Slicing Droids: Program Slicing for Smali Code

Johannes Hoffmann, Martin Ussath, Thorsten Holz
Ruhr-University Bochum
firstname.lastname@rub.de

Michael Spreitzenbarth
Friedrich-Alexander-University Erlangen-Nuremberg
michael.spreitzenbarth@cs.fau.de

ABSTRACT

The popularity of mobile devices like smartphones and tablets has increased significantly in the last few years with many millions of sold devices. This growth also has its drawbacks: attackers have realized that smartphones are an attractive target and in the last months many different kinds of malicious software (short: *malware*) for such devices have emerged. This worrisome development has the potential to hamper the prospering ecosystem of mobile devices and the potential for damage is huge.

Considering these aspects, it is evident that malicious apps need to be detected early on in order to prevent further distribution and infections. This implies that it is necessary to develop techniques capable of detecting malicious apps in an automated way. In this paper, we present SAAF, a Static Android Analysis Framework for Android apps. SAAF analyzes smali code, a disassembled version of the DEX format used by Android's Java VM implementation. Our goal is to create program slices in order to perform data-flow analyses to backtrack parameters used by a given method. This helps us to identify suspicious code regions in an automated way. Several other analysis techniques such as visualization of control flow graphs or identification of ad-related code are also implemented in SAAF. In this paper, we report on program slicing for Android and present results obtained by using this technique to analyze more than 136,000 benign and about 6,100 malicious apps.

1. INTRODUCTION

Within the past several years, the popularity of smartphones and other kinds of mobile devices like tablets has risen significantly. This fact is accompanied by the large amount and variety of mobile applications (typically abbreviated as *apps*) and the increased functionality of the mobile devices themselves. Several mobile operating systems are available, with iOS and Android being the most popular ones according to latest studies [12]. As a side effect of this popularity, centralized application marketplaces like *Google Play* and *Apple's App Store* have massively grown. Such marketplaces enable developers to upload their own applications in a convenient way and users can download these apps directly to their mobile devices. Besides the official markets from platform vendors (*e. g.*, Google and Apple) and manufacturers (*e. g.*, Samsung and HTC), a large number of unofficial third-party marketplaces have emerged. Most of these markets contain thousands of apps and have millions of downloaded apps per month. For example, the official Google marketplace had at the end of 2011 nearly 400,000 applications in stock and more than 10 billion downloads [5].

This fast growth rate also has a downside: attackers have realized that rogue apps can be used to target smartphones and in the recent past, malicious software for smartphones became popular. According to Juniper, the number of malicious apps targeting the Android platform has risen more than 3,000% in the last half of 2011 to over 13,000 malicious samples [16]. With this number in mind, the chance of getting infected by malicious apps has risen to more than 7% depending on the country. Symantec's analysis of the *Android.Bmaster* malware [24] demonstrates the possibilities that an author of malware has: in this particular case, the amounts of money charged for a premium SMS that was sent by the malicious app is between 15 and 30 cents. Multiplying such small amounts with the number of potentially infected devices suggests that attacks against smartphones are a lucrative venue for adversaries. Unfortunately, it is hard to independently verify these numbers and objective measurements are missing. However, the often predicted rise of malicious software [4, 19, 21, 22, 25] might finally come true given the recent developments.

Considering these aspects, it is evident that malicious apps need to be detected early on in order to prevent further distribution and infections. In addition, we need to consider the growth rates of the app markets—a manual analysis is infeasible and automated approaches are needed to tackle this problem. This implies that it is important to develop efficient and automated analyzing techniques that allow for a reliable assessment of an analyzed app.

In this paper, we present SAAF, a static malware analysis framework for Android apps that is able to recognize suspicious behavior patterns in an automated way. SAAF analyzes smali code, a disassembled version of the DEX format used by *Dalvik*, Android's Java VM implementation. This approach enables us to perform a robust analysis that overcomes limitations of state-of-the-art tools that rely on disassemblers to Java code, a process that is fragile in practice [9]: Enck et al. introduced DED, a decompiler capable of analyzing about 94% of the total classes in the applications studied. In contrast, SAAF only failed to analyze a single app out of more than 140,000 apps during our evaluation.

Due to the prevalence and timeliness of malware for smartphones, this topic has received a lot of attention recently (*e. g.*, [6, 8, 9, 20, 30]) and our approach extends prior work in this area. SAAF performs static data-flow analysis [2, 11] (more precisely *program slicing* [1]) to backtrack the parameters used by a given method. This enables us to identify suspicious code regions within a given

SAC'13 March 18-22, 2013, Coimbra, Portugal.

app, we can for example detect whether a given app sends premium SMS in an automated way. Several other analysis techniques such as visualization of control flow graphs, a Manifest parser, and identification of ad-related code are also implemented in the tool. Based on our current prototype, we report on analysis results of more than 136,000 benign apps and about 6,100 malicious apps. We confirm several findings reported in other papers for smaller sample sets [9].

In summary, we make the following contributions in this paper:

- We introduce SAAF, a static analysis framework for Android apps. SAAF implements different program analysis techniques such as data-flow analysis and visualization of control flow graphs.
- In an empirical study, we analyzed more than 136,000 benign apps and about 6,100 malicious ones. Our results confirm observations previously reported for smaller sample sets and provide some new insights into typical Android apps.

