2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)

Deobfuscating Android Native Binary Code

Zeliang Kan[1], Haoyu Wang[1], Lei Wu[2], Yao Guo[3], Guoai Xu[1]
[1] Beijing University of Posts and Telecommunications [2] PeckShield, Inc. [3] Peking University

***Abstract*—In this paper, we propose an automated approach to facilitate the deobfuscation of Android native binary code. Specifically, given a native binary obfuscated by Obfuscator-LLVM (the most popular native code obfuscator), our deobfuscation system is capable of recovering the original Control Flow Graph. To the best of our knowledge, it is the first work that aims to tackle the problem. We have applied our system in different scenarios, and the experimental results demonstrate the effectiveness of our system based on generic similarity comparison metrics.**

I. Introduction

Android apps have received widespread adoption in recent year. Besides the standard Android programming model in Java, Android Native Development Kit (NDK) is introduced to allow developers to include native code binaries (write in C/C++) in their apps.

Problem: Recent work suggested that native code had been widely used in Android apps, which severely complicated the process of static analysis. As Java bytecode can be more easily decompiled, malware developers usually hide the malicious payload and core functionalities in the native code to evade detection. Even worse, the native code can be obfuscated, which further increases the difficulty of security analysis. For example, the towelroot exploit (CVE-2014-3153), one of the the biggest Root Exploits family in Android, was found obfuscated at the native code level by O-LLVM. It took a lot of efforts for security researchers to dive into the technical details of the code due to the obfuscation.

Although several attempts have discussed the deobfuscation of Android apps, all of the them focus on Java-level deobfuscation. For example, DeGuard [3] was proposed to reverse layout obfuscation (naming obfuscation) generated by ProGuard. Their key idea is to learn a probabilistic model over a large number of non-obfuscated apps and to use this model to deobfuscate new APKs. Layout obfuscation is the easiest one in Android app obfuscation, which does not alter the program logic (e.g., control flow) of the apps. To the best of our knowledge, none of previous studies have attempted to tackle native code deobfuscation yet.

Our approach: In this paper, we propose a novel approach for automated deobfuscation of Android native binary code. Technically, our system works by combining static taint analysis and symbolic execution to remove the obfuscation introduced by different obfuscating techniques in O-LLVM. O-LLVM [4] is one of the most widely used code obfuscator for both x86 and ARM platforms. It is implemented as middle-end passes in the Low Level Virtual Machine (LLVM) compilation process, which offers guaranteed compatibility with LLVM, including *Instruction Substitution (InsSub)*, *Bogus Control Flow (BCF)* and *Control Flow Flattening (CFF)*. One key feature of our system is that we introduce taint analysis to make semantic level deobfuscation in both general and instruction optimization situation. We also exploit flow-sensitive symbolic execution to rebuild the seriously obfuscated control flow. To overcome the challenge of basic block splitting, we chopped the original control flow, selected analysis targets through static features, and dynamically adjust the analysis target sequence to maximize context inheritance.

Experiments on multiple benchmarks and real-world cases suggest that our approach is capable of accurately deobfuscating samples obfuscated by O-LLVM. We believe our tool can be used by security analysts to make it easier to inspect native code, even it is heavily obfuscated.

II. Approach Overview

Goal. We use the term *deobfuscation* to refer to the process of removing the effects of code obfuscation from the native binary, and ideally recover the original code and control flow before obfuscation. For a given APK as input, our system first extracts the native binary and determines whether it is obfuscated, then analyzes and transforms the code to obtain a functionally equivalent form that is simpler and easier to understand. For the non-trivial code, the deobfuscation result is rarely the same as the original code, however, it is close to the original and much easier to understand compared to the obfuscated version.

Key Techniques. To address the aforementioned challenges, we rely on taint analysis and enhanced symbolic execution to recover O-LLVM obfuscated code. Taint analysis is used to address the challenge introduced by instruction optimization, which performs a globally feature matching to comprehensively detect all obfuscations introduced by O-LLVM and identify instructions needed to be rewrite by tracking tainted registers. Symbolic execution is used to reconstruct the control flow ruined by Control Flow Flattening (CFF). We first identify basic blocks that maintain original operations based on an ARM-specific basic block classification approach. Then we perