Efficient Code Obfuscation for Android

Aleksandrina Kovacheva

University of Luxembourg
aleksandrina.kovacheva.001@student.uni.lu

Abstract. Recent years have witnessed a steady shift in technology from desktop computers to mobile devices. In the global picture of available mobile platforms, Android stands out as a dominant participant on the market and its popularity continues rising. While beneficial for its users, this growth simultaneously creates a prolific environment for exploitation by vile developers which write malware or reuse software illegally obtained by reverse engineering. A class of programming techniques known as code obfuscation targets prevention of intellectual property theft by parsing an input application through a set of algorithms aiming to make its source code computationally harder and time consuming to recover. This work focuses on the development and application of such algorithms on the bytecode of Android, Dalvik. The main contributions are: (1) a study on samples obtained from the official Android market which shows how feasible it is to reverse a targeted application; (2) a proposed obfuscator implementation whose transformations defeat current popular static analysis tools while maintaining a low level of added time and memory overhead.

Keywords: Android, bytecode obfuscation, Dalvik, reverse engineering.

1 Introduction

Ever since the early 1990s, devices combining telephony and computing have been offered for sale to the general public. Evolving to what is presently referred to as "smartphones", their extensive usage is indisputable: a report from February 2013 estimated the total number of smartphone devices sold only in 2012 as surpassing 1.75 billion [1]. Due to their wide ranging applicability and high mobility smartphones have been preferred over stationary or laptop computers as access devices to personal information services such as e-mail, social network accounts or e-commerce websites. By the end of 2012, the mobile market was dominated with a ratio of 70% by the Android platform [1].

The huge market share as well as the sensitivity of the user data processed by most applications raise an important security question regarding the source code visibility of the mobile software. Firstly, developers have an interest of protecting their intellectual property against piracy. Moreover, an alarming 99% of the mobile malware developed in 2012 has been reported to target Android platform users and inspection reveals both qualitative and quantitative growth [2]. Hence, Android applications code protection is crucial to maintaining a high level of

B. Papasratorn et al. (Eds.): IAIT 2013, CCIS 409, pp. 104–119, 2013.

trust between vendors and users which in turn reflects in a correct functioning of the Google Play market itself.

In general, there are two main approaches towards software protection: enforcing legal software usage policies or applying various forms of technical protection to the code. This work concentrates on the latter, more precisely on a technique called *code obfuscation*. In the context of information security the term obfuscation encompasses various deliberately made modifications on the control-flow and data-flow of programs such that they become computationally hard to reverse engineer by a third party. The applied changes should be semantic preserving with ideally negligible or minor memory-time penalty.

2 Related Work

The idea of obfuscation was first suggested by Whitfield Diffie and Martin Hellman in 1976 in their paper "New Directions in Cryptography" [3] where they introduced the usefulness of obscuring cryptographic primitives in private key encryption schemes for converting them into public key encryption schemes. In 1997 C. Collberg et al. presented obfuscation together with a relevant taxonomy as a way of protecting software against reverse engineering [4]. The theoretically computational boundaries of obfuscation are examined in a work titled "On the (Im)possibility of Obfuscating Programs" by Boaz Barak et al. in 2001 [5]. In the latter, the authors give a framework for defining a perfect or *black-box* obfuscator and show that such a primitive cannot be constructed. Although in theory perfect obfuscation is proven impossible, there is a discrepancy with its practical aspects.

In applied software engineering obfuscation is used on various occasions, mainly for protection against reverse engineering, defending against computer viruses and inserting watermarks or fingerprints on digital multimedia. Usually, a software vendor does not try to hide the functionalities of their program, otherwise software would never be accompanied by user documentation which is not the case. Rather, the developer aims at making unintelligible the implementation of a selected set of functions.

To recover the original code of an application, bytecode analysis is most often used. By applying both *dynamic* and *static* techniques, in the context of Android applications, it is possible to detect an over-privileged application design, find patterns of malicious behavior or trace user data such as login credentials. Due to its simplicity over bytecode for other architectures, Dalvik bytecode is currently an easy target for the reverse engineer. The following listed set of analysis tools and decompilers is a representative of the largely available variety: `androguard` [6],`baksmali` [7], `dedexer` [8], `dex2jar` [9], `dexdump` [10], `dexguard` [11], `dexter` [12], `radare2` [13].

