PANDORA Applies Non-Deterministic Obfuscation Randomly to Android

Mykola Protsenko, Tilo Müller
Friedrich-Alexander University Erlangen-Nuremberg, Germany
mykola.protsenko@fau.de, tilo.mueller@cs.fau.de

Abstract

Android, a Linux-based operating system, is currently the most popular platform for mobile devices like smartphones and tablets. Recently, two closely related security threats have become a major concern of the research community: software piracy and malware. This paper studies the capabilities of code obfuscation for the purposes of plagiarized software and malware diversification. Within the scope of this work, the PANDORA (PANDORA Applies Non-Deterministic Obfuscation Randomly to Android) transformation system for Android bytecode was designed and implemented, combining techniques for data and object-oriented design obfuscation. Our evaluation results indicate deficiencies of the malware detection engines currently used in 46 popular antivirus products, which in most cases were not able to detect samples obfuscated with PANDORA. Furthermore, this paper reveals shortcomings of the Androsim tool and potentially other static software similarity algorithms, recently proposed to address the piracy problem in Android.

1 Introduction

As indicated by recent studies, the smartphone market experienced explosive growth in recent years. From 2010 to 2011 sales soared up 42% [13], in the interval 2011 to 2012 the increase reached even 47% [14]. Among all smartphone platforms, Android keeps dominating with 72% share of the sales [14].

Obviously, the high popularity and market penetration could not leave hackers, malware developers, and pirates indifferent. Within the last few years, the theft of Android applications and malware distribution have reached extensive dimensions [40, 39]. This paper addresses the closely related problems and aims to study the capabilities of malware developers and software pirates. For this purpose, an Android bytecode obfuscation system, called PANDORA (PANDORA Applies Non-Deterministic Obfuscation Randomly to Android), was implemented on top of the Soot framework [33]. PANDORA contains both obfuscating transformations of the program’s object-oriented design and data obfuscation techniques. As the name suggests, it performs all transformations as randomly as possible, being able to produce many different obfuscated versions of a program. Moreover, it operates on the bytecode level, which facilitates its application with malicious intents to conceal malware or application piracy.

The results of our practical evaluation have shown high effectiveness of the PANDORA system utilized to prevent detection of known malware samples and recognition of plagiarized software. This indicates weaknesses of the current detection engines and static software similarity algorithms. At the same time, it witnesses good diversification quality of the transformations we implemented.

The rest of this paper is organized as follows: Section 2 provides background information and overviews related publications. Section 3 describes the design of the PANDORA transformation system, which is then evaluated in Section 4. We conclude in Section 5.

2 Background and Related Work

This section gives some background information on the topics closely related to the scope of this paper and provides an overview of important related research publications.

2.1 Android Platform

Android is an open source Linux-based operating system designed to run on mobile devices, like smartphones, tablets and netbooks. Among the reasons of its popularity, the ease of Android software development and distribution certainly takes not the last place.

The source code of Android applications is written in the Java language and converted into Dalvik bytecode. The corresponding virtual machine (Dalvik VM) is register-based and is designed especially to meet the needs of mobile platforms, restricted in memory and computation power. The usual way of application distribution is given by online download centers also known as application markets, with Google
we are interested in software similarity algorithms in the
were borrowed from Majumdar, Drape, and Thomborson
Diversity can be considered a branch of the obfuscation meth-
verse engineering, i.e., the comprehension and modification
ject to intensive research over the past few years. A short
2.4 Android Security Threats

Another evaluation of the antivirus products for the An-
droid platform was performed recently by Rastogi, Chen, and
Jiang [25] using their DroidChameleon framework. Their
study is based on a similar transformation set as ours and
concludes similar results. One of the main differences lies in
our conscious decision to leave out dynamic transformations
such as code encryption, as the one having very low stealth
and resilience against dynamic analysis [6]. Also PANDORA
has a slightly richer static transformation set. Furthermore,
our study includes evaluation of the transformations for the goal of plagiarism detection mitigation.

