obtained in various ways:

- with and without registers
- with and without resolving to superclasses
- with and without API calls invoked inside advert packages

This gives us overall eight ways to obtain API calls. Furthermore, for each of these API calls, we keep the number of invokes of given API call.

For this extraction, we use our own script that is able to process APK (and DEX) files. It scans code of every method and extracts every `invoke-*` instruction inside the DEX file.

Since we have over two hundreds of thousands of samples and there are over tens of millions of different API calls, we need to restrict ourselves to a smaller set of API considered calls.

As a first attempt, we take a straightforward rule and consider 5,000 API calls that are invoked in most malware samples. Our feature vector has a length of 5,000, and each feature represents the number of invocation of one of 5,000 most common pre-selected API calls. There are a couple of reasons for this decision. First of all, this approach is one of the easiest to implement and hence suitable for the first attempts. Furthermore, we do not want to consider the most common API calls on the clean set because some malware could use many "clean" API calls to "hide" its malicious behavior.

The 5,000 distinct most common API calls were chosen because 5,000-th most common API call is still used in about 5% of malware applications (specific fraction depends on the way of extracting features – more accurate results can be seen in

| Including registers | Yes | Yes | Yes | Yes |
|---|---|---|---|---|
| Resolv. to superclasses | Yes | Yes | No | No |
| Filtered adverts | Yes | No | Yes | No |
| Recall | 96.99% | 97.11% | 97.04% | 97.15% |
| Specificity | 83.97% | 84.44% | 83.19% | 84.73% |

| Including registers | No | No | No | No |
|---|---|---|---|---|
| Resolv. to superclasses | Yes | Yes | No | No |
| Filtered adverts | Yes | No | Yes | No |
| Recall | 96.99% | 97.12% | 97.07% | 95.31% |
| Specificity | 85.67% | 85.71% | 84.71% | 85.97% |

Table 5.1: Accuracy of first algorithms

Figure 5.1). Furthermore, by using over 5,000 API calls the training procedure started to be significantly time-consuming and adding more features would mean even longer processing time.

As a result, we got the list of APK files with feature vector containing the number of invocations of each of the most common API calls. This is the suitable input for any machine learning algorithm. We decided to take Extra Randomized Trees (the more randomized version of Random Forests) as a suitable machine learning algorithm. We used Python library `sklearn` and its `ExtraTreesClassifier` class of its `ensamble` module. We further discuss this choice in Chapter 6.

When we trained the algorithm and tested its accuracy on the test set, we obtained reasonable results. The recall is about 97%, and specificity is about 85%. For detailed results see Table 5.1. As we can see the best results were achieved when we resolve classes to their superclasses, we omit the parameters and include all classes including those that belong to advert packages. Thus we use this way of gathering API calls in more advanced machine learning approaches.

Overall we see that if the algorithm receives a clean sample on its input, it classifies the sample as clean fairly reliably. On the other hand, when it receives malicious sample it may happen that the sample will be falsely classified as clean. From the other point of view, a sample that the algorithm classifies as malware is most probably indeed malicious. However, it may not catch every malware it evaluates.

This makes sense because there is supposed to be some set of malicious characteristics in malware samples. If no suspicious sign is found inside the APK file, the sample should be marked as clean. It is quite possible that the malicious behavior is simply not detected inside some malware samples, but detecting suspicious behavior inside benign samples should be extremely rare.