Evidently, there are numerous tools to the help of the Android reverse engineer. They can be used either separately or to complement each other. The same diversity cannot be claimed for software regarding the code protection side, especially concentrating on Dalvik bytecode. Most existing open-source

and commercial obfuscators work on source code level. The reason is that effective protection techniques successfully applied on Java source code have been suggested previously [14]. Furthermore, Java code is architecture-independent giving freedom to design generic code transformations. Of the here listed Dalvik bytecode obfuscation tools the first two are open-source, the last (which also modifies the source code) is commercial: `dalvik-obfuscator` [15], `APKfuscator` [16], DexGuard [17]. Unfortunately, the open-source tools have the status of a proof-of-concept software rather than being used at regular practice by application developers.

3 A Case Study on Applications

There exist an extensive set of works examining Android applications from the viewpoint of privacy invasion as can be seen in [18–21]. The here presented case study aims to show that bytecode undergoes few protection. If present, obfuscation is very limited with regards to the potential transformation techniques which could be applied, even for applications which were found to try protecting their code.

3.1 Study Methodology

To obtain a sample apps set, a web crawler was developed downloading the 50 most popular applications from each of the 34 categories available on the market. There were applications in repeating categories, thus the actual number of the examined files was 1691. The study was performed in two stages. Initially, automated static analysis scripts were run on bytecode for a coarse classification the purpose of which was profiling the apps according to a set of chosen criteria. A secondary, fine grinding examination, was to manually select a few "interesting" apps and looking through the code at hand. The following enumerated were used for apps profiling:

1. **Obfuscated versus non-obfuscated classes.** A study on the usage of `ProGuard` which is an available in the Android SDK code obfuscator was an easy target. Since this tool applies variable renaming in a known manner, the classes names and contained methods were processed with a pattern matching filter according to the naming convention i.e. looking for minimal lexical-sorted strings. A class whose name is not obfuscated, but contains obfuscated methods was counted as an obfuscated class.
2. **Strings encoded with Base64.** Several applications were found to contain "hidden" from the resources files in the form of strings encoded with Base64. Manual examination of a limited number of these revealed nothing but `.gif` and `flash` multimedia files. However, this finding suggests that it might be common practice that data is hidden as a string instead of being stored as a separate file which is why this criteria was considered relevant to the study.

3. **Dynamic loading.** Dynamic loading allows invocation of external code not installed as an official part of the application. For the initial automation phase its presence was only detected by pattern matching check of the classes for the packages:
   `Ldalvik/system/DexClassLoader`
   `Ljava/security/ClassLoader`
   `Ljava/security/SecureClassLoader`
   `Ljava/net/URLClassLoader`
4. **Native code.** Filtering for the usage of code accessing system-related information and resources or interfacing with the runtime environment was also performed. For the coarse run only detecting the presence of native code in the following packages was considered:
   `Ljava/lang/System`
   `Ljava/lang/Runtime`
5. **Reflection.** The classes definition table was filtered for the presence of the Java reflection packages for access to methods, fields and classes.
6. **Header size.** The header size was also checked in referral to previous work suggesting bytecode injection possibility in the `.dex` header [22].
7. **Encoding.** A simple flag check in the binary file for whether an application uses the support of mixed endianess of the ARM processor.
8. **Crypto code.** With regards to previous studies on inappropriate user private data handling as well as deliberate cryptography misuse, the classes were also initially filtered for the usage of the packages:
   `Ljavax/crypto/`
   `Ljava/security/spec/`

3.2 Results Review

The distribution of applications according to the percentage of obfuscated code with ProGuard is shown on table 1. On table 2 are noted the absolute number of occurrences of each factor the apps were profiled for. The automated study reveals that encoding strings in base64 is quite common practice: 840 applications containing a total of 2379 strings were found and examined, shown on table 3. To determine the file format from the decoded strings the `python magic` library[1] was used. Unfortunately, 1156 files which is 48.59% of the total encoded files could not be identified by this approach and using the Unix `file` command lead to no better results. The remaining set of files was divided into multimedia, text and others. As a final remark to table 3 is that the percentage marks the occurrences in the 1241 successfully identified files.

A set of several applications was selected for manual review, the selection criteria trying to encompass a wide range of possible scenarios. Among the files were: (1) a highly obfuscated (89.7%) malicious application[2]; (2) a very popular social application with no obfuscation and a large number of packages; (3) a popular mobile Internet browser with 100% obfuscated packages; (4) an application

[1] `https://github.com/ahupp/python-magic`
[2] Detected by the antivirus software on the machine where the download occurred.