2.4.2 Piracy

Android application piracy has become a major concern in the developer and research communities over the past few years [39, 23, 18, 8, 4, 17]. Recent research by Zhou et al. [39] indicates that 5-13% of the applications in third-party markets are repackaged from the Google Play Market, and are therefore most probably stolen. Furthermore, as previously mentioned, repackaging constitutes the main way of malware spread [40]. To address this issue, Potharaju et al. [23] proposed to enhance markets with a plagiarism detection system which rejects applications recognized as repackaged. The proposed system utilizes a software similarity algorithm based on the restricted representation of a method’s syntactic structure, which involves only method invocation instructions.

Zhou et al. [39] presented a prototype plagiarism detection tool called DroidMOSS, which was used in their study of repackaged Android applications. This tool utilizes fingerprints computed from a sequence of opcodes using the *fuzzy hashing* approach [16].

As a part of the Androguard toolbox, Desnos and Gueguen [8] presented the Androsim application similarity measurement tool. The underlying algorithms utilize fingerprints based on the instruction sequences, properties of the methods’ CFGs, API calls, strings, and exceptions. As the only publicly available plagiarism detection tool for the Android platform, this tool was used in our evaluation described in Section 4.

3 The PANDORA System

We describe the design and key implementation principles of the PANDORA transformation system. In order to facilitate analysis and manipulation of Android applications, our system was designed on top of the Soot framework [33], which provides all necessary functionality regarding analysis, modification, and generation of Java bytecode. Retargeting from Dalvik to Java bytecode and back again is performed using the *Dex2Jar* tool. A direct support of the Dalvik bytecode is planned for inclusion in the next release of Soot [2].

The current version of the PANDORA system contains 7 data obfuscation techniques and 6 transformations aiming to obscure the object-oriented design of a program. Furthermore, name obfuscation of classes, methods and fields based on the Java Bytecode Obfuscator (JBCO) tool [1] is included.

One of the main overall design goals of our proposed transformation system is to maximize the diversification impact of the implemented transformations by applying them as *random* as possible to introduce *non-determinism* in the obfuscation process. That is, the parts of the program’s code to be transformed as well as the parameters of the obfuscation techniques are selected randomly.

Since a detailed description of the transformations and their implementation would blow up the scope of this paper, the following subsections provide a bird’s eye perspective of the PANDORA transformations.

3.1 Data Obfuscation

The goal of the transformations falling into this category is to obscure the usage of variables, and therefore operations on basic data types and objects. Currently implemented techniques are described below.

3.1.1 String Encryption

String constants constitute an important source of information for reverse engineers [6] and are often utilized by software similarity measurement tools. The current version of the PANDORA system provides two simple *polyalphabetic* string encryption algorithms, i.e., algorithms substituting each string character with a varying another character [3]. Both the key and the encryption method are randomly chosen in PANDORA.

**Vigenère Encryption** The first implemented encryption method is Vigenère [32], an improvement of the Caesar cipher [3]. It uses another string as a key, which periodically determines the shift for each symbol of the text being encrypted.

**Dynamic Key Encryption** The second encryption algorithm used in our work is denoted as the *Dynamic Key Encryption*. The shift of plaintext characters is defined by a key sequence $k_i$, a pseudorandom number sequence generated using the *Linear Congruential Method* [19]. The generation of the key sequence is described by

$$k_{i+1} = (a \cdot k_i + b) \mod m$$

with $a$ denoting the *multiplier*, $b$ being the *increment*, $m$ being the *modulo*, $k_{i+1}$ and $k_i$ denoting the $(i+1)$-th and the $i$-th key of the sequence, respectively.

3.1.2 Integer Encoding

The two integer encodings implemented in PANDORA are able to perform operations of addition, subtraction, multiplication, and comparison in a homomorphic manner, i.e., the decoding into unobfuscated form is not required. However, one operation in the cleartext representation corresponds to a few operations in the obfuscated representation. For the
non-supported operations, e.g., division, the original value has to be restored prior and encoded again after the operation. Note that both of our encodings do not preserve the overflow behavior of a variable.

