Formal Methods Meet Mobile Code Obfuscation
Identification of Code Reordering Technique

Aniello Cimitile*, Fabio Martinelli‡, Francesco Mercaldo‡, Vittoria Nardone*, Antonella Santone*
*Department of Engineering, University of Sannio, Benevento, Italy
{cimitile, vnardone, santone}@unisannio.it
‡Institute for Informatics and Telematics, National Research Council of Italy (CNR), Pisa, Italy
{fabio.martinelli, francesco.mercaldo}@iit.cnr.it

***Abstract*—Android represents the most widespread mobile environment. This increasing diffusion is the reason why attackers are attracted to develop malware targeting this platform. Malware writers usually use code obfuscation techniques in order to evade the current antimalware detection and to generate new malware variants. These techniques make code programs harder to understand and they change the signature of the application making ineffective the signature extraction work. We propose a method based on formal methods able to identify whether a mobile application is obfuscated. In this preliminary work we identify one of the most widespread obfuscation technique: the code reordering. We test our method on a real-world dataset composed by Android trusted and ransomware samples, obtaining encouraging results.**

***Keywords*—Obfuscation, Malware, Android, Model Checking, Formal Methods**

I. Introduction

In recent years Android has become the most popular operating system for mobile devices. Due to the popularity of the platform, the number of Android applications has grown massively. However, as the number of mobile app grows, software theft such as unauthorized duplication and plagiarism is also increasing fast.

As demonstrated in [1], with increased incentives and low barriers to entry, plagiarists and clones have followed. In order to combat cloning, mobile markets need robust techniques identify these clones, and in order to defend their intellectual property, developers use obfuscation to try to make their code less understandable.

Reverse engineering of Android applications is very easy and this is the reason why obfuscation techniques are employed to protect intellectual property of software. The main aim of obfuscation is to make software harder to understand.

From the malware writers point of view, obfuscation is employed to make a malware identified from current antimalware technologies not more recognizable applying trivial code transformation. Many studies demonstrate that the signature based detection mechanism currently provided by free and commercial antimalware can be easily evaded with code obfuscation [2]–[7].

There is a need to protect users from malicious applications and developers from plagiarists who wish to benefit from a legitimate developer's hard work.

The evolution of Android malware has made incontestable progress in the last few years and it often follows in the footsteps of PC-based malware, except that it happens at an accelerated pace, as Symantec experts report[1]: the conclusion is that Android malware authors started to obfuscate their code in an attempt to slow down discovery and detection by current antimalware technologies.

The techniques adopted by attackers are increasingly sophisticated: the current antimalware are not able to detect malware when the signature mutates [8]. During its propagation, malware code changes its structure, through a set of transformations, in order to elude signature-based detection strategies. Indeed, polymorphism and metamorphism are rapidly spreading among malware targeting mobile applications [4].

In order to help the current antimalware in detection we propose a model checking based methodology able to identify whether an Android application is obfuscated. In this preliminary study we focus on the code reordering obfuscation technique. This transformation is aimed at modifying the instructions order in *smali* methods. A random reordering of instructions has been accomplished by inserting *goto* instructions with the aim of preserving the original runtime execution trace.

Paper structure. The paper proceeds as follows: Section II discusses the related literature; Section III describes the methodology; Section IV illustrates the results of experiments and finally, conclusions and future works are drawn in the Section V.

II. Related Work

Several works in literature evaluated the effectiveness of existing mobile malware detection mechanisms. In [5], authors present ADAM, an automated system for evaluating the detection of Android malware. Using ADAM, researchers apply a set of trivial obfuscation techniques to a dataset containing 222 malware samples. For each antimalware software, results show how each of the obfuscations can significantly reduce the detection rate.

Authors in[2] tested 11 antimalware solutions using 10 malware samples and 10 obfuscated ones. In the evaluation

[1]https://www.symantec.com/connect/blogs/obfuscating-embedded-malware-android
[2]https://www.aisec.fraunhofer.de/content/dam/aisec/Dokumente/Publikationen/Studien_TechReports/deutsch/042013-Technical-Report-Android-Virus-Test.pdf

they show that only 7 on 10 antimalware are able to recognize the sample without transformations; while using the altered samples the malware identification decreases drastically.