Table 1. Obfuscation ratio. The row with # marks the absolute number of applications with obfuscated number of classes in the given range. The row with % marks the percentage this number represents in the set of the total applications.

OBF	100%	(100 − 80]	(80 − 60]	(60 − 40]	(40 − 20]	(20 − 0)	0%	*Total*
#	82	291	196	166	283	423	250	**1691**
%	4.85	17.21	11.59	9.82	16.74	25.01	14.78	**100**%

Table 2. Profiling the set of applications according to the given criteria: OBF (total obfuscated classes), B64 (apps containing base64 strings), NAT (native code), DYN (dynamic code), REF (reflection), CRY (crypto code), HEAD (apps with header size of `0x70`), LIT (apps with little endian byte ordering). The row with # marks the absolute numbers of occurrences, % marks the percentage this number represents in the set of the total applications.

	OBF	B64	NAT	DYN	REF	CRY	HEAD	LIT
#	41.839	840	629	224	1519	1236	1691	1691
%	46.74	49.68	37.20	13.25	89.83	73.09	100	100

Table 3. Classification of the base64 encoded strings. Categories are denoted as follows: TXT for text, MUL for multimedia, OTH for other.

	# files	%total	DATA TYPE			
unknown	1156	48.59	non-identified data			
known	1241	51.41	type	#	%	category
			ASCII text	56	4.51	TXT
			GIF	48	3.87	MUL
			HTML	3	0.24	OTH
			ISO-8859 text	1	0.08	TXT
			JPEG	33	2.66	MUL
			Non-ISO extended-ASCII text	24	1.93	TXT
			PNG	522	42.06	MUL
			TrueType font text	548	44.17	MUL
			UTF-8 Unicode text	1	0.08	TXT
			XML document	2	0.16	OTH

which `androguard` (DAD) and `dexter` failed to process; (5) an application which is known to use strings encryption and is claimed to be obfuscated as well; (6) an application containing many base64 encoded strings; (7-10) four other applications chosen at random.

With the exception of application (4) all files were successfully processed by the `androguard` analysis tool. The source code of all checked obfuscated methods was successfully recovered to a correct Java code with the `androguard` plugin for Sublime Text[3]. The control-flow graphs of *all* analyzed files was

[3] `http://www.sublimetext.com/`

recovered successfully with `androgexf.py`. However, in some applications the excessive number of packages created an inappropriate setting for adequate analysis thus the graphs were filtered by pattern-matching the labels of their nodes. Having the graphs of all applications simplified revealed practices such as implementation of custom strings encryption-decryption pair functions and having their source code implementation hidden in a native library (seen in two of the analyzed files). Reviewing the graph of application (4) was a key towards understanding why some tools break during analysis: they simply do not handle cases of Unicode method or field names (e.g. 文章:`Ljava/util/ArrayList;`). On the other hand, `baksmali` did fully recover the mnemonics of the application, Unicode names representing no obstacle.

Regarding the permissions used in the malicious application, it is no surprise that it required the `INTERNET` permission. In fact, being at-first-sight a wallpaper application, it had as much as 27 permissions including install privileges, writing to the phone's external storage and others. This result only comes as confirmation to what previous studies have already established as user privacy invasive practices [20].

3.3 Study Conclusion

The main conclusion of both automated and manual inspection is that even in cases where some tools hindered recovering the bytecode mnemonics or source code, there is a way round to obtain relevant information. Where a given tool is not useful, another can be used as complement. Reversing large applications may be slowed down due to the complexity of the program graph, but with appropriate node filtering a reasonable subgraph can be obtained for analysis. To prevent information extraction by static analysis some applications made use of Java reflection or embedding valuable code in a native library. Apart from using ProGuard to rename components and decrease program understandability, no other code obfuscation was found. Using Unicode names for classes and methods could be regarded as an analogical type of modification: it merely affects program layout not the control flow.

3.4 Remarks

A number of considerations need to be taken into account when reviewing the results of the performed study. (1) All applications studied were available through the official Google Play market as of March 2013. (2) Only freely available applications were processed: the results will highly likely differ if identical examinations were performed on payed applications. (3) The set of popular applications in the Google Play market differs with the country of origin of the requesting IP address: the download for this study was executed on a machine located in Bulgaria.