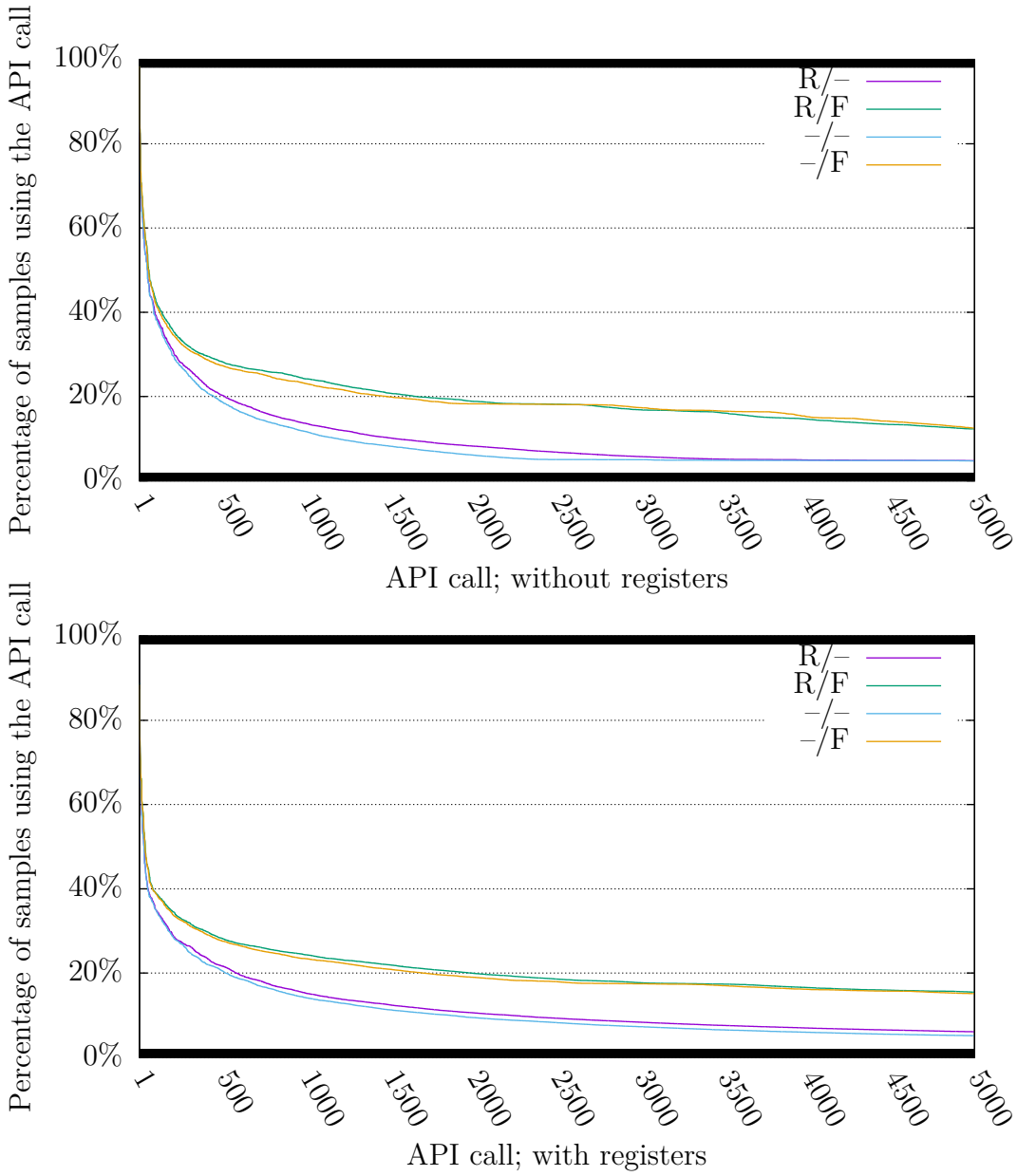


Figure 5.1: Number of malware applications that use given API at least once

The curves are described by two letters. “R” means we resolve to superclasses in this case and “F” means we filter out the API calls inside the advert packages. If the letter is missing these approaches were not use in case of corresponding curves. Top image covers the case that the registers are part of the features, while the lower one covers the case that only the names of methods are considered as features.

## 5.1 Suspicious API calls

In this section, we discuss API calls that are the most important for classifying whether given sample is malicious or not according to the used machine learning algorithm. Note that we omitted description of parameters and return type.

- `BufferedOutputStream;-><init>` and `FileOutputStream;-><init>`

  Both of these API calls are used to create new files on the device. This may be a sign of ransomware (malware that encrypts the files on infected devices and then demands money for decryption of those files). In order to work, this malware needs to create encrypted versions of the files and store them on the device. Other malware may use these API calls for downloading files with malicious payload from the Internet.

- `System;->loadLibrary`

  As we discussed in Section 4.2 this API call is used for loading binary libraries. This obfuscates the code in order to avoid possible detection.

- `Context;->getPackageManager`

  This API call is used for retrieving information about installed packages. Benign usually do not need such information. Usage of this API call can be seen as a potential privacy threat. As such, this API call serves as an indicator of malicious conduct.

# 6. Algorithm choice

As mentioned in the previous chapter we have chosen Extremely randomized trees (more precisely `ExtraTreesClassifier` from Python `scikit` library) as a suitable algorithm. We discuss this choice in this section.

When we take a look at input feature vectors, we can see that the zero is repeated extremely often (ratio of zeros in feature vectors varies from 70% to 90% depending on the method used for extracting API calls), while higher numbers are much less frequent. For details see Table 6.1 and Figure 6.1 (note that all the methods of extraction API calls generate similar histograms, thus arbitrary method was chosen for the figure in order to preserve the transparency). This is in correspondence of the meaning of these numbers. Many aplication use only some of the 5,000 most common API calls. Furthermore, if there is an invocation of an API call, the applications tend not to use the same API call on several different places of the bytecode.[1]

Hence the importance of difference between zero and one in a feature vector seems significantly higher than the importance of the difference between, for example, 500 and 501. In other words it is a great difference whether an application uses given API call (e.g. API call for sending premium SMS) at all or not, while it does not matter so much if it does so on 500 or 501 different places in the bytecode.