2. HIGH-LEVEL OVERVIEW OF SAAF

SAAF is able to perform static analysis of Android applications and also supports a human analyst understanding a given app. The tool offers a graphical user interface and also supports automated analysis tasks such as automated unpacking and disassembling of applications. Furthermore, the tool can visualize the contents and code of a given app, and also offers several inspection tools to perform various program analysis tasks. In the following, we first provide a high-level overview of SAAF and then discuss implementation details specifically related to data-flow analysis in Section 3.

SAAF covers an important aspect of an app analysis process: *automated static analysis*. Our implemented variant of data-flow analysis [2, 11], namely *program slicing* [1], enables our tool to automatically search for constant values which are used as parameters in defined method invocations. This way, the analyst can for example determine if an application is able to send short messages to a hardcoded number—which would result in a strong misuse potential of this application. This search is called *static backtracking* and we describe it in detail in the next section. All found constants are stored in a MySQL database and are accessible for later analyses. Based on these results the analyst can, *e. g.*, let some heuristic decide which apps are worth a more thorough inspection because they might exhibit malicious behaviour: long sleep intervals, hardcoded telephone numbers, calls to `sudo` and so on. See Section 4.1 for more details.

Furthermore, SAAF does support an analyst doing a manual inspection. After an application is loaded within the framework, the analyst has access to options such as:

- Navigate through the application contents which is presented in a tree structure. Smali and optionally decompiled Java code is accessible, which is colored, and links to labels and methods are clickable.
- Control flow graphs (CFGs) can be generated and exported.
- SAAF offers the possibility to search for several program components, *e. g.*, strings and invocations.
- SAAF knows about ad package paths and can ignore classes inside them.

An automatic static analysis should run in the background, possibly on a large set of applications. Obviously, a GUI would hamper such a task and SAAF thus offers a lot of command line options to properly work without a GUI.

Being a static analyzer, SAAF is expected to work fast for our use case. Many applications need to be analyzed in a short amount of time to quickly get an idea which applications need to be investigated more closely by means of a more expensive manual or dynamic analysis. A static analysis of a typical app from our evaluation set is completed in less than 10 seconds on average. Sometimes the process is even faster, if the application is small.

3. STATIC BACKTRACKING

The ability to perform static data-flow analyses [2, 11] of method parameters (called *static backtracking* in this paper) is one of the core components of SAAF. It enables the analyst to define a set of methods of interest with their respective signature (parameters), in order to see whether they obtain any constants as input. This is of interest if, for example, the analyst wants to determine if some application is able to send short messages to a hardcoded number or with any hardcoded message text—both of which indicate a suspicious usage of this feature. There are numerous methods with parameters which are worthwhile to analyze and we later present results from our analyses in Section 4. First, we describe the general workflow of our tool and then provide more detail on the implementation of the slicing process. Afterwards, we illustrate the workflow with an actual example.

3.1 General Workflow

SAAF is based on static analysis methods and thus the first step is to dissect Android applications. Such applications are packaged in APK files, which are more or less ZIP compressed files with the compiled bytecode, additional metadata such as the Manifest file, and additional resources such as image or audio files. SAAF unpacks these APK files in the following way in order to perform the data-flow analysis and further analysis operations:

1. The analyst loads an Android application (APK file) or specifies at least one from the command line.
2. SAAF unpacks the contents of the app and generates smali files for all classes, using the *android-apktool*. Working directly on the bytecode enables us to obtained a detailed view of the code and overcomes limitations of tools that rely on decompiling the bytecode to Java code [9].
3. SAAF then parses the smali files and creates an appropriate object representation of its contents. More precisely, we process the Manifest file, basic blocks of the methods, fields, and all opcodes.

At this point, the static analysis can begin since all relevant information is unpacked and available in a usable form for further processing. SAAF will then perform the *program slicing* [1], which is explained in the next section. As noted above, our backtracking is a kind of data-flow analysis that specifically focuses on the analysis of method parameters to determine if certain parameters have static values that are relevant for the analysis process.

3.2 Program Slicing

In order to perform our backtracking of method parameters and to perform the slicing, a *slicing criterion* must be defined. In our case, the criterion consists of the following information:

- method name and full classname of its corresponding class,
- method signature, and the
- index of the parameter that shall be backtracked.

The slicing criterion fully specifies the relevant opcodes that invoke the desired methods in the analyzed application. Such a criterion enables us to search for *use-def chains*. We first search through all invoke opcodes for matching ones. Afterwards, we translate the given parameter index to a particularly used register in the decompiled code (*use* information). We check the previous opcodes in the corresponding basic blocks and determine whether the opcodes perform some operation with the currently tracked register. In other words, we perform a *backward slice*. Generally speaking, we iden-

tify all opcodes that modify or use the tracked register and will backtrack all the interactions until we find a *constant* (*def* information). We will explain in detail what is considered to be a constant in Section 3.3, the intuitive notion is that for example a hardcoded string would be a constant.

SAAF has an internal queue where all registers are stored which have not yet been backtracked. The queue is initially filled with the registers found during the first search for matching invoke opcodes and SAAF backtracks each register until the queue is empty. It is eventually filled as the logic finds opcodes inferring with the tracked register that make use of additional registers. The queue stores the registers name and its exact opcode location in the program in order to backtrack it at some time later.

If a tracked register v_x is overwritten by register v_y by the means of a *move* opcode, register v_y will of course be backtracked from this instruction on instead of v_x; this is called *aliasing* [26]. The same is true for all opcodes that put a result into the tracked register: all involved registers are added to the queue and are later backtracked. If the tracked register itself is not part of the value registers, it will not be backtracked anymore.