Researchers in [4] evaluated 10 antimalware tools using 6 original and transformed malware samples belonging to 6 different families. They concluded that all the antimalware products are susceptible to common evasion techniques.

Dalla Preda et alius [9] demonstrate that an algorithm for control obfuscation by opaque predicate insertion can be systematically derived as an abstraction of a suitable semantic transformation; the proposed method method assumes that an attacker has a constrained observation on the behavior of the program.

In [2] authors demonstrated that by using simple code transformations to existing Android malware that are well recognized by malware detectors turns it in a version that is not anymore recognized by the most malware detectors.

Scientific community has also proposed several methods to identify mobile malware in Android environment. As matter of fact, recently, the possibility to identify the malicious payload in Android malware using a model checking based approach has been explored in [10]–[13]. Starting from payload behavior definition authors formulate logic rules and then test them by using Android malware.

In [14] the authors introduce the specification language CTPL (Computation Tree Predicate Logic) confirming the malicious behavior of thirteen Windows malware variants using as dataset a set of worms dating from 2002 to 2004.

Song et al. present an approach to model Microsoft Windows XP binary programs as a PushDown System (PDS) [15]. They evaluate 200 malware variants (generated by NGVCK and VCL32 engines) and 8 benign programs.

The tool PoMMaDe [16] is able to detect 600 real malware, 200 malware generated by two malware generators (NGVCK and VCL32), and proves the reliability of benign programs: a Microsoft Windows binary program is modeled as a PDS which allows to track the stack of the program.

Song et al. [17] model mobile applications using a PDS discover private data leaking working at Smali code level.

III. Methodology

In this section we present our model checking based methodology. It is based on two main processes:

1) Translation of Java Bytecode into formal models.
2) Translation of obfuscation technique behaviour into temporal logic formulae.

Figure 1 shows the schema of our methodology.

The first process uses the Calculus of Communicating Systems (CCS) of Milner [18], the specification language used to generate the formal model. It formally describes the analyzed mobile application. Starting from *.class* files written in Java Bytecode we produce a CCS specification. The transformation from Java Bytecode to CCS is performed by defining a translation function. It is defined for each Java Bytecode instructions, each one of them is translated into CCS processes. This process has been already used in our previous works [10]–[12], [19] for other purposes. In particular, it has been used to identify malware families. Now, we are going to use this formal-based process to try to identify one of the most common obfuscation technique: *code reordering*. Clearly, some changes have been introduced to operate in this new context.

Code reordering has the aim of modifying the instructions order in the methods. A random reordering of instructions is accomplished by inserting goto instructions. Obviously, the transformations preserve the original runtime execution traces, i.e., the business logic of the applications is not modified by the transformations.

Figure 2 shows, in the left side, a simple automaton related to a CCS model that is the result of the application of the transformation function, defined in the first process of our methodology, applied to a fragment of an original application. Successively, the original app (a fragment) has been obfuscated, with code reordering, producing a different CCS process whose automaton is shown in the right side of Figure 2. It is worth noting that the behavior of the two automata is the same but in the automaton in the right side more goto edges have been introduced. Note that in both the automata the initial state is recognized with an in-going edge.

The second process aims to investigate whether an application is obfuscated. Code reordering is the analysed obfuscation technique. The main aim of our methodology is to verify whether an application is obfuscated with this technique. Thus, we use mu-calculus logic, [20] as a branching temporal logic, to express code reordering characteristic. The CCS processes obtained by the first step are used to verify properties. Codes described as CCS processes are first mapped to labeled transition systems and a model checker is used. In our approach, we invoke the Concurrency Workbench of New Century (CWB-NC) [21] as formal verification environment. When the result of the CWB-NC model checker is *true*, it means that we consider the sample under analysis as obfuscated with code reordering technique, otherwise the sample is considered not obfuscated with code reordering.

Model checking approaches [22] have been investigated in conjunction with a variety of disciplines such as biology [23], [24], design pattern [25], information flow [26], clone detection [27] among others. In this paper we address the problem of code obfuscation detection exploiting the power of model checking.