Furthermore, we can expect that there may be some combined dependencies. For example if there is both `Intent`'s `setType` and `putExtra` it may suggest that the application is sending e-mails. Such behavior may seem suspicious, especially combined with another API calls that reads potentially sensitive data such as the

[1] In the case that we resolve API calls to superclass we may and will have some method calls that eventually resolves to the same result. Thus, in this case, we should not talk about "same" method but "similar" method. General meaning still holds though.

| | | | | |
|---|---|---|---|---|
| Including registers | Yes | Yes | Yes | Yes |
| Resolv. to superclasses | Yes | Yes | No | No |
| Filtered adverts | Yes | No | Yes | No |
| Frequency of zeros | 88.65% | 71.40% | 86.01% | 71.40% |

| | | | | |
|---|---|---|---|---|
| Including registers | No | No | No | No |
| Resolv. to superclasses | Yes | Yes | No | No |
| Filtered adverts | Yes | No | Yes | No |
| Frequency of zeros | 84.21% | 70.25% | 82.78% | 69.69% |

Table 6.1: Frequency of zero in feature vector based on extraction method used

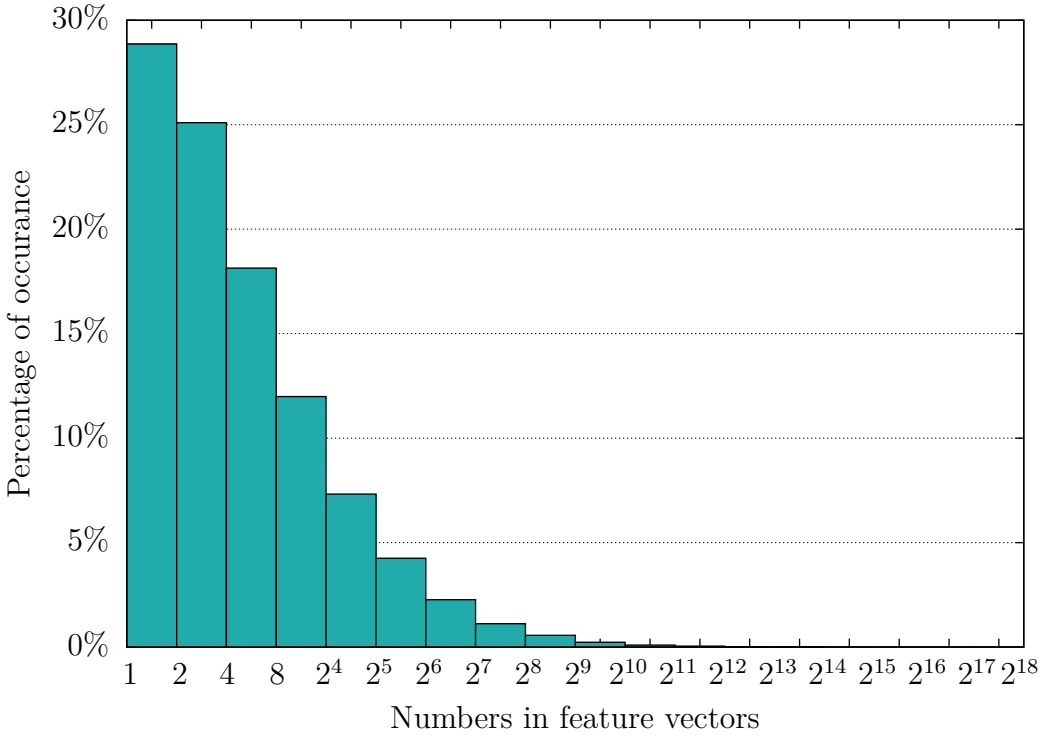


Figure 6.1: Histogram of non-zero numbers in feature vectors

contact list. Thus we may expect there to be such combined clues.

## 6.1 Linear regression

All of these facts suggests that linear regression models may give poor results. When we tried the logistic regression, we did not get that bad results though. Firstly, we needed to convert input vectors to booleans (i.e. we only keep the information if the given API call appears in an application and not how many times) to get good results. After this transformation, we achieved 91.5% recall and 86.94% specificity.