Until a found constant terminates the backward slicing, several opcodes require special handling in order to find constants of interest. Due to space constraints, we are unable to describe in this paper in detail how our tool deals with arrays, fields, basic block boundaries, method invocations, return values and the like. However, SAAF handles all opcodes and employs a combination of *backward* and *forward slicing* to find all constants which might get assigned as a parameter to the slicing criterion.

3.3 Constants

We terminate the analysis process when one of the following conditions holds:

- A constant value is assigned to the tracked register.
- An object reference is written into the tracked register.

These two cases end the search for constants for a tracked register: the first marks our goal to find assigned constants in the bytecode which finishes our search for *def* information. All opcodes of the *const-x* type provide such informations in addition to some others, *e. g.*, mathematical operations or initialized fields and arrays. They assign constants to registers, *e. g.*, strings or integers. In both cases the register will be overwritten and has an unknown semantically meaning before the assignment, which is irrelevant for our analysis. While the first one adds a resulting constant to our search, the second one terminates our search. If the register is overwritten with some reference, we will still see all involved constants for this object, which is explained in the example in the next section.

Apart from opcodes that put a constant of a specific type into the tracked register, we are also interested in the following aspects that might be encountered during the search, if they are somehow linked to the tracked register r:

- Fields and arrays with their types, names, initial and assigned values if a value is copied from them to r.
- Unknown (API) methods if they are called and return a value which is assigned to r. Known methods are part on the use-def chain and all return values are tracked.
- Variable names and types for found constants.
- Opcodes that overwrite r with something else, *e. g.*, if an exception is moved to it.

If such cases are found, they are added to the result set in a proper format and are tagged accordingly. We name them simply *search results* or constants for the rest of the paper. These results store additional meta-information such as the line number, the filename, and other relevant information that is helpful during the analysis process. These search results reveal used values such as telephone numbers in the best case and show at least relevant information such as a called method which returns used data in other cases.

3.4 Example

To give a small example of use-def chains, we provide a small Java program and the corresponding decompiled smali code in Listings 1 and 2. Both are shortened for the sake of brevity.

If two slicing criterions for this program are set to search for the parameters *destination number* and *message* (1st and 3rd parameter) of the *sendTextMessage()* method in Android's API, the following constants are identified (the *f* value is explained later on):

- The number `12345` (*f=0*) for the 1st parameter,
- the method `a.t.T.getDeviceId()` (*f=1*) and
- the string "`imei = `" (*f=1*) for the 3rd parameter.
- Additionally, the invocations of `getSystemService()` (*f=1*) and `toString()` (*f=0*) are found next to the string `phone` (*f=2*).

The telephone number is found by the means of program slicing: SAAF translates it to register *v1* in line 33 and starts the slicing process which promptly finds that the register gets a field value assigned at line 29. Possible values for this field are then found by searching for corresponding setters, which are found in the constructor in line 4-5. The register for the message content is translated to register *v3* in line 33. The slicing process then reveals that the register gets the value of register *p1* assigned in line 30, which relates to the method parameter in line 25. In a next step, SAAF searches for all method invocations for the method `sendMsg()` with the corresponding signature and package path. The slicing process will therefore continue at line 22 and will backtrack register *v4* as the corresponding parameter. Register *v4* gets some unknown value assigned through the `toString()` invocation of the *StringBuilder* object which is assigned to register v2 (see line 20-21). As the value itself is unknown, SAAF will now treat the method as a constant and further search for more constants that interfere with this object (*v2*). It determines that register *v0* is passed as a parameter to the method invocation of `append()` on the corresponding *StringBuilder* object referenced by *v2*. This way, in lines 17-19 register *v0* will also be tracked and this will lead to the call of `a.t.T.getDeviceId()` which returns and writes something unknown to *v0*, in this case, the IMEI. *v3* is now also tracked because `getDeviceId()` was invoked on it and as *v2* is still tracked, the string "`imei = `" and "`phone`" will be found in a similar way.

Note that our approach to track involved registers in method calls where no smali code is accessible, *e. g.*, API methods such as the aforementioned `append()`, might result in inaccurately identified constants. Nevertheless, this behavior is relevant for data structures where values are added and internally "mixed up". This way, the detection logic finds all parameters of previous method invocations

Listing 1: Sample Java code.

```
1  private String number = "12345";
2
3  public void work() {
4    StringBuilder sb = new StringBuilder();
5    String s = "imei = ";
6    sb.append(s);
7    TelephonyManager tm = (TelephonyManager)
8    getSystemService(Context.TELEPHONY_SERVICE);
9    String imei = tm.getDeviceId();
10   sb.append(imei);
11   sendMsg(sb.toString()); }
12
13 private void sendMsg(String text) {
14  SmsManager sms = SmsManager.getDefault();
15  sms.sendTextMessage(number, null, text, null, null); }
```