IV. Preliminary Evaluation

In following section we present the evaluation results of our method on code reordering identification.

A. Dataset

To evaluate our method we use a dataset of real word Android samples, belonging to the ransomware typology and to the Google Play market[3], and the obfuscated version of the same dataset injected with the core reordering transformation.

[3]https://play.google.com/

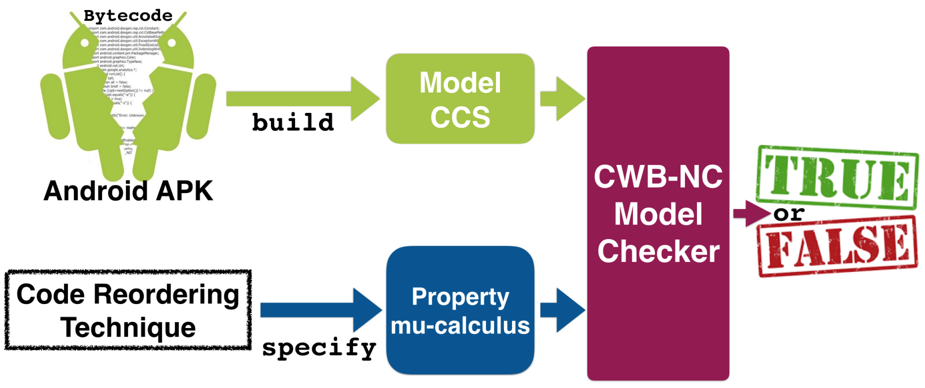


Fig. 1. The Workflow of our Approach

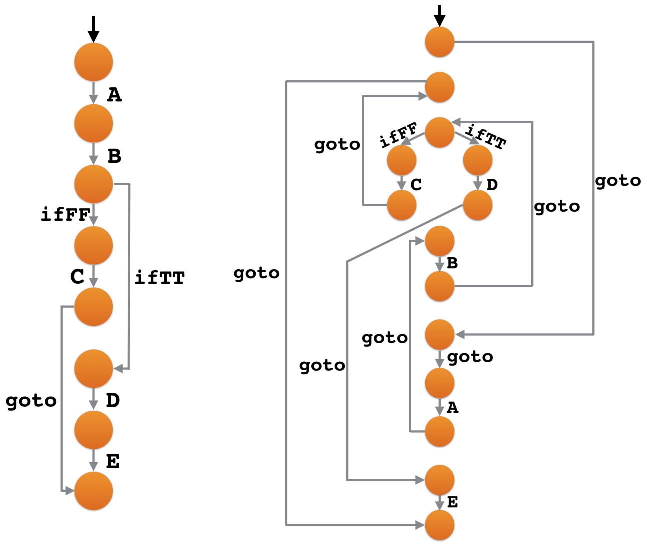


Fig. 2. Two automata

We tested ransomware because it is currently represents one of the most dangerous threats in mobile environment [10], [28]. The samples belonging to the ransomware category are able to impede the access to smartphone resources and demand a payment for restoring the functionality and the resources. The real world samples examined in the experiment were gathered from a collection of freely available 672 samples [4]. The samples are labeled as ransomware, koler, locker, fbilocker and scarepackage [29] and appeared from December 2014 to June 2015. In order to download legitimate applications, we crawled the Google Play market using an open-source crawler[5]. The obtained trusted dataset includes 500 samples belonging to all the different categories available on the market.

We submitted the dataset belonging to ransomware category and to Google Play one in order to confirm respectively their maliciousness and their trustiness to the VirusTotal service[6]. VirusTotal provides a public API to submit and scan applications, access finished scan reports of 57 different antimalware without the need of using the HTML website interface.

In order to generate morphed applications, we developed a framework[7] able to inject several obfuscation levels in Android applications: (i) changing package name; (ii) identifier renaming; (iii) data encoding; (iv) call indirection; (v) code reordering; (vi) junk code insertion.

To apply the transformation, the tools extract the dex file of the application which is totally unreadable; for this reason the tools convert this dex files to a more understandable form represented by the smali language. The smali syntax is loosely based on Jasmin's/dedexer's syntax[8], and supports the full functionality of the dex format (annotations, debug info, line info) and Dalvik opcodes. Previous works [2] demonstrated that antimalware solutions fail to recognize the malware after these transformations. We applied our method to the morphed dataset in order to identify if the code reordering transformation is applied. In a second step we apply our methodology on the original dataset to verify whether our approach discriminates between morphed sample and original ones.

[4]http://ransom.mobi/
[5]https://github.com/liato/android-market-api-py
[6]https://www.virustotal.com
[7]https://github.com/faber03/AndroidMalwareEvaluatingTools
[8]http://jasmin.sourceforge.net/

For the sake of clarity Figure 3 shows the bytecode snippet related to the set of the device admin privileges before the application of code reordering technique, while bytecode snippet in Figure 4 shows the previous code snippet after the application of the code reordering morphing technique.

The code is related to the ransomware capability to obtain admin permission at runtime. As matter of fact, it demands to the user to install a video/flash player application. Once the user has grant the admin permission and has installed the application, the device will be locked and to get back the control of the device, the user have to pay the ransom. The *ACTION_ADD_DEVICE_ADMIN* Intent triggers the process of enabling the user to obtain the admin permission, the *DEVICE_ADMIN* represents the administrator component and *ADD_EXPLANATION* String represents an optional CharSequence providing additional explanation for why the admin is being added (in this case the explanation is ”To continue, you need activate the application. Click to activate/enable”). The user believes to install a video play, but he/she confirms to install the payload able to perform the malicious action. The snippet was retrieved using BytecodeViewer tool[9], an open-source Java and Android reverse engineering suite.

Considering that the signature provided by current antimalware are mainly based on opcode and/or using the Control/Call Graph of the application, it is very easy to circumvent the detection leaving intact the business logic of the application by using the application of the code reordering.

B. Results

To estimate the code reordering identification performance of our methodology we compute the metrics of precision (PR), recall (RC), F-measure (Fm) and Accuracy (Acc), defined as follows:

$$PR = \frac{TP}{TP+FP}; \; RC = \frac{TP}{TP+FN};$$
$$Fm = \frac{2PR\,RC}{PR+RC}; \; Acc = \frac{TP+TN}{TP+FN+FP+TN}$$

where TP is the number of samples that was correctly identified as belonging to the obfuscated dataset (True Positives), TN is the number of samples correctly identified as not belonging to the obfuscated dataset (True Negatives), FP is the number of samples that was incorrectly identified as obfuscated samples (False Positives), and FN is the number of samples that was not identified as belonging to the obfuscated dataset (False Negatives).

Table I shows the results of the experiment: we obtain an accuracy equal to 0.93 in code reordering obfuscation identification.

As Table I shows, we test a ransomware dataset composed by 483 obfuscated samples, this happens because the compilation process after the transformation injection is not always successful: for this reason the number of ransomware obfuscated samples is less (483) if compared with original ones (672). The samples downloaded from Google Play were successfully obfuscated.

[9]https://bytecodeviewer.com/

The results are symptomatic that the logic rules associated to the code reordering obfuscation are effectively representative of the obfuscation technique. We are able to identify the code reordering technique both in malware samples and in legitimate ones, from this point of view our method is transparent to the maliciousness or the trustiness of the application. As side effects, the methodology is able to discriminate obfuscated samples from samples without obfuscation.

C. Time Performance Evaluation

In order to measure the performance, we used the *System.currentTimeMillis()* Java method that returns the current time in milliseconds. Table II shows the performance of our method. In particular, we consider the overall time to analyse a sample as the sum of two different contributions: the average time required to extract the class files of the application using the dex2jar tool ($t_{dex2jar}$) and the time required to obtain the response from our tool ($t_{response}$). These two values are the average times, i.e., they are computed as the total time employed by the tool to process the samples divided the number of samples evaluated.