As these numbers do not seem to be as bad at first glance, having recall only 91.5% is quite dissatisfactory as it means the high rate of false positives. Given the problem, in hand, we consider false positives as significantly worse than false negatives as, we can expect the most samples that the algorithm will classify to be benign application making the recall even more important.

We are aware that we can actually lower the false positives rate and increase false negative rate by changing threshold of probability for classification, but we did not get significantly better results (we are not able to improve recall without a large drop in specificity). For the ROC curve see Figure 6.2 to get an idea about dependencies of these two quantities. Furthermore, we can improve such logistic regression by adding combined dependencies or taking more suitable preprocessing function than signum. Nevertheless, adding combined dependencies would increase the time for training (and evaluating) that is already quite high and finding a suitable function for preprocessing would not be so straightforward, too (e.g. as seen from Figure 6.1 even logarithmic function does not seem to be perfect). For these reasons, we rather choose an algorithm that addresses mentioned problems in a better way.

## 6.2 k-Nearest Neighbour

We also tried using the k-Nearest Neighbour classifier. Further description of the algorithm can be found in [9]. Overall this gave us poor results. With recall 88.82% and specificity 83.49%, we have already used better algorithms.

Furthermore, this algorithm is not suitable for our task for various additional reasons. One of them is that the algorithm needs large piece of memory to work as it needs to store whole training set. Since we have hundreds of thousands of samples and by each sample, we have up to 5,000 features there are over a billion integers that need to be stored. Even more problematic is the fact that finding closest neighbour or neighbours is very time-consuming. While classification time

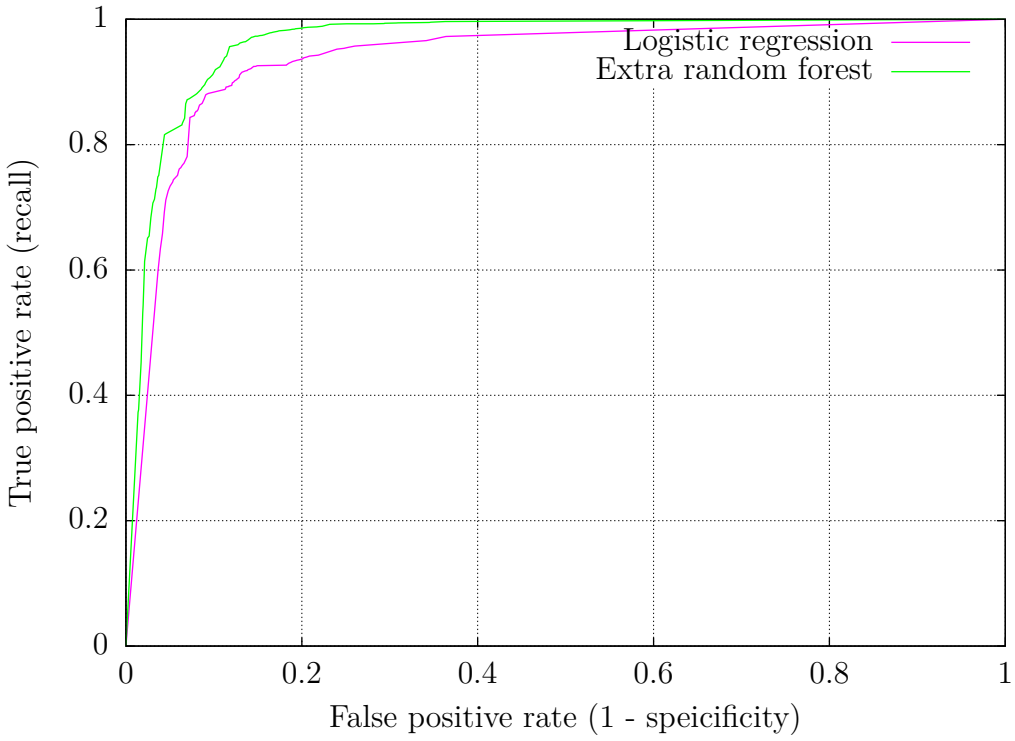


Figure 6.2: ROC curve

can be significantly shortened by using some approximation (this has been an object of extensive research, for example in [10]), we do not cover these advanced techniques and rather simple choose entirely different algorithm.

## 6.3 Other algorithms

For the completeness, we also tried some basic neural networks. Results were quite bad thought. The specificity did not exceed 75% even though we tried to use various parameters (number of layers, transfer function, etc.). While we believe this accuracy can be improved by further changing the architecture of the neural network, the basic result was bad enough that we chose to use different algorithms which have better results from the beginning.

We also tried to use support vector machine. Such algorithm, unfortunately, took so much time we were not able to get any results.