Listing 2: Sample smali code.

```
1  .field private number:Lj/l/String;
2
3  .method public constructor <init>()V
4    const-string v0, "12345"
5    iput-object v0, p0, Lxmpl;->number:Lj/l/String;
6    return-void
7
8  .method public work()V
9    new-instance v2, Lj/l/StringBuilder;
10   invoke-direct {v2}, Lj/l/StringBuilder;-><init>()V
11   const-string v1, "imei = "
12   invoke-virtual {v2, v1}, Lj/l/StringBuilder;->append(Lj
       /l/String;)Lj/l/StringBuilder;
13   const-string v4, "phone"
14   invoke-virtual {p0, v4}, Lexample;->getSystemService(Lj
       /l/String;)Lj/l/Object;
15   move-result-object v3
16   check-cast v3, La/t/TelephonyManager;
17   invoke-virtual {v3}, La/t/TelephonyManager;->
       getDeviceId()Lj/l/String;
18   move-result-object v0
19   invoke-virtual {v2, v0}, Lj/l/StringBuilder;->append(Lj
       /l/String;)Lj/l/StringBuilder;
20   invoke-virtual {v2}, Lj/l/StringBuilder;->toString()Lj/
       l/String;
21   move-result-object v4
22   invoke-direct {p0, v4}, Lxmpl;->sendMsg(Lj/l/String;)V
23   return-void
24
25 .method private sendMsg(Lj/l/String;)V
26   const/4 v2, 0x0
27   invoke-static {}, La/t/SmsManager;->getDefault()La/t/
       SmsManager
28   move-result-object v0
29   iget-object v1, p0, Lxmpl;->number:Lj/l/String;
30   move-object v3, p1
31   move-object v4, v2
32   move-object v5, v2
33   invoke-virtual/range {v0 .. v5}, La/t/SmsManager;->
       sendTextMessage(Lj/l/String;Lj/l/String;Lj/l/
       String;La/a/PendingIntent;La/a/PendingIntent;)V
34   return-void
```

to such objects and eventually constants. The only register that is not backtracked this way is the "*this*" reference for non-static invokes. This solution covers implicit method invocations such as constructors.

In order to later distinguish between good and probably inaccurate results, we tag such found constants as *fuzzy*. This tag is an integer *f* and indicates how (in)accurate the result is from our analysis point of view. A fuzziness value of *f=0* means that the result is accurate. Values higher than 0 indicate more inaccuracy and express our uncertainty about the precision of this result. Each such backtracked register will increment its fuzzy value by 1. The value is passed on to tracked registers, which are added to the register queue due to the currently tracked register. An added register will therefore inherit the fuzziness value of the register it is related to. This approach enables us to control the overestimation of the analysis phase and the fuzzy value implies a metric to "measure" this uncertainty. The fuzziness values *f* of the found constants in the example are given in parentheses in the above list.

4. EVALUATION

We now present several evaluation results obtained with the help of SAAF. In total, we successfully analyzed 136,603 free samples which we crawled from the *Google Market* (nowadays known as *Google Play*) in the mid of 2011. Only one application out of 136,604 could not be analyzed due to a 4 byte UTF-8 string that could not be stored in our MySQL database. Our malware set consists of 6,187 samples of old and new malware samples, and should cover samples from most of the known malware families. The evaluation was performed on two different servers, one was equipped with an Intel Xeon E5620@2.40GHz CPU and one with a E5630@2.53GHz CPU. Both have 24GB of RAM and no SSDs.

Table 1: Top 10 permissions.

Malware	%	Market	%
INTERNET	93.45	INTERNET	86.20
READ_PHONE_STATE	77.51	ACCESS_NETWORK_STATE	52.08
SEND_SMS	63.69	WRITE_EXTERNAL_STORAGE	34.01
ACCESS_NETWORK_STATE	52.63	READ_PHONE_STATE	32.46
WRITE_EXTERNAL_STORAGE	48.07	ACCESS_COARSE_LOCATION	22.98
RECEIVE_SMS	40.32	ACCESS_FINE_LOCATION	22.42
RECEIVE_BOOT_COMPLETED	36.90	VIBRATE	18.05
READ_SMS	22.42	WAKE_LOCK	12.36
ACCESS_WIFI_STATE	21.45	ACCESS_WIFI_STATE	11.01
VIBRATE	20.36	CALL_PHONE	8.99

For the rest of this paper, applications from the crawled *Google Market* will be typically referred to as samples from the Market and the malware samples will simply be called malware or malware set.

4.1 Analysis Results

Permissions: We analyzed the permissions used by both types of applications and present the results in Table 1. Analyzing the permissions reveals the permissions one would expect for malicious applications: almost all applications have the permission to access the Internet and most of the other permissions grant access to sensitive information like the various identifiers of the phone and, more importantly, to read and send short messages. Since criminals attempt to generate revenue from premium short message services, it is obvious that the corresponding permission is placed third in the total ranking. The only exception is the last permission, `VIBRATE`, which seems to be a remnant from trojanized popular games which can often be found in third party markets. This assumption is backed up by the fact that this permission is also ranked high in the analyzed applications from the *Google Market*.

The permission ranking for the Market applications is also similar as one would expect: most applications access the Internet, determine the network state, and identify the device. The only exception is the 10th permission `CALL_PHONE`. Applications with these permissions are hard to categorize. There are obvious ones such as dialers or SIP applications, but also a lot of applications that have the same prefixed package path, but fall into many categories as games, "reader applications" for websites, and so on. Many applications look like they were either developed by the same person(s) who reuse old components, or were created by application builders. One package path can be found in 1,100 applications and another one in over 600 applications. All these applications request the same permissions for the corresponding package path. It seems that the applications do not really make use of the granted permissions, especially of the `CALL_PHONE` permission.

Another interesting fact is the average number of requested permissions per applications. Applications from the Market request 4.4 permissions on average, while malicious applications requests 8.9 permission on average. A possible explanation is that a trojanized application needs the original permission set in addition to the new ones required for the added malicious code.