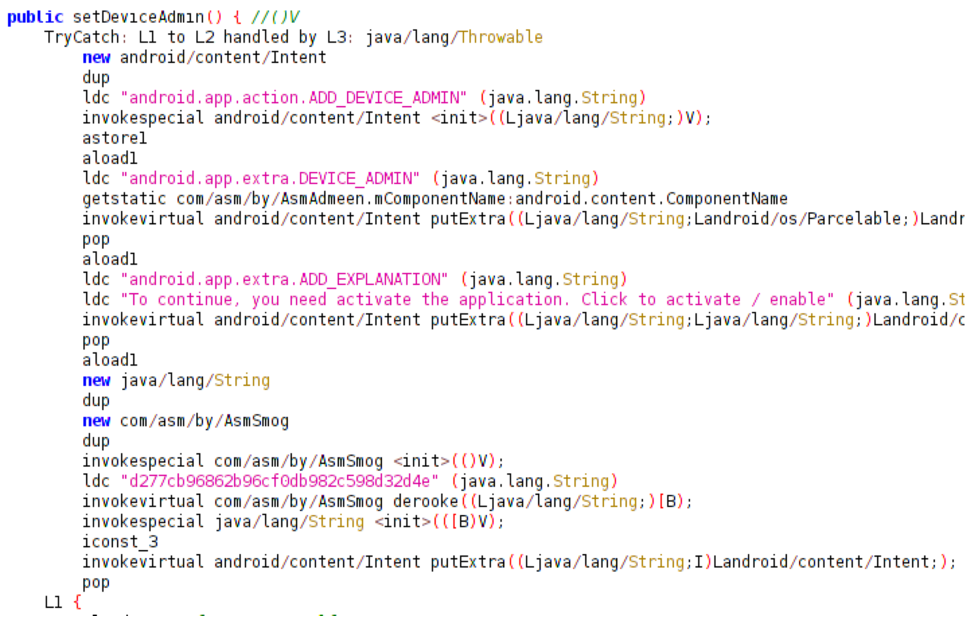
The machine used to run the experiments and to take measurements was an Intel Core i5 desktop with 4 gigabyte RAM, equipped with Microsoft Windows 7 (64 bit).

V. Conclusion and Future Work

Nowadays, Android malware detectors are able to identify a malware using its signature, usually extracted from manual inspection. The signature is usually based on code structure, for instance using the control flow graph and/or control dependency graph. Therefore, it is very easy to evade the mechanism of signature recognition by trivial alteration of code structure. This is the reason why in the Android malware landscape exists the so-called families of malware, i.e., samples that implement similar high-level malicious behaviour, but with different code structure. The scientific community also developed tools able to automatically change the malware signature (in order to demonstrate the weakness of current antimalware technologies), applying trivial code transformation, such as code reordering. In this preliminary study we apply formal methods to verify whether a sample is injected with the code reordering transformation, one of the most widespread technique to alter malware signature. We have implemented our method in a prototype tool. Thus, we evaluate our method on Android ransomware malware, one of the most recent attack plaguing the mobile environment and on applications downloaded from Google Play. Thus, we evaluated the ransomware and trusted dataset and another one, composed by the same samples after code reordering transformation, in order to demonstrate the effectiveness of our method. The results are very encouraging. As future work, we plan to formulate logic rules able to identify others obfuscation techniques, as junk code insertion, data encoding and changing package/method name and to investigate the use of equivalence checking [30], [31].