**Split Modulo** The first integer obfuscation splits a variable in two parts: a quotient and a rest. This representation is a slight generalization of the one proposed by Majumdar, Drape, and Thomborson [22]. For a given modulo value \( m > 1 \), an integer variable \( x \) is split into the variables \( a \) and \( b \) as follows:

\[
a = \frac{x}{m}, \quad b = x \mod m
\]

where “/” denotes integer division. The original value can be obtained from \( a \) and \( b \) as:

\[
x = a \cdot m + b
\]

**Linear Encoding** The second integer encoding, presented by Collberg, Thomborson, and Low [7], applies a linear transformation to a variable. Given an integer variable \( x \), a multiplier parameter \( \alpha > 0 \) and an increment parameter \( \beta \), the linear transformation is performed as:

\[
z = \alpha \cdot x + \beta
\]

with \( z \) being the obfuscated form of value \( x \). To avoid overflows, \( z \) should be stored in the variable of the next size integer type, i.e., \( \text{long} \) for \( \text{int} \) type of \( x \). The original value can be restored as

\[
x = (z - \beta)/\alpha
\]

### 3.1.3 Array Index Shift

To slightly obscure the usage of arrays in a program and to make the order of stored elements appear random, the access to an array’s elements can be modified by cyclic shifts of indices. For a given constant shift value \( s > 0 \), the access to the \( i \)-th element of an array \( A \) of the length \( l \) is replaced by

\[
A[i] \Rightarrow A[(i + s) \mod l]
\]

### 3.1.4 Locals Composition

A further method to modify the usage of local variables implemented in PANDORA is their composition to container objects. This makes a group of variables appear semantically related.

To perform this composition, a set of local variables of the same type is selected. For this set, a container is created for storing the values of the variables. After the definition, the value of a variable is saved in the container and can be extracted from there before it is used again. Currently, two types of containers are available: arrays and maps. For a composition, each variable of the group is assigned a unique key at obfuscation time. In case of an array, a key corresponds to the array index. In case of a map, a randomly generated key can be of type string, character, or integer.

### 3.2 OOD Obfuscation

The Object-Oriented Design (OOD) obfuscation methods constitute the second class of transformations implemented in the PANDORA system. The goal of these transformations is to modify and distort the original design and structure of applications by affecting their object-oriented building blocks such as fields, methods, and classes. Because of their nature, such transformations are referred to as false refactorings [7].

#### 3.2.1 Drop Modifiers

The first and probably the simplest OOD obfuscation transformation is Drop Modifiers. It discards access restrictions on application classes and their members, i.e., it makes fields, methods, and classes public and removes the final modifier. Applied on its own, this transformation has low potency. However, it opens the door for numerous other transformations described below by permitting various actions on objects.

#### 3.2.2 Extract Method

The Extract Method transformation was designed in order to unleash more powerful techniques upon additional pieces of program code. It implements a simplified version of the outline method refactoring [12] by outlining the complete body of a given method. It creates a new method with the same parameters and the same body as the given one, and replaces the body of the original method with an invocation statement for the newly created method, passing all parameters to it.

This transformation is applied on methods with signatures (method names and parameter types list) that cannot be changed, like onClick methods of the GUI elements or onCreate methods of Android Activity classes. Such extractions make the bodies of methods eligible for further transformations like Move Methods and Merge Methods, which are described below.

#### 3.2.3 Encapsulate Field

Another rather auxiliary transformation is Encapsulate Field [12]. For a given field, it creates a setter and a getter method, i.e., methods that take over the definition of the field’s value and its extraction, respectively. Then, some of the direct accesses to the encapsulated field are replaced with corresponding methods.
The goal of this transformation is to slightly camouflage field usage. Additionally, it creates new methods in the declaring class. These new methods become a target of other obfuscation techniques (see below), which further increase the complexity of the field access.

3.2.4 Move Field

The Move Field refactoring moves a field from one application class to another [12]. For static fields this transformation is straightforward: all references of the old field have to be replaced with references to the new one. Moving instance fields, however, is a more challenging task. Each access to the instance field requires a base object, hence, after a field is moved, a corresponding reference to an object of the new hosting class must be created in the old one.