4 Implementing a Dalvik Bytecode Obfuscator

This section suggests an implementation of a Dalvik bytecode obfuscator including four transformations whose main design accents fall on fulfilling the generic and cheap properties.

In the context of this work the term "generic" denotes that the transformations are constructed in aspiration to encompass a large set of applications without preliminary assumptions which must hold for the processed file. On Android this can be a challenge since an application has to run on a wide range of devices, OS versions and architectures. Thus, it is crucial that any applied code protection would not decrease the set of application running devices. When a transformation is characterized as "cheap", this is in referral to previously published work by Collberg et. al. on classifying obfuscating transformations [4]. By definition, a technique is cheap if the obfuscated program $\mathcal{P}'$ requires $\mathcal{O}(n)$ more resources than executing the original $\mathcal{P}$. Resources encompass processing time and memory usage: two essential performance considerations, especially for mobile devices. Following is a description of the general structure of the Dalvik bytecode obfuscator[4] as well as details on the four transformations applied.

4.1 Structure Overview

The input of the tool is an APK file which can be either processed by ProGuard i.e. with renamed classes and methods, or not modified at all. Auxiliary tools used during the obfuscation are the pair `smali` assembly and `baksmali` disassembly. The application is initially disassembled with `baksmali` which results in having a directory of `.smali` files. These files contain mnemonics retrieved from the immediate bytecode interpretation. Three of the transformations parse, modify the mnemonics and assemble them back to a valid `.dex` file using `smali`. One transformation modifies the bytecode of the `.dex` file directly. After the modifications have been applied, the `.dex` file is packed together with the resource files, signed and is verified for data integrity. This last step yields a semantically equivalent obfuscated version of the APK file. Figure 1 summarizes the entire obfuscation process.

Adopting this workflow has the advantage of accelerating the development process by relying on a `.dex` file assembler and disassembler pair. However, a disadvantage is that the implemented obfuscator is bound by the limitations of the used external tools.

4.2 Transformations

The tool can apply four transformations designed such that all of them affect both the data and control flow. The transformations targets are: calls to native libraries, hardcoded strings, 4-bit and 16-bit numeric constants. Native

[4] `https://github.com/alex-ko/innocent`

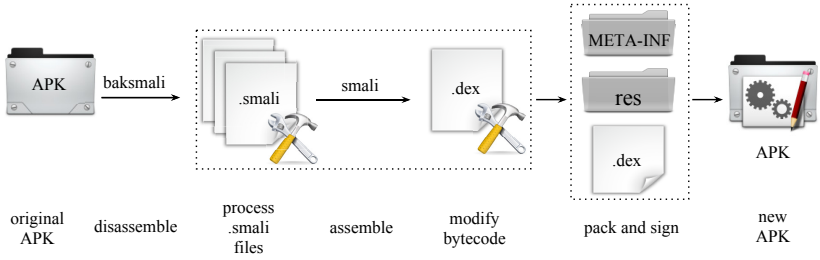

Fig. 1. Workflow of the obfuscator

calls are redirected through external classes in methods that we call here "wrappers". Strings are encrypted and numeric constants are packed in external class-containers, shuffled and modified. The fourth modification injects dead code which has a minor effect on the control flow, but makes the input APK resistant to reverse engineering with current versions of some popular tools which is why it is called "bad" bytecode injection.

Adding Native Call Wrappers. While native code itself is not visible through applying static analysis, calls to native libraries cannot be shrunk by tools such as ProGuard. The reason is that method names in Dalvik bytecode must correspond exactly to the ones declared in the external library for them to be located and executed. This transformation does not address the issue with comprehensive method names since this depends on the developer. However, another source of useful information is the locality of the native calls i.e. by tracking which classes call particular methods metadata information for the app can be obtained. To harden the usage tracking process one could place the native call in a supplementary function, what is referred here as a native call wrapper. The exact sequence of steps taken is on the following schematic figure:

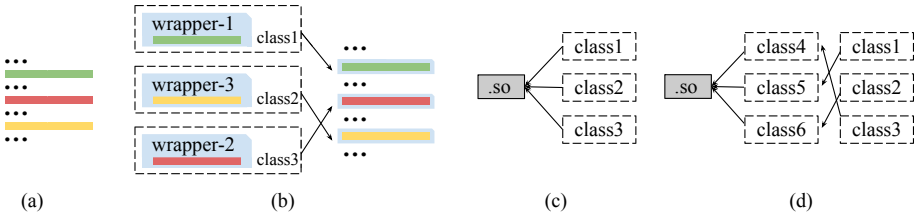

Fig. 2. Adding native call wrappers

Let us have a class containing three native calls which are highlighted on (a). For each unique native method a corresponding wrapper with additional arguments is constructed redirecting the native call. To complicate the control flow, the wrappers are scattered randomly in external classes from those located originally. As a final step each native call is replaced with an invocation of its respective wrapper as shown in (b).

The impact of this transformation on the call graph can be seen as a transition from what is depicted in (c) to the final result in (d). Initially, the locality of the native method calls give a hint on what the containing class is doing. After applying the transformation once, the reversing time and effort is increased by locating the wrapper and concluding that there is no connection between the class containing the wrapper and the native invocation. If the transformation is applied more than once, the entire nesting of wrappers has to be resolved. Usually, a mobile application would have hundreds of classes to scatter the nested wrapping structures: a setting that slows down the reversing process.

Packing Numeric Variables. The idea behind this transformation stems from what is known in literature as opaque data structures [23]. The basic concept is to affect data flow in the program by encapsulating heterogeneous data types in a custom defined structure.

The target data of this particular implementation are the numeric constants in the application. The bytecode mnemonics are primarily scanned to locate the usages and values of all 4-bit and 16-bit constants. After gathering those, the obfuscator packs them in a 16-bit array (the 4-bit constants being shifted) in a newly-created external class as shown on (a) in the schematic figure below. Let us call this external class a "packer". The numeric array in the packer is then processed according to the following steps. Firstly, to use as little additional memory as possible, all duplicated numeric values are removed. Next, the constants are shuffled randomly and are transformed in order to hide their actual values. Currently, three transformations are implemented: XOR-ing with one random value, XOR-ing twice with two different random values and a linear mapping. Then, a method stub to get the constant and reverse the applied transformation is implemented in the packer. Finally, each occurrence of a constant declaration is replaced with an invocation to the get-constant packer method.

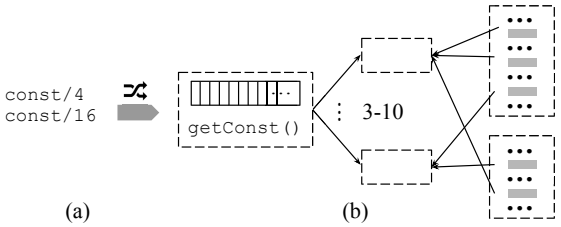

Fig. 3. Packing numeric variable constants

The transformation thus put represents not much of added complexity to the program. To further challenge the reverser, the packer class creates between 3 and 10 replicas of itself, each time applying anew the shuffling and the selection of the numeric transformation to the array. This means that even if the obfuscated application has several packer classes which apply, for example, the XOR-twice transformation, in each of them the two random numbers will differ as well as the data array index of every unique numeric value. Designed like this, the transformation has the disadvantage of data duplication. However, an advantage

that is possible due to this reduplication is removing the necessity that a single class containing constants is calling the get-constant method of the same packer which is shown on (b) in the figure above.

To summarize, control flow is complicated by multiple factors. Firstly, additional classes are introduced to the application i.e. more data processing paths in the program graph for the reverser to track. Then, in each packer class the array constant values will diverge. Lastly, different packers are addressed to retrieve the numeric constants in a single class and the reverser would have to establish that the connection between each of the different packer calls is merely data duplication.

Strings Encryption. The decision to include this transformation in the tool is motivated by the fact that none of the here cited open-source tools implements strings encryption at the moment of submission. Moreover, the transformation is designed in such a way that it aspires to add more control flow complexity than what is currently found to be implemented [11] and instead of using a custom algorithm (usually simply XOR-ing with one value) the strings here are encrypted with the RC4 stream cipher [24].

The figure on the right gives an overview to how the transformation works. The classes containing strings are filtered out. A unique key is generated for and stored inside each such class. All strings in a class are encrypted with the same class-assigned-key. Encryption yields a byte sequence corresponding to each unique string which is stored as a data array in a private static class field. This results in removing strings from the constant pool upon application re-assembly thus preventing from visibility with static analysis. A consideration to use static class fields for storing the encrypted strings is the relatively small performance impact. Decryption occurs during runtime, the strings being decoded once upon the first invocation of the containing class. Whenever a given string is needed, it is retrieved from the relevant class field.

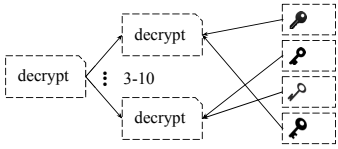

Fig. 4. Strings encryption

Analogically to previous transformations, adding control flow complexity is at the cost of duplication. The decryption class creates between 3 and 10 semantically equivalent replicas of itself in the processed application as shown in the figure. Each class containing strings chooses randomly its corresponding decryption class.

To summarize, there are several minor improvements of the here suggested implementation over what was found in related works. Encrypting the strings in each class with a unique key slows down automatic decryption because the keys are placed at different positions and need to be located separately for each class. Designing the transformation by using a decryption-template approach allows in principal the developer to modify this template: they can either choose to strengthen potency and resilience or change easily the underlying encryption/decryption algorithm pair. Finally, the added control flow complexity is increased by the supplementary decryption classes.

Injecting "Bad" Code. The proposed here transformation has as main purpose to defy popular static analysis tools without claiming to be highly resilient. In fact, it is shown that a simple combination of known exploits is enough to cause certain tools to crack and produce an output error. There are two defeat target tool types: decompilers and disassemblers performing static analysis. The used techniques are classified in previous works as "preventive" [4] for exploiting weaknesses of current analysis tools.

To thwart decompilers an advantage is taken from the discrepancy between what is representable as legitimate Java code and its translation into Dalvik bytecode. Similar techniques have been proposed for Java bytecode protection [25]. The Java programming language does not implement a `goto` statement, yet when loop or switch structures are translated into bytecode this is done with a `goto` Dalvik instruction. Thus by working directly on bytecode it is possible to inject verifiable sequences composed of `goto` statements which either cannot be processed by the decompilers or do not translate back to correct Java source code.

To thwart disassemblers several "bad" instructions are injected directly in the bytecode. Execution of the bad code is avoided by a preceding opaque predicate which redirects the execution to the correct paths. This technique has already been shown to be successful [26]. However, since its publishing new tools have appeared and others have been fixed. The here suggested minor modifications are to include in the dead code branch: (1) an illegal invocation to the first entry in the application methods table; (2) a packed switch table with large indexes for its size; (3) a call to the bogus method we previously created such that it looks as if it is being used (not to be removed as dead code).

4.3 Evaluation of the Modified Bytecode

To verify the efficiency of the developed tool a set of 12 test applications was selected among the huge variety. The apps profiling is given in Appendix A. The performance tests of the modified applications were executed on two mobile devices: (1) HTC Desire smartphone with a customized Cyanogenmod v7.2 ROM, Android v2.3.7; (2) Sony Xperia tablet with the original vendor firmware, Android v4.1.1.

During the development process all transformations were tested and verified to work separately. On table 4 are given the results of their combined application. The plus sign should be interpreted as that the transformations have been applied consequently (e.g. `w+o+p` means applying adding wrappers then obfuscating strings then packing variables).

With the exception of the bad code injection on the facebook application, every application undergoing the possible combinations of transformations was installed successfully on both test devices. An observation on the error console logs for the facebook application suggests that the custom class loader of this app conflicts with the injected bad code [27]. The rest of the transformations did not make the app crash. For the Korean ebay app no crash occurred, but not all of the UTF-8 strings were decrypted successfully i.e. some messages which should

Table 4. Testing the obfuscated applications on HTC Desire and Sony Xperia tablet. The transformations abbreviations are as follows: `w` adding native wrappers, `o` obfuscating strings, `p` packing variables, `b` adding bad bytecode. The black bullet indicates successful install and run after applying the series of transformations.

APP	w	w+o	w+o+p	w+o+p+b
com.adobe.reader.apk	●	●	●	●
com.alensw.PicFolder.apk	●	●	●	●
com.disney.WMPLite.apk	●	●	●	●
com.ebay.kr.gmarket.apk	●	●	●	●
com.facebook.katana.apk	●	●	●	○
com.microsoft.office.lync.apk	●	●	●	●
com.rebelvox.voxer.apk	●	●	●	●
com.skype.android.access.apk	●	●	●	●
com.teamlava.bubble.apk	●	●	●	●
cz.aponia.bor3.czsk.apk	●	●	●	●
org.mozilla.firefox.apk	●	●	●	●
snt.luxtraffic.apk	●	●	●	●

have been in Korean appeared as their UTF-8 equivalent bytes sequence. The most probable reason is that large alphabets are separated in different Unicode ranges and smali implements a custom UTF-8 encoding/decoding[5] which might have a slight discrepancy with the encoding of python for some ranges. Finally, the Voxer communication app did not initially run with the injected bad code. This lead to implementing the possibility to toggle the verification upon bytecode injection. By setting a constant in the method as verified its install-time verification can be suppressed. Enabling this feature let the Voxer app run without problems. However, verifier suppression is disabled by default for security considerations.

Besides the upper mentioned, no other anomalies were noted on the tested applications. No noticeable runtime performance slowdown was detected while testing manually. The memory overhead added by each transformation separately is shown on Appendix B. Because the applications differ significantly in size, for a better visual representation only the impact on the least significant megabyte is shown.

Finally, some of the popular reverse engineering tools were tested against the modified bytecode. Two possible outcomes were observed: either the tool was defeated i.e. did not process the app at all due to a crash, or the analysis output is erroneous. Tools which crashed were: `baksmali`, `apktool`, `DARE` decompiler, `dedexer`, `dex2jar`. Tools with erroneous output were: `androguard`, `JD-GUI`.

4.4 Limitations

To be effective, the transformations had to comply with the Dalvik verifier and optimizer [28]. Moreover, the workflow used by the obfuscator relies on external

[5] `https://code.google.com/p/smali/source/browse/dexlib/src/main/java/org/jf/dexlib/Util/Utf8Utils.java`

tools which imply their own constraints. Hence, it is worth noting the limitations of the proposed transformations.

Native Call Wrappers is applied only to native methods which have no more than 15 registers. The reason is that `smali` has its own register implementation distinguishing between parameter and non-parameter registers and is working only by representing methods with no more than 15 non-parameter registers. In case more registers need to be allocated, the method is defined with a register range, not a register number. Defined so to ease the editing of smali code, this has its restrictions on our transformation.

Packing Numeric Variables is applied only to the 4-bit and 16-bit registers, because there is a risk of overflowing due to the applied transformation when extended to lager registers. Clearly, a transformation shifts the range of the possible input values. Regarding the simple XOR-based modifications, the scope is preserved but a linear mapping shrinks the interval of possible values. Also, packing variables was restricted only to numeric constant types because in Dalvik registers have associated types i.e. packing heterogeneous data together might be a type-conversion potentially dangerous operation [29].

5 Conclusion

This work accented on several important aspects of code obfuscation for the Android mobile platform. To commence, we confirmed the statement that currently reverse engineering is a lightweight task regarding the invested time and computational resources. More than 1600 applications were studied for possible applied code transformations, but found no more sophisticated protection than variable name scrambling or its slightly more resilient variation of giving Unicode names to classes and methods. Some applications used strings encryption during runtime. Yet, these applications themselves had hardcoded strings visible with analysis tools.

Having demonstrated the feasibility of examining randomly selected applications, a proof of concept open-source Dalvik obfuscator was proposed. Its main purpose is introducing a reasonable slowdown in the reversing process. The obfuscator performs four transformations all of which target both data flow and control flow. Various analysis tools were challenged on the modified bytecode, showing that the majority of them are defeated.

Android is merely since five years on the market, yet because of its commercial growth much research is conducted on it. The evolution of the platform is a constantly ongoing process. It can be seen in its source code that some of the now unused bytecode instructions were former implemented test instructions. Possible future opcode changes may invalidate the effects our transformations. Moreover, analysis tools will keep on getting better and to defeat them newer, craftier obfuscation techniques will need to be applied. This outwitting competition between code protectors and code reverse engineers exists ever since the topic of obfuscation has been established of practical importance. So far, evidence proves this game will be played continuously.

Acknowledgements. This paper is a derivation of the author's thesis work which was supervised by Prof. Alex Biryukov and Dr. Ralf-Philipp Weinmann while in candidature for a master degree at the University of Luxembourg. The author wishes to thank them for their guidance, especially Dr. Ralf-Philipp Weinmann for his valuable advice and support.