We are interested in the amount of applications which have the permission to access the Internet and additionally at least one permission that grants access to sensitive user data. Such applications *could* leak data by the means of sending it anywhere over the network connection. Our evaluation takes into account permissions that grant access to device serial numbers, SMS/MMS, location data, contacts, logs, account data, and the calendar. From the malware set, 90.6% have a permission set which allows such information leakage. This is not surprising, as most malware aims at either stealing data or sending premium rate messages. The Market appli-

cations are less likely to leak private information, but still 67,4% of these apps have a permission set allowing this. This is partly caused by ad frameworks which are permission hungry in order to identify their "customers", but additionally from unexperienced developers which accidentally over-privilege their applications, as Felt *et al.* found out [10].

IMEI/IMSI: Many application authors use the IMEI and IMSI number from a device as a unique identifier. While 53% of the malware determines at least one of these numbers through the appropriate API call, only 23% applications from the Market set do so. These numbers seem rather small, but compared to the applications which have the required permission `READ_PHONE_STATE`, cf. Table 1, these numbers look reasonable. One possible explanation is that the Market set is approximately one year old and the strategies of included ad frameworks have changed since, as one would expect a higher amount of these unique identifiers.

Executed Commands: We also backtracked the strings that are passed to the `exec` Android API method, which executes a command in a command shell. We found that malware mainly uses this function to call the `su` binary in order to install applications, remount partitions, or to enumerate running processes. Executing such commands with elevated privileges is crucial for the malware to obtain a higher level of rights. Interestingly, some of these commands can also be found in the Market set, albeit with a much lower frequency. Malware makes use of this method in 18.9% of the samples, while only 6.4% of the applications from the Market set call this function. Since many people root their smartphones to gain access to additional features, this percentage looks reasonable.

SMS API Usage: Finding applications that are able to send short messages to hardcoded numbers and possible hardcoded content might give a strong clue to classify such software as malicious. A total of 54% applications call such API methods in the malware set, while only 3.4% do so in the Market set. This huge number for malicious applications seems reasonable, as a lot of samples were found in recent time which abuse premium number services. While there are also legitimate use cases, a small amount of the applications in the official market also make use of this feature, *e. g.*, an SMS forwarder application. The following numbers are an excerpt from the numbers SAAF found during the analysis phase, all of them representing premium numbers that were identified in an automated way: 69229, 7781, 1121, 10086, 9685, 9818, 4157, 4545, 7790, 79067, 8014, 80888, 7061, 7250, 1065-71090-88877, and many more.

Alarm Usage and Thread.sleep(): During the analysis we also evaluated the usage of API functions that halt the execution of the active thread or which (periodically) wake the application up after a specified amount of time. While such a behavior is not malicious per se, it still might indicate that a given application might sleep for a certain amount of time first before becoming active, a typical behavior seen for Windows malware. As expected, such API usage was commonly found in both application sets: 56.9% of the malicious applications and 44.4% of the market applications call at least one such API function.

We were interested in the average amount of time applications use the `Thread.sleep()` method throughout both application sets. We therefore queried SAAF for all values that are passed to the sleep function. This is an integer value and specifies how many milliseconds the calling thread should sleep. We calculated the average value of all obtained values which are between 0 and 86,400,000 (24 hours) and which have a fuzzy level of 0, *i. e.*, are absolutely accurate backtracking results. The finding is very interesting, because both sets have very similar values. While malware goes to sleep for 21.9 seconds on average, market applications do so for 22.9 seconds. We have not analyzed why the results are almost identical and leave it for a future evaluation. As a last note, we found values that were negative—which should produce an exception during runtime—or extraordinary high (many days or weeks up to years).

System Services: Table 2 provides an overview of the overall used identifiers for *Android System Services* for all applications, determined by the used `Context.getSystemService(...)` API function parameter.

Table 2: Top 10 system services.

#	Malware	Market
1	phone	window
2	connectivity	layout_inflater
3	notification	location
4	layout_inflater	phone
5	activity	connectivity
6	alarm	audio
7	location	input_method
8	input_method	notification
9	window	sensor
10	audio	vibrator

Malware uses the *TelephonyManager* service the most, which is requested by the identifier "phone". This service has many methods which reveal a lot information about the state of the phone, whether a data connection is established, which network the phone is connected to, serial numbers, country codes and so on. Because many malware samples send short messages to premium services, the software often determines the country the phone is currently located, so the right premium number can be chosen; otherwise the premium rate service number might not be available if chosen wrongly. The *ConnectivityManager*—which is requested by the id "connectivity"—serves a similar role as the *TelephonyManager* regarding network connectivity changes and the like.

While malware seems to be very interested about the phone's state, samples from the market set most often care about window (activity) and layout states, which the first two used system services reveal for this sample set. The third entry "location" refers to the *LocationManager*. It is used to request the phone's location based on GPS or network coordinates. Most ad frameworks make use of this feature to serve targeted ads to the user.

The other requested system services throughout the two sets are more or less normally distributed, if one recalls that malware often piggybacks non-malicious applications or mimics those.

Loaded Libraries: Looking at the search results for libraries which are called by the analyzed applications, we found that malware most often loads a library called *androidterm*. This library is part of a terminal and provides functionality to access the Android command-line shell. We additionally found suspicious identifiers for loaded libraries such as *ScanVirus*, *VirusBackRunner*, *ScanController* and *scan_engine*. Such strings indicate that the malware might be related to some kind of fake anti-virus campaign, a popular attack vector in the desktop computer world [28] that recently became relevant for smartphones as well.

The legitimate applications mostly use applications to manipulate images, *e. g.*, *lept* (Leptonica lib) or libraries which are often used in games, *e. g.*, *andenginephysicsbox2dextension* (2D engine), *gdx* (libgdx, a game development lib) or *unity* (3D engine).