Acknowledgments

This work has been partially supported by H2020 EU-funded projects NeCS and C3ISP and EIT-Digital Project HII and PRIN “Governing Adaptive and Unplanned Systems of Systems”



```
public setDeviceAdmin() { //()V
    TryCatch: L1 to L2 handled by L3: java/lang/Throwable
        new android/content/Intent
        dup
        ldc "android.app.action.ADD_DEVICE_ADMIN" (java.lang.String)
        invokespecial android/content/Intent <init>((Ljava/lang/String;)V);
        astore1
        aload1
        ldc "android.app.extra.DEVICE_ADMIN" (java.lang.String)
        getstatic com/asm/by/AsmAdmeen.mComponentName:android.content.ComponentName
        invokevirtual android/content/Intent putExtra((Ljava/lang/String;Landroid/os/Parcelable;)Landr
        pop
        aload1
        ldc "android.app.extra.ADD_EXPLANATION" (java.lang.String)
        ldc "To continue, you need activate the application. Click to activate / enable" (java.lang.St
        invokevirtual android/content/Intent putExtra((Ljava/lang/String;Ljava/lang/String;)Landroid/c
        pop
        aload1
        new java/lang/String
        dup
        new com/asm/by/AsmSmog
        dup
        invokespecial com/asm/by/AsmSmog <init>(()V);
        ldc "d277cb96862b96cf0db982c598d32d4e" (java.lang.String)
        invokevirtual com/asm/by/AsmSmog derooke((Ljava/lang/String;)[B);
        invokespecial java/lang/String <init>(([B)V);
        iconst_3
        invokevirtual android/content/Intent putExtra((Ljava/lang/String;I)Landroid/content/Intent;);
        pop
    L1 {
```

Fig. 3. Bytecode snippet related to a ransomware sample with lock-screen abilities identified by the following hash: *0a0a48cc3e43ae888786c4da36b6ce76dfef3eaf8697118901bc971cd788816b*

TABLE I. PERFORMANCE EVALUATION

Label	Original	Obfuscated	TP	FP	FN	TN	PR	RC	Fm	Acc
Ransomware	672	483	411	0	72	672	1	0.85	**0.91**	**0.93**
Trusted	500	500	500	0	0	500	1	1	**1**	**1**

TABLE II. TIME PERFORMANCE EVALUATION (VALUES ARE EXPRESSED IN SECONDS)

$t_{dex2jar}$	$t_{response}$	**Total Time**
9.39 s	10.48 s	19.88 s

REFERENCES

[1] Crussell, J., Gibler, C., Chen, H.: Attack of the clones: Detecting cloned applications on android markets. In: European Symposium on Research in Computer Security, Springer (2012) 37–54

[2] Canfora, G., Di Sorbo, A., Mercaldo, F., Visaggio, C.A.: Obfuscation techniques against signature-based detection: a case study. In: 2015 Mobile Systems Technologies Workshop (MST), IEEE (2015) 21–26

[3] Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, ACM (2013) 329–334

[4] Rastogi, V., Chen, Y., Jiang, X.: Catch me if you can: Evaluating android anti-malware against transformation attacks. Information Forensics and Security, IEEE Transactions on **9**(1) (Jan 2014) 99–108

[5] Zheng, M., Lee, P.P.C., Lui, J.C.S.: Adam: An automatic and extensible platform to stress test android anti-virus systems. In: Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA'12 (2013) 82–101

[6] Martinelli, F., Mercaldo, F., Saracino, A.: Bridemaid: An hybrid tool for accurate detection of android malware. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. ASIA CCS '17, New York, NY, USA, ACM (2017) 899–901

[7] Canfora, G., Mercaldo, F., Visaggio, C.A.: A classifier of malicious android applications. In: Availability, Reliability and Security (ARES), 2013 Eighth International Conference on, IEEE (2013) 607–614

[8] Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Proceedings of 33rd IEEE Symposium on Security and Privacy (Oakland 2012), IEEE (2012)

[9] Dalla Preda, M., Giacobazzi, R.: Control code obfuscation by abstract interpretation. In: Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on, IEEE (2005) 301–310

[10] Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A.: Ransomware steals your phone. Formal methods rescue it. In: 11th International Federated Conference on Distributed Computing Techniques, DisCoTec, Springer (2016)

[11] Battista, P., Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.: Identification of android malware families with model checking. In: International Conference on Information Systems Security and Privacy. SCITEPRESS. (2016)

[12] Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A.: Download malware? No, thanks. How formal methods can block update attacks. In: Formal Methods in Software Engineering (FormaliSE), 2016 IEEE/ACM 4th FME Workshop on, IEEE (2016)

[13] Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A.: Hey malware, i can find you! In: 2016 IEEE 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE). (June 2016) 261–262

[14] Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking, Springer (2005)