## 6.4 Decision Trees

Finally, we choose the Decision Tree classifier as it nicely deals with mentioned problems. Such algorithm does not need to keep large data for classification and produces decision fairly quickly. Furthermore, the training of the algorithm itself is reasonably fast.

It also does not have a problem with differences between zero and one in feature vector being more significant than the same difference between higher numbers; it just chooses the best split overall. This is a quite important advantage as no pre-processing is required.

Finally, given its nature it should find combined clues quite nicely.

We tried to use Random Forest algorithm instead of the single Decision Tree as a straightforward improvement. While this algorithm change provided a significant improvement in terms of recall and specificity we were able to get even better results with a slightly different algorithm.

As a final algorithm we chose so called Extremely Randomized Trees (algorithm similar to the Random Forests with randomness included when choosing the best split – further description can be found in [11]). This final choice was made purely on the basis of having slightly better results when using Extremely Randomized Trees compared to basic Random Forests.

# 7. Dataset specifics

We got a basic classifier capable of finding most malware, but if we take a look at our data, we may find strange patterns. There are quite a lot non-unique fingerprints (i.e. feature vectors generated as in Chapter 5). Only about 40% of the fingerprints are unique. Even more interesting is this statistic broken into clean and malware set. Our training clean set consists of 91,968 samples that generates 51,339 unique fingerprints (only about 56% of samples generate unique fingerprint). In the training malware set, we have 53,062 samples overall and only 15,044 unique fingerprints (percentage of malware samples that generates unique fingerprint is as low as 28%).

This fingerprint collision may reasonably happen in cases where there are a very low number of counted API calls. However, is seems strange in case when there are exactly 1,510 calls of one specific method, 961 other one, 628 yet another one and so on. Even more strange is the fact that there are malicious and benign samples with the same fingerprint.

Upon studying some of those samples, we have found a couple of common causes of this strange result. In almost every case this means that the APKs have (almost) identical DEX file. In some cases there is a difference in a single string, in the other cases, there are just renamed classes, and the bytecodes are the same. So while the APKs are technically different, the actual code and behavior of given applications is identical. This may be the reason why there are so few unique fingerprints for malware. In case of malware samples it makes sense to create samples that may be technically different but with the same behavior just to get several versions out on the market – for example, there may be a slightly higher chance to get through antiviruses as they may not catch all the versions of given malware. On the other hand, this kind of copying makes little sense when working with clean samples.

More interesting are the cases where some malicious and some benign samples generate the same fingerprint. We have taken a couple of samples and manually inspected them. It turned out that these were so called repacks of benign application. In order to create such malware, one needs to find an application that uses binary libraries. Such application is downloaded and unzipped, the library is then swapped for malicious one. Then the application is packed into the new APK and signed with a different key. The malicious behavior reveals itself once the code of the library is executed. As mentioned earlier this kind of malicious behavior is undetectable by simple static analysis of DEX file, and we would need to implement more advanced approaches such as dynamic analysis.

The similar and quite common case is when the DEX file inside the APK just loads another DEX file that is hidden among other files packed inside the APK

archive, and no other action is done by the original DEX file. While this behavior may seem strange at first by itself, this is actually also used in benign files in order to prevent the original code being retrieved.

Further examples include usage of so called master key exploits. This was a bug in older versions of Android in verifying packages. In general, this exploit means that the DEX file we analyse (the DEX file that is verified when installing the application) is not actually the DEX file that is executed.

Finally, a significant portion of samples with such problematic fingerprints turned out to be just wrongly classified by our ground rule. Remember that we take the decision of an antivirus as our golden standard and this actually may happen. While this may be a bit problematic for our machine learning algorithm, this fact has a quite important positive side. If we already have some classification of samples, these fingerprints may help to find potentially falsely classified samples. It is rare for two samples with the same bytecode to behave differently (i.e. it is unlikely that one of them is malicious and the other benign). Thus if we have two files that generate the same fingerprint while being classified into opposite classes there is high probability of one of them being falsely classified.

Note that colliding fingerprint means that there are samples that our classifier has no means of detecting. Overall we have 6,755 samples on training set that has the same fingerprint as at least one clean sample and one malware sample. This is significant 4.66% of our training set.