Reflections: SAAF evaluates whether an application loads classes or calls methods with the Java Reflection API. There are many

legitimate use cases for this which we will not describe in this paper, but reflections can also be used to obfuscate the control flow, since no method invocation can be observed if a method is executed through this API. We found that 36.5% of the malicious and even 57.5% of the Market applications make use of the reflection API.

It is of interest for us to determine for which goal the API is used. On the one hand, the called methods indicate that malware tries to hide some of its functionality, as the following exemplary classes are loaded throughout this sample set:

- `android.os.SystemProperties`,
- `telephony.SmsManager`,
- `android.telephony.cdma.CdmaCellLocation`,
- `java.net.InetAddress` and
- `android.content.pm.PackageManager`.

Effectively, the malware samples use reflections to hide the fact that they want to access sensitive APIs. On the other hand, Market applications mostly use the API inside the included ad frameworks, as the found parameters clearly indicate (but are omitted for the sake of brevity).

Intents: In the Android operating system, intents are used to perform special actions and to communicate with (other) application components. For the malware set, the two most commonly used intents clearly indicate the program's purpose, namely `SMS_SENT` and `SMS_DELIVERED` (both have no package prefix). In contrast, the market applications use normal system intents most often such as for example:

- `android.net.conn.CONNECTIVITY_CHANGE`,
- `android.intent.action.SCREEN_ON` and
- `android.intent.action.BATTERY_CHANGED`.

SSL Sockets: Although many popular libraries and most used API functions enable data exchange over the HTTP(S) protocol, some developers chose to create plain sockets to communicate with some end point. If security matters, they hopefully encrypt the data. Android provides the widely used SSL/TLS protocol for such cases. Creating a secure socket is rather easy, but old Android versions by default chose a ciphersuite that contains a lot of old ciphers. If not changed before a connection is established, old Android versions up to version 2.2 will provide ciphers such as `DES-CBC-MD5` or `EXP-RC2-CBC-MD5`. This behavior was changed in later versions, but Android 2.2 was quite common when the market applications were crawled and even now these old versions are installed on about 25.5% of all devices [3]. The distribution was calculated during a 14 day period ending at June 1, 2012, as stated on the website.

For us, it is interesting to see how many applications create SSL sockets and how many manually set the ciphersuites. We found that 153 malware samples do so and only 31 set ciphersuites. The same pattern is visible for market applications: only 943 out of 7,174 do so. This is a bad behavior for applications which are also available for old devices, as it might introduce severe security implications, *e. g.*, an attacker could perform a man-in-the-middle attack and trick the application into accepting a weak ciphersuite.

Content Provider: Most of the data stored on Android devices can be accessed through so called *Content Providers*. To access data, a developer has to form a URI describing the desired content, for example `content://calendar/calendars`. The provider parses this URI and will serve requested data from the according data structures, and it additionally ensures that required permissions are at hand (if required). Applications from the Google Market set have a very diverse set of used URIs which access a lot of different Android Content Providers throughout the applications such as the calendar, contacts, and SMS/MMS databases. The most accessed content by malware samples is related to the SMS/MMS Content Providers, by a large margin. Of course the malware samples also access the aforementioned providers, but most URIs are SMS/MMS related.

Native Code and Classloaders: Java and also Android allow calls to native code which is not available in Java bytecode but is located in native libraries, *e. g.*, `.so` shared objects. We found that 9.43% of the malicious apps perform such calls and 4.61% of the market applications do so.

Another feature of Java and Android is the ability to load classes from arbitrary destinations, *e. g.*, from the Internet, and to run the included code within these classes. While not being malicious per se, this feature can be abused to hide (malicious) code from being analyzed. We analyzed how many applications use the ClassLoader (sub)classes in any way and found that only 0.87% of the malicious apps to so. On the contrary, 22.05% of the apps from the market do so. This high number seems very unlikely, so we invested how many apps contain code which uses classloaders and which is not located in ad framework package paths. Excluding these ad classes, 3,04% of all apps remain. This is still almost 4-times the amount for malicious apps. We leave it for a further evaluation to determine what these apps actual do, but seeing this feature as a typical one for malicious apps to evade detection or analyses seems vague.

Crypto: The last finding we want to report about is the presence of cryptography in the analyzed applications. SAAF analyzed the applications and looked for usages of the `doFinal()` method in the class `java.crypto.Cipher` method. If an application makes use of the built-in cryptographic routines, they most likely call this function.

Out of all malware samples, 17.6% use this function and 25.3% of all Market applications do so. This indicates that the usage of cryptographic functions is no indicator for malicious behavior. Unfortunately, SAAF is at this point unable to detect self-written cryptographic functions, so the real usage of cryptographic functions in the malware is likely higher.

4.2 Malware Analysis

We now give a brief excerpt of the analysis of three kinds of malware samples. The first sample is from the *DougaLeaker.A* family and was discovered in April 2012 with a MD5 checksum of `91d57eb7ee2582e0600f21b08dac9538`. The sample steals contact data and sends it to a remote host. For this sample, SAAF was able to extract two URLs to where stolen data might be transmitted, namely

- `http://depot.bulks.jp/get44.php` and
- `http://depot.bulks.jp/movie/movie44.mp4`.

These URLs were found as arguments to the Android and Apache URI parsers. A manual analysis revealed that the data is sent to the first found URL.