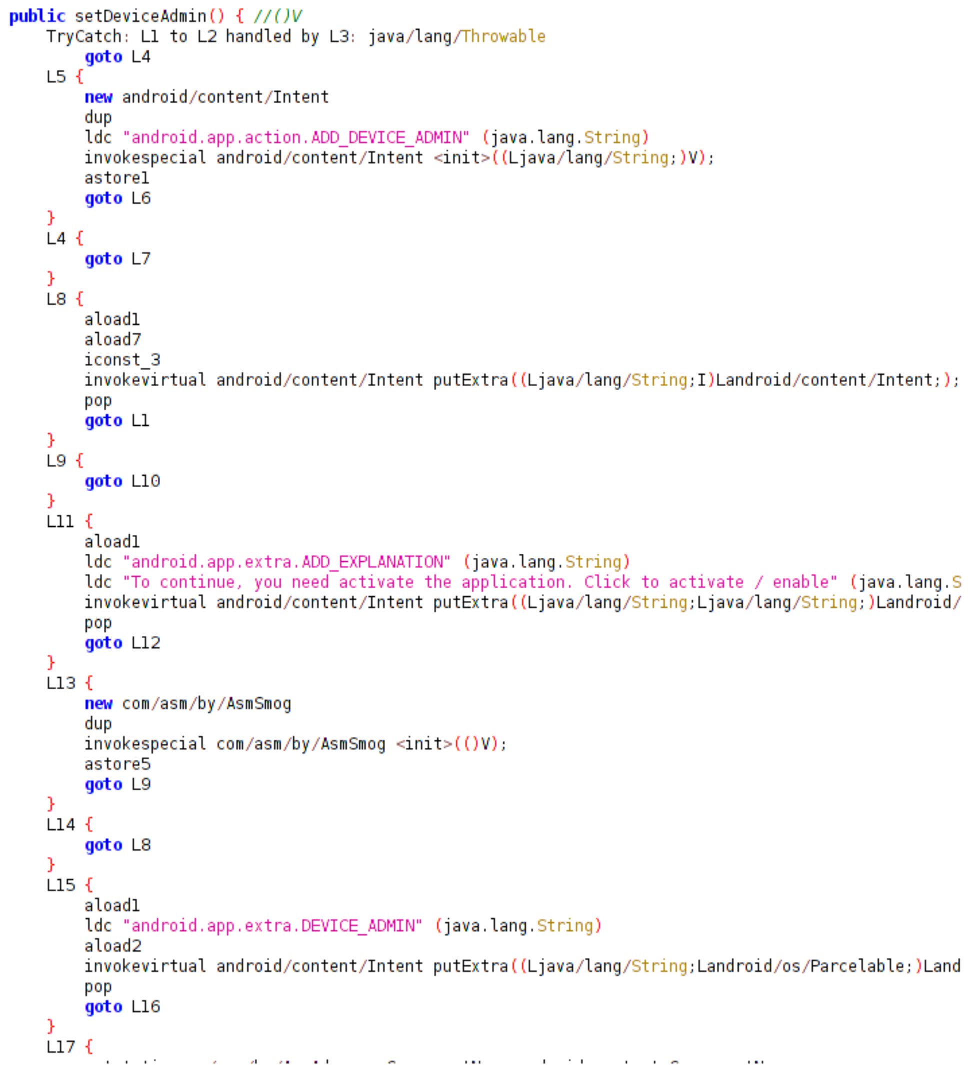
[15] Song, F., Touili, T.: Efficient malware detection using model-checking, Springer (2001)

[16] Song, F., Touili, T.: Pommade: Pushdown model-checking for malware detection. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM (2013)

[17] Song, F., Touili, T.: Model-checking for android malware detection, Springer (2014)

[18] Milner, R.: Communication and concurrency. PHI Series in computer science. Prentice Hall (1989)