While this seems like a quite high percentage, further analysis reveals that the ratio of actually problematic samples may not be so high. The majority of groups of samples that have the same fingerprint contain far more samples of one class than the samples of the other one (i.e. the impurity of groups of samples with the same fingerprint is usually low – for further details see Figure 7.1). If we assume that our algorithm classifies each sample with ambiguous fingerprint as the prevailing class in the corresponding group, we get the percentage of guaranteed false negatives as low as 0.54% and 0.81% of guaranteed false positives (under the assumption that training set appropriately describes real distribution). As we can not distinguish samples with the same fingerprint, these are samples that we classify incorrectly no matter what algorithm we choose. Additionally, as these samples have an identical number of most 5,000 API calls we can assume that these samples (or at least vast majority of them) have identical bytecode thus any algorithm that is based on extracting API calls will classify them incorrectly, too.

These numbers are higher estimates though, at least some of the samples in these groups are wrongly classified by our gold standard classifier, making the number of samples that we classified incorrectly even lower.

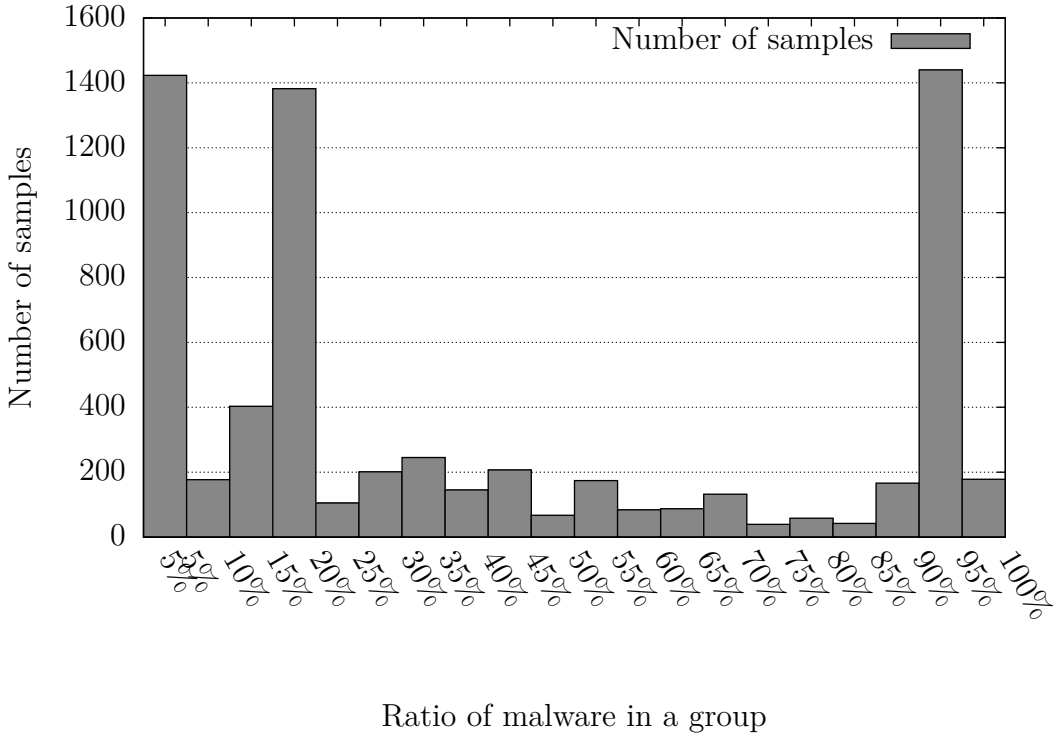


Figure 7.1: Number of samples with ambiguous fingerprint by the ratio of malware within the group of samples with given fingerprint

# 8. Ratio-based algorithm

Considering the most used API calls in the malware set has obviously some disadvantages. One of them is that the majority of extracted API calls are not significant but just the common API calls in general. For example when resolving API calls to superclasses most common call is `LJava/lang/Object;-><init>()V` i.e. the constructor of Java basic object class. This is because any invocation of constructor without parameter of class that does not inherit from any special Java nor Android class resolves to this call. Among the selected API calls we have more such insignificant calls that are just common in general.

## 8.1 Simple approach

To address this issue, we tried different approaches to select API calls. Instead of simply choosing the most common API calls in the malware set we choose the API calls based on the ratio of usage on malware set and clean set selecting those API calls that are relatively most common in the malicious applications.

The goal here is to cover significant API calls for all (or at least as many as possible) types of malware. For this reason we omit the information about how often is given API call invoked in an application for the matter of computing the ratio of the number of API calls, but we consider only the information whether given API call is invoked at all or not.

As shown in Chapter 7 it is common to have several files with the same bytecode, we need to avoid focusing on the samples with most common fingerprints too much. In order to do so, we modify the training set appropriately (we do not make changes in the test set to keep the obtained results as close to the real-world dataset as possible): For every sample in the training set we generate fingerprint as in Chapter 5, then we divide the samples from training set into the groups with the same fingerprint. We remove all but one sample from each group (i.e. we consider only one sample per group). In a case of groups with both malware and clean samples, we classify such sample of the class that prevails in the group.