The next sample belongs to the *Rufraud* family and presents itself as the popular game *Angry Birds*. The sample itself is a fake installer and sends premium rate short messages after the user is tricked into agreeing being charged. These samples were found at the end of 2011 in the *Google Market* and our sample has the MD5 checksum `95a04cfc5ed03c54d4749310ba29dda9`. Depending on the country the user resides in, the malware sends premium SMS to different providers. SAAF is able to extract 19 distinct destination numbers (1121, 1171, 1645, 17013, 1874, 4157, 4545, 69229, 7540, 7781, 7790, 79067, 8014, 80888, 81185, 9014,

90901599, 9090199, 92525), but SAAF is not able to extract the text part of the messages. Instead, we found a string which is used as a key to decrypt the message text that is supposed to be sent (`E273FED8415F7B1D8CFEAC80A96CFF46`). The results additionally reveal that the sample checks whether the messages were successfully sent or not, as the following two strings indicate which are passed to the Android IntentFilter class, namely `SMS_DELIVERED` and `SMS_SENT`. Such results give a strong hint that the analyzed application has malicious intents.

As a third example we provide an overview of three variants of the *Gone in 60s* "malware" that appeared in September 2011 on the official *Google Market*. This malware copies private information as the SMS database and contact details from a phone to a remote location. The user is then presented with an access code which enables him to see all the stolen (copied) data and the application removes itself. All this happens in less than 60 seconds, hence the name. For each sample SAAF found several constants, the most interesting ones being an URI to query the SMS database ("content://sms") and an URL pointing to the drop zone which is used in an HTTP POST request ("http://gi60s.com/upload.php").

These results indicate the desired functionality of this particular application. It accesses the SMS database and connects to a website to where it uploads the stolen data. These results extracted by SAAF support a malware researcher and enable her to obtain an initial overview of a given Android app.

5. LIMITATIONS

We now clarify the limitations of our current prototype. Since SAAF is a static analyzer, it has all the drawbacks static analyzers have in general [23]. For example, information available only at runtime is not available, and the usage of encryption methods and obfuscation put a heavy burden on such tools. Despite these general drawbacks that SAAF shares with other static analysis tools, our framework could be improved in the following ways. First, although being able to detect the usage of the *Reflection* API and additionally often also the called class and method, SAAF currently does not backtrack into methods found this way while backtracking a register. This is a feature we intend to implement in the future. Despite the effects of heavy obfuscation and dynamic code loading, the analyst can easily see if and to what extend such features are used as the automatic analysis will reveal these. Such indicators might give a strong clue of the intents of an application.

The fact that SAAF is able to analyze all but one application does not mean that it is guaranteed to always find all constants. There are likely corner cases where code constructs are misinterpreted and something is missed. For obvious reasons, we have been unable to check all found results against expected results.

6. RELATED WORK

Security aspects of smartphones have received a lot of attention recently and we are not the first to introduce analysis techniques for mobile apps. In the following, we discuss how SAAF relates to prior work in this area. We focus our discussion on other static analysis approaches, especially since many papers have been published in this area concurrently to our work. Afterwards, we also briefly discuss dynamic analysis techniques, which are basically complementary to our approach.

ANDROGUARD is a toolset to decompile and analyze a given application, with the goal to detect malicious apps via signature matching [6]. We do not require decompilation, since this processing step might introduce imprecision in cases where the decompiler does not successfully reconstruct valid Java code, a problem that often occurs in practice [9]. Furthermore, SAAF does not only rely on pattern matching or static signatures and ANDROGUARD is not working in a fully automated way, which implies that the tool is not designed to handle a large corpus of apps.

Recently, Kim et al. presented a tool called *ScanDal* that is able to automatically detect privacy leaks [17]. The authors evaluate SCANDAL with a very small sample set containing only 8 malicious and 90 benign samples. In addition, SCANDAL has some limitations, as it does not support reflection-related APIs and JNI. Our evaluation set is three orders of magnitude larger and we expect that it provides a more realistic overview of the current threat landscape.

Another tool closely related to our approach is DED, a decompiler that recovers source code of Android applications directly from their installation images [9]. The tool infers lost types, performs Dalvik VM-to-Java VM bytecode retargeting, and translates class and method structures. Furthermore, the authors studied 1,100 benign Android apps to better understand the security characteristics. Compared to DED, SAAF has a higher coverage: the authors of DED report that about 94% of the total classes in the applications studied could be recovered. In contrast, we found that we could analyze more than 99.99% of more than 140.000 apps that we studied, with only 1 exception. Our analysis results confirm the findings reported by Enck et al. on a larger sample set.

Grace et al. demonstrate with their RISKRANKER approach how to scan application markets for unknown malware samples [15]. Their approach is two-tiered: within the first detection stage, RISKRANKER tries to find apps with native code that use known root exploits or that send premium SMS messages. The second stage deals with obfuscated apps and tries to detect malicious apps that are encrypted or that load additional code. The system found 322 new instances of malware in a total set of 118,318 applications collected from various Android markets. The focus of SAAF lies on static analysis techniques.

There are several papers that touch on various Android-related topics and these papers also have a small overlap with our work. We briefly discuss this kind of work but do not elaborate it in detail since the overlap is rather small. For example, Elish et al. implemented a static approach for malware identification based on user-centric data dependence analysis [7]. DROIDMOSS uses the fuzzy hashing method to detect repackaged apps in third-party Android markets [29]. The detection of privacy leaks and the unwanted access to user-related data on the Android platform has been explored extensively in the past. Some examples for detecting and mitigating these leaks are provided by Gibler et al. [13] and King et al. [18]. Regarding the in-app ad-networks, Stevens et al. [27] and Grace et al. [14] performed some similar research.