[19] Martinelli, F., Mercaldo, F., Nardone, V., Santone, A.: How discover a malware using model checking. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. ASIA CCS '17, New York, NY, USA, ACM (2017) 902–904



```
public setDeviceAdmin() { //()V
    TryCatch: L1 to L2 handled by L3: java/lang/Throwable
        goto L4
    L5 {
        new android/content/Intent
        dup
        ldc "android.app.action.ADD_DEVICE_ADMIN" (java.lang.String)
        invokespecial android/content/Intent <init>((Ljava/lang/String;)V);
        astore1
        goto L6
    }
    L4 {
        goto L7
    }
    L8 {
        aload1
        aload7
        iconst_3
        invokevirtual android/content/Intent putExtra((Ljava/lang/String;I)Landroid/content/Intent;);
        pop
        goto L1
    }
    L9 {
        goto L10
    }
    L11 {
        aload1
        ldc "android.app.extra.ADD_EXPLANATION" (java.lang.String)
        ldc "To continue, you need activate the application. Click to activate / enable" (java.lang.S
        invokevirtual android/content/Intent putExtra((Ljava/lang/String;Ljava/lang/String;)Landroid/
        pop
        goto L12
    }
    L13 {
        new com/asm/by/AsmSmog
        dup
        invokespecial com/asm/by/AsmSmog <init>(()V);
        astore5
        goto L9
    }
    L14 {
        goto L8
    }
    L15 {
        aload1
        ldc "android.app.extra.DEVICE_ADMIN" (java.lang.String)
        aload2
        invokevirtual android/content/Intent putExtra((Ljava/lang/String;Landroid/os/Parcelable;)Land
        pop
        goto L16
    }
    L17 {
```

Fig. 4. Bytecode snippet related to the code in Figure 3 after the application of code reordering technique. The code appears hard to understand after the application of the code reordering obfuscation technique

[20] Stirling, C.: An introduction to modal and temporal logics for ccs. In Yonezawa, A., Ito, T., eds.: Concurrency: Theory, Language, And Architecture. LNCS, Springer (1989) 2–20

[21] Cleaveland, R., Sims, S.: The ncsu concurrency workbench. In Alur, R., Henzinger, T.A., eds.: CAV. Volume 1102 of Lecture Notes in Computer Science., Springer (1996)

[22] Barbuti, R., De Francesco, N., Santone, A., Vaglini, G.: Reduced models for efficient CCS verification. Formal Methods in System Design **26**(3) (2005) 319–350

[23] Ceccarelli, M., Cerulo, L., Ruvo, G.D., Nardone, V., Santone, A.: Infer gene regulatory networks from time series data with probabilistic model checking. In: 3rd IEEE/ACM FME Workshop on Formal Methods in Software Engineering, FormaliSE 2015, Florence, Italy, May 18, 2015, IEEE Computer Society (2015) 26–32

[24] Ceccarelli, M., Cerulo, L., Santone, A.: De novo reconstruction of gene regulatory networks from time series data, an approach based on formal methods. Methods **69**(3) (2014) 298 – 305

[25] Bernardi, M., Cimitile, M., De Ruvo, G., Di Lucca, G., Santone, A.: Integrating model driven and model checking to mine design patterns. Communications in Computer and Information Science **586** (2016) 99–117

[26] Barbuti, R., De Francesco, N., Santone, A., Tesei, L.: A notion of non-interference for timed automata. Fundam. Inform. **51**(1-2) (2002) 1–11

[27] Santone, A.: Clone detection through process algebras and java bytecode. In: Proceeding of the 5th ICSE International Workshop on Software Clones, IWSC 2011, Waikiki, Honolulu, HI, USA, May 23, 2011, ACM (2011) 73–74

[28] Mercaldo, F., Nardone, V., Santone, A.: Ransomware inside out. In: Availability, Reliability and Security (ARES), 2016 11th International Conference on, IEEE (2016) 628–637

[29] Andronio, N., Zanero, S., Maggi, F.: Heldroid: Dissecting and detecting mobile ransomware. In: Research in Attacks, Intrusions, and Defenses. Springer (2015) 382–404

[30] De Francesco, N., Lettieri, G., Santone, A., Vaglini, G.: Heuristic search for equivalence checking. Software and System Modeling **15**(2) (2016) 513–530

[31] De Francesco, N., Lettieri, G., Santone, A., Vaglini, G.: Grease: A tool for efficient ”nonequivalence” checking. ACM Trans. Softw. Eng. Methodol. **23**(3) (2014) 24:1–24:26