By this sampling, we get 15,044 different malware samples and 51,339 different clean samples. We computed the ratio of API call usage on this selected subset of samples.

To avoid overfitting, we need to restrict ourselves to some subset of API calls. Otherwise, we would end up with thousands of very specific API calls that just happens to be specific for malware training set we have at our disposal. Firstly we tried to use API calls that are used in 5, 10 or 100 malware samples or more. For the reference, we note that 5,000-th most common API call was invoked on 1,431

different samples of our new training malware set.

Such thresholds gave us extremely overfitted algorithms. These algorithms correctly identified about 70% of samples of the clean set and barely any samples from malware set (about 8%). This approach has even quite poor performance even on training set as it is so focused on a small subset of samples. It correctly classified only a third of malware samples in the training set.

Due to this extreme overfitting, we need to increase the threshold of invocation to 1,000 different samples. Effectively we get a combination of ratio-based approach and our initial algorithm: By choosing the high threshold, we significantly reduce the number of available API calls to common ones, out of this reduced set we choose a subset of API calls based on their ratio of usage on malware samples and benign samples. This approach provides best results so far with 97.19% specificity and 92.24% recall.

Please note that this method for selecting features is a bit more expensive regarding the complexity. In the case of the simple approach described in Chapter 5 we skip all but 5,000 pre-selected API calls when scanning the clean set. Even though we need to evaluate the number of every API call in malware set to obtain the 5,000 most common ones, ignoring majority of API calls in the clean set may be significantly helpful regarding memory consumption as clean samples are on average larger in terms of size and the number of API calls. However, in this ratio-based approach, we cannot ignore as much API calls in the clean set (we can ignore at least some because we restricted our set of API calls to those calls that are invoked on at least 1,000 different malware samples) as we need to evaluate ratio for nearly every present call. Not to mention we also need to generate fingerprints based on simpler heuristic to obtain the groups of samples with identical bytecode in the first place.

## 8.2 Iterative approach

The simple ratio-based approach discussed in the previous section still has some room for improvement. In order to cover as many types of malware as possible, we propose a following iterative approach. At each iteration, we work with a small set of API calls that are sufficient to classify a subset of malware samples from our training dataset. Every iteration, we focus on the set of malware that we did not classify correctly in the previous iteration and extend our set of API calls appropriately. In this way, we are able to find features that are important for classifying multiple types of malware.

Even when using a small set of API calls our algorithm achieves high accuracy on the training set. In order to have set of problematically classifiable samples, we further temporarily decrease the complexity of the algorithm. As we use Decision

Trees, we significantly reduced allowed depth of such trees to achieve desired results. This reduced algorithm finally provides false negatives even on the training set, thus providing appropriate input to the iteratively learned algorithm.

In order to obtain our final algorithm, we follow these steps: We run the algorithm for five iterations. For each iteration, we compute its ratio of usage in the malware set and clean set and choose 1,000 API calls that are relatively most common in the malware set. We build reduced machine learning algorithm based on these 1,000 API calls. Given the reduced nature of used algorithm, there is a reasonably large set of falsely classified malware samples. In every other iteration, we compute the ratio of used every API call ignoring the malware samples that the algorithm in the previous iteration classified correctly. We choose 1,000 relatively most common API calls based on this new ratio and add them to our set of selected API calls. The new reduced algorithm is then learned using this extended set of API calls.

This way we obtain 5,000 significant API calls. Based on these calls we build one final algorithm without any restriction regarding the depth of the Trees. In this way, we obtained the best algorithm so far. Depending on how much we favor high recall over high specificity we can obtain a different evaluation of the algorithm. If a reasonable setting is used, the algorithm has recall 97.17% and specificity 93.65%. For overview how well the algorithm performs on different setting see Figure 8.1. In the figure there is added ROC curve of algorithm obtained in Chapter 5 for comparison.