One of the first dynamic malware and privacy leak detection systems was TAINTDROID [8]. It is an efficient and dynamic taint tracking system that provides real-time analysis reports by leveraging Android's execution environment. This system was complemented with a fully automated user emulation and reporting system by Lantz and is available under the name DROIDBOX [20]. Both systems are unable to track native API calls and are slow compared to a static analysis approach like SAAF or the ones discussed above. DROIDRANGER implements a combination of a permission-based behavioral foot printing scheme to detect samples of already known malware families and a heuristic-based filtering scheme to detect unknown malicious apps [30]. With this approach, the authors were able to detect 32 malicious samples inside the official Android Market back in June 2011. Within their dynamic part, they use a kernel module to log system calls used by known Android malware. As discussed above, such dynamic

approaches are complementary to SAAF: static and dynamic analysis approaches can be combined to enhance the analysis results.

7. CONCLUSION

In this paper, we introduce SAAF, a static analysis framework for Android apps. SAAF analyzes smali code, a disassembled version of the DEX format used by *Dalvik*, Android's Java VM implementation. Since no decompilation to Java is performed, the approach is more robust and resilient to code obfuscation techniques or insufficiencies of decompilers. Compared to state-of-the-art tools that are capable of analyzing about 94% of the total classes in the applications studied, SAAF successfully analyzed more than than 140,000 apps studied for this paper. SAAF mainly performs data-flow analysis based on program slicing to analyze the structure of an app, but is also capable of performing other kinds of analysis. We surveyed a large collection of Android apps and malware, and our evaluation results confirm some insights previously obtained on smaller sample sets [9].

Acknowledgments: This work has been supported by the Federal Ministry of Education and Research (grant 01BY1020 – MobWorm). We would like to thank Tilman Bender, Christian Kröger, and Hanno Lemoine for their work on SAAF.

SAAF is written in Java and its source code can be found at `http://code.google.com/p/saaf/`.

8. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. *SIGPLAN Not.*, 25(6), June 1990.

[2] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3), Mar. 1976.

[3] Android Developers. Platform Versions, June 2012. `http://developer.android.com/resources/dashboard/platform-versions.html`.

[4] M. Becher, F. C. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf. Mobile Security Catching Up? Revealing the Nuts and Bolts of the Security of Mobile Devices. In *IEEE Symposium on Security and Privacy*, 2011.

[5] E. Chu. 10 Billion Android Market downloads and counting, Dec. 2011. `http://googlemobile.blogspot.com/2011/12/10-billion-android-market-downloads-and.html`.

[6] A. Desnos and G. Gueguen. Android: From reversing to decompilation. *In Proc. of Black Hat Abu Dhabi*, 2011.

[7] K. O. Elish, D. Yao, and B. G. Ryder. User-Centric Dependence Analysis For Identifying Malicious Mobile Apps. In *Workshop on Mobile Security Technologies*, 2012.

[8] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[9] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security Symposium*, 2011.

[10] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *ACM Conference on Computer and Communications Security*, 2011.

[11] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Comput. Surv.*, 8(3), Sept. 1976.

[12] Gartner Inc. Gartner Smartphone Marketshare 2012 Q1, May 2012. `http://www.gartner.com/it/page.jsp?id=2017015`.

[13] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. *Trust and Trustworthy Computing*, June 2012.

[14] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[15] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. *International Conference on Mobile Systems, Applications and Services*, 2012.

[16] Juniper Networks Inc. 2011 Mobile Threats Report, February 2012. `http://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2011-mobile-threats-report.pdf`.

[17] J. Kim, Y. Yoon, and K. Yi. ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications. *Workshop on Mobile Security Technologies (MoST)*, 2012.

[18] J. King, A. Lampinen, and A. Smolen. Privacy: Is There An App For That? *Symposium On Usable Privacy and Security*, 2011.

[19] J. Kleinberg. The Wireless Epidemic. *Nature*, 449(20), 2007.

[20] P. Lantz. droidbox - Android Application Sandbox, February 2011. `http://code.google.com/p/droidbox/`.

[21] N. Leavitt. Malicious Code Moves to Mobile Devices. *IEEE Computer*, 33(12), 2000.

[22] N. Leavitt. Mobile Phones: The Next Frontier for Hackers? *IEEE Computer*, 38(4), 2005.

[23] A. Moser, C. Kruegel, and E. Kirda. Limits of Static Analysis for Malware Detection. In *Annual Computer Security Applications Conference (ACSAC)*, Dec. 2007.

[24] C. Mullaney. A Million-Dollar Mobile Botnet, Feb. 2012. `http://www.symantec.com/connect/blogs/androidbmaster-million-dollar-mobile-botnet`.

[25] J. Oberheide and F. Jahanian. When Mobile is Harder Than Fixed (and Vice Versa): Demystifying Security Challenges in Mobile Environments. In *Workshop on Mobile Computing Systems and Applications (HotMobile)*, February 2010.

[26] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5), Sept. 1994.

[27] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating User Privacy in Android Ad Libraries. *Workshop on Mobile Security Technologies (MoST)*, 2012.

[28] B. Stone-Gross, R. Abman, R. A. Kemmerer, C. Kruegel, and D. G. Steigerwald. The Underground Economy of Fake Antivirus Software. In *Workshop on Economics of Information Security (WEIS)*, 2011.

[29] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Droidmoss: Detecting repackaged smartphone applications in third-party android marketplaces. *ACM Conference on Data and Application Security and Privacy*, 2012.

[30] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. *Symposium on Network and Distributed System Security*, 2012.