References

1. Gartner News (February 2013, press release), http://www.gartner.com/newsroom/id/2335616
2. Kaspersky Lab: 99% of all mobile threats target Android devices, http://www.kaspersky.com/about/news/virus/2013/99_of_all_mobile_threats_target_Android_devices
3. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory IT-22(6), 644–654 (1976)
4. Collberg, C., Thomborson, C., Low, D.: A Taxonomy of Obfuscating Transformations, Technical Report 148, Department of Computer Science, University of Auckland, New Zealand (1997)
5. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S.P., Yang, K.: On the (Im)possibility of Obfuscating Programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001)
6. Androguard project home page, https://code.google.com/p/androguard/
7. Smali/Baksmali project home page, https://code.google.com/p/smali/
8. Dedexer project home page, http://dedexer.sourceforge.net/
9. Dex2jar project home Page, https://code.google.com/p/dex2jar/
10. Dexdump, Android SDK Tools, http://developer.android.com/tools/help/index.html
11. Bremer, J.: Automated Deobfuscation of Android Applications, http://jbremer.org/automated-deobfuscation-of-android-applications/
12. Dexter project home page, http://dexter.dexlabs.org/
13. Radare2 project Home Page, http://radare.org/y/?p=download
14. Collberg, C., Thomborson, C., Low, D.: Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs (1998)
15. Schulz, P.: Dalvik-obfuscator project GitHub page, https://github.com/thuxnder/dalvik-obfuscator
16. Strazzere, T.: APKfuscator project GitHub page, https://github.com/strazzere/APKfuscator
17. DexGuard main page, http://www.saikoa.com/dexguard
18. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android Permissions Demystified. University of California, Berkeley (2011)
19. Gommerstadt, H., Long, D.: Android Application Security: A Thorough Model and Two Case Studies: K9 and Talking Cat. Harvard University (2012)
20. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (2011)
21. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: Proceedings of the 20th USENIX Security Symposium (2011)
22. Strazzere, T.: Dex Education: Practicing Safe Dex, Blackhat, USA (2012)

23. Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection (2009) ISBN-13: 978-0321549259
24. Cypherpunks (mailing list archives), RC4 Source Code, http://cypherpunks.venona.com/archive/1994/09/msg00304.html
25. Batchelder, M.R.: Java Bytecode Obfuscation, Master Thesis, McGill University School of Computer Science, Montréal (2007)
26. Schulz, P.: Dalvik Bytecode Obfuscation on Android (2012), http://www.dexlabs.org/blog/bytecode-obfuscation
27. Reiss, D.: Under the Hood: Dalvik patch for Facebook for Android (2013), http://www.facebook.com/notes/facebook-engineering/under-the-hood-dalvik-patch-for-facebook-for-android/10151345597798920
28. Android Developers Website, http://developer.android.com/index.html
29. Bornstein, D.: Dalvik VM Internals (2008), https://sites.google.com/site/io/dalvik-vm-internals

Appendix

A Profiles of the Applications Selected for Testing

Table 5. Profiles of the test applications. The label abbreviations are identical to those in the case study of applications. The black bullet marks a presence of the criteria. The label MISC stands for "miscellaneous" and indicates notable app features. In the facebook app, CCL stands for the custom class loader.

APP	OBF	NAT	DYN	REF	CRY	MISC
com.adobe.reader.apk	0%	•	○	•	•	SD card
com.alensw.PicFolder.apk	100%	•	○	•	○	camera
com.disney.WMPLite.apk	5%	•	○	•	•	graphics
com.ebay.kr.gmarket.apk	0%	•	○	•	•	UTF-8 text
com.facebook.katana.apk	84%	•	•	•	•	CCL
com.microsoft.office.lync.apk	0%	•	○	•	•	phone calls
com.rebelvox.voxer.apk	0%	•	○	•	•	audio, SMS
com.skype.android.access.apk	0%	•	○	•	○	audio, video
com.teamlava.bubble.apk	0%	•	○	•	•	graphics
cz.aponia.bor3.czsk.apk	0%	•	○	•	○	GPS, maps
org.mozilla.firefox.apk	0%	•	•	•	•	internet
snt.luxtraffic.apk	0%	○	○	○	○	GPS, maps

B Memory Overhead Results

Table 6. Measuring the memory overhead of the transformations