3.2.5 Move Method

The Move Method transformation moves a method from one application class to another [12]. Analogously to the Move Field operation, moving static methods is performed easily by replacing the corresponding invocation statements, whereas instance methods require base objects. Furthermore, if the moved method makes use of the implicit this parameter, this must be added explicitly to the parameter list and passed on each invocation.

3.2.6 Merge Methods

A further method obfuscation transformation is Merge Methods, which is performed as follows. If two methods declared in the same class have equal return types and same access modifiers, their bodies can be interleaved, and their invocations can be replaced with calls to the merged method [7]. Interleaving is achieved by merging the parameter lists, local variables, and bodies, i.e., code instructions. The selection of a body to be executed is performed using the new key parameter added to the merged method’s parameter list.

4 Evaluation

The practical evaluation of the PANDORA transformation system was performed within two test scenarios: malware diversification and pirated application diversification.

During all current evaluation tests, the whole transformation system was configured to achieve maximum performance in terms of diversity and complexity, without considering or trying to limit possible increase of execution time, memory consumption, and application size.

4.1 Transformation Levels

To study the impact of different transformation types, we define multiple obfuscation levels:

- **Level 0 (L0)** The L0 transformation level denotes plain identifier renaming (including renaming of the corresponding Manifest entries) as the only obfuscation technique applied.
- **Level 1 (L1)** Identifier renaming combined with all data obfuscation transformations described in Section 3.1 are denoted as the L1 level.
- **Level 2 (L2)** The L2 transformation level includes all techniques of the OOD obfuscation described in Section 3.2 in combination with identifier renaming.
- **Level 3 (L3)** The L3 level is the obfuscation configuration which is constituted by all name, data, and OOD obfuscations available in PANDORA.

4.1.1 Malicious Code Injection

Another transformation level, also denoted as L3-Inj, corresponds to the injection of malicious code into a benign application, followed with the level L3 obfuscation of the merged code. The motivation behind this level is given by the assumption that the overall effect of the performed obfuscations depends on the amount of the application’s code. Within our evaluation tests, the OI Notepad application was used to be merged with malware samples.

4.1.2 Rebuilding Malware

An additional technique, used to evaluate the detection performance of antivirus products, is application rebuilding. In this context, rebuilding refers to decoding an application with the Apktool and encoding it back. A rebuilt application is then signed using the Dex2Jar tool. Although rebuilding does not affect contents of the apk file like resources and Manifest, it can slightly modify the bytecode order [38], and it certainly changes the hash of an application. Therefore, this technique is utilized to sort out antivirus products whose detection algorithms are based solely on the hash signature of submitted files.

4.2 Malware Diversification

For the malware diversification scenario, we have selected 20 known samples from different malware families provided by Spreitzenbarth et al. [29]. Each of the samples was submitted to the VirusTotal website [34] in its original state and after its transformation according to the levels L0 to L3 described above. Our evaluation took place in February 2013.
Note that since submissions were made in the time over one week, we cannot completely exclude the possibility of changes in the detection engines or signature databases.

Through the rest of this paper, the detection rate of a sample denotes the number of antivirus products classifying it as malicious, varying from 0 to 46. At the same time, the detection rate of an antivirus program refers to the number of samples it was able to detect, lying between 0 and 20.

The malware detection rates presented above were considered independent from the individual detection performance of single antivirus programs. Next, the evaluation results are discussed from the antivirus-centric point of view.

Table 2 summarizes the detection rates that are achieved on the top-10 of our tested antivirus products. These antivirus programs were selected according to their detection rates on the L3-Inj transformation level, i.e., the software included in the table has shown the best performance on the L3-Inj sample set among other products.