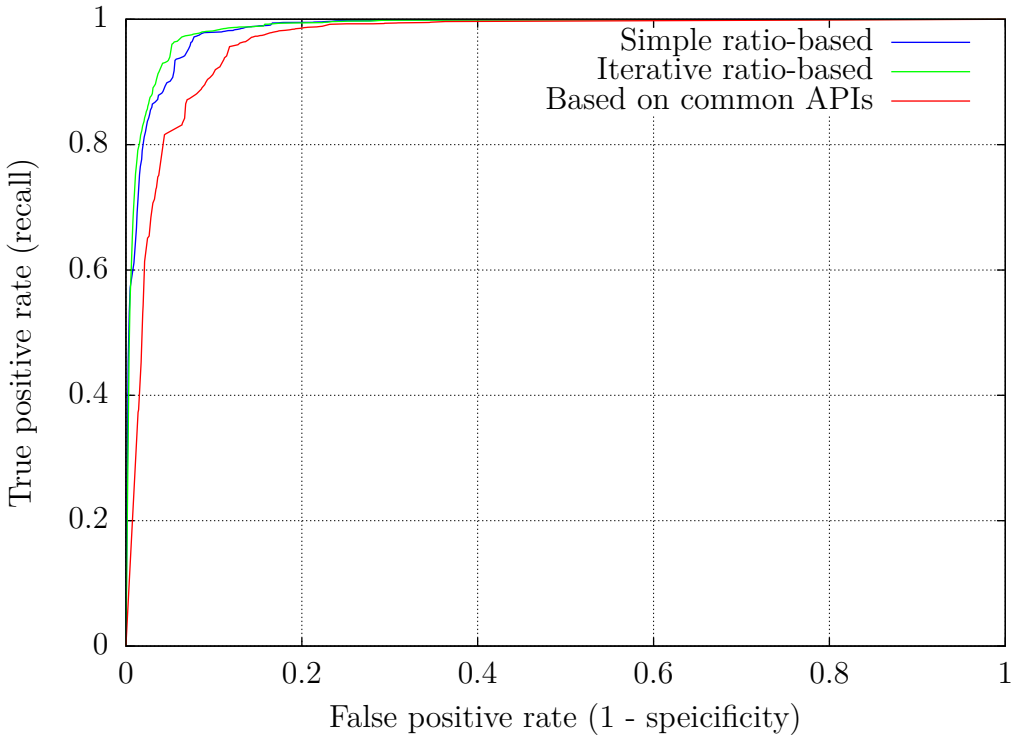


Figure 8.1: ROC curve of algorithm with ratio-based selection of API calls

# 9. Possible improvements

As we mentioned in Section 4.2 the approach we took has quite a few limitations, most notably it does rely on payload being inside the DEX file or files.

There are various ways to counter these limitations. For example, the algorithm would greatly benefit from a combining with a classifier for binary files. This help the classifier detects payload hidden inside binary libraries. Another way is to add By this way we could fight some other problematic aspects such as loading a DEX file with actual payload from a different file, using reflection.

Even the static scanning can be further improved. There is a possibility to take into account structure of the methods. We could work with the vicinity of API calls, for example, we could integrate into feature vector what strings or constants are loaded before the actual API call.

We could add the information about entry points into the feature vectors. It may be worth to mark the API calls that are executed in receivers (i.e. methods called upon some actions such as booting the device) or services (i.e. background processes).

To get an even more robust overview of application behavior, we could analyse control flow graphs. Such graphs serve exhaustive description of application code. On the other hand, this would bring another challenge – how to turn these graphs into features suitable for machine learning.

Furthermore, the algorithm could serve as a classifier of malicious samples into various families if provided with labelled input. This could give the user interesting input into the further analysis.

# Conclusion

In this thesis, we have discussed several ways to extract and refine API calls from an APK file in order to obtain input for further classification by machine learning. While analysing the extracted feature vectors, we found out that the majority of malware samples on the market are the samples that have been already seen with just changed names of methods or other insignificant changes. The way we extract API calls is a suitable approach for detecting such similarities. This detection on its own can be nicely used for finding falsely classified samples in the dataset.

The main objective of this thesis was to build an algorithm for detecting malicious Android applications. In the thesis, we tested multiple machine learning algorithms and numerous approaches to create appropriate feature vectors.

Our final algorithm uses the iterative approach in order to obtain appropriate API calls for the best classification. The algorithm is based on Extra Random Trees and it correctly classifies 91.17% of malware samples while preserving low positive rate of 2.83%, thus achieving the goal of the thesis. While this accuracy may not be good enough for using the algorithm as a stand-alone anti-virus, it certainly shows promising results. At least it can be used as a part of more complex classifiers that consists of several different classifiers to improve its accuracy.

# Bibliography


[1] Mobile operating system market share worldwide.

[2] Operating system market share worldwide.

[3] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[4] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*, pages 86–103. Springer, 2013.

[5] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.

[6] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. In *Intelligence and security informatics conference (eisic), 2012 european*, pages 141–147. IEEE, 2012.

[7] Dex format.

[8] Product review and certification report – may/2017.

[9] David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66, 1991.

[10] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1312, 2011.

[11] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

# List of Figures

# List of Tables