These results show that even the best antivirus software was able to detect only 6 out of 20 samples transformed at PANDORA’s obfuscation level L3 or L3-Inj. Comparing our results with those obtained by Zheng et al. [38] and their ADAM system, we want to emphasize two points. First, more than a year after the ADAM publication, simple transformations like rebuilding and renaming are still able to defeat most malware detection products. Second, no antivirus program can show a sufficiently good performance on the samples after the strongest transformation level of PANDORA.
Summarizing the results of the antivirus products evaluation described above, one can categorize the utilized signatures in those based on: hash values, identifier names, strings and other program data, call graph and object-oriented design, resources inside the application package, and potentially class hierarchy. Note that some antivirus programs seem to use different signature types for different malware samples.

4.3 Plagiarized Application Diversification

The next scenario studies the confrontation between software pirates and plagiarism detection tools. Similarly to the case of malware, this evaluation works both ways: similarity measurement algorithms have to show resilience against transformation techniques available to pirates, whereas “good” transformations are distinguished by the ability to confuse popular software similarity algorithms.

For this part of the evaluation, 13 open source Android applications are assumed to be “stolen”. As a main opponent for PANDORA, we use the AndroSim tool from the Androguard toolbox.

4.3.1 Similarity Experiment

The AndroSim tool gives the similarity score of 100% to identical applications [9]. The possible similarity values reached by distinct applications, however, are unknown.

To assess the similarity score threshold, below which the recognition of a program as “plagiarized” becomes impractical, a similarity measure between all 78 distinct pairs of the selected 13 applications were computed, which inferred the mean value close to 10%. This defines the goal of program transformation tools in the context of the application piracy scenario.

4.3.2 Concealing Plagiarism

Each of the “stolen” applications was transformed according to the levels L0, L1, L2 and L3. For each application and each transformation level, the similarity with the original version is measured using AndroSim. The results indicate that level L0 and L1 transformations yield a relatively high similarity with the original program, about 90% and 60% on average respectively, and therefore the detection of plagiarism is possible with high precision. The L2 and L3 transformation levels, however, reach a similarity distribution very close to that of distinct programs, on average about 13% and 8% respectively, so the detection of a copy becomes intractable for the AndroSim tool.

For other software similarity algorithms than AndroSim, which are not available as public tools, we can only speculate about their potential performance against the transformation of PANDORA. Knowing the way PANDORA changes a program’s structure and methods of the fingerprint extraction, we can assume that our proposed transformation system shows high performance against most of the static software similarity algorithms published in the literature, with exception of methods based on Inheritance Structure by Tamada et al. [30]. Recent approaches dedicated especially to the Android piracy problem presented by Potharaju et al. [23] and Zhou et al. [39], are also anticipated to fail recognition of transformed application copies. The PANDORA transformation system, however, cannot be expected to defeat dynamic similarity measurement approaches, e.g., algorithms based on the sequence of API calls recorded at runtime [31, 26, 27].

4.3.3 Diversifying Effect

The goal of artificial software diversity [6] is to generate multiple semantically equivalent versions of a program, but with different code. Ideally, randomly generated versions of the same application are recognized as different applications by software similarity algorithms. The grade to which a program version produced by the PANDORA system differs from each other is evaluated now.

For this evaluation, 5 different versions of the application GhostCommander were generated with PANDORA configured at level L3. The similarities between all pairs of these versions were measured again with the AndroSim tool. The measurement outcome indicates that the similarity of different versions has a very dense distribution and is close to 35%. This almost reaches the values which could correspond to different applications, and hence witnesses that the proposed transformation system achieves a high non-determinism in the produced code.

5 Conclusion

We presented the PANDORA transformation system for Android applications. A system capable of unleashing the evils of the piracy and malware upon innocent developers and users.

Our study indicates inefficiency of static software similarity analysis algorithms in recognition of transformed application copies. This observation opens up a question of the possible theoretical and practical borders for the capabilities of similarity algorithms. The current state-of-the-art in this area raises the impression that any static similarity analysis approach can be countered by dedicated program transformation techniques. The correctness of this statement requires further investigation. Dynamic software similarity analysis algorithms, however, can be expected to have higher resilience against static program transformations.

Limitations of existing solutions are also revealed in the current antivirus and malware detection tools. This might
drive to the conclusion that static detection techniques are not to be considered reliable as such, promoting the use of dynamic and behavior-based approaches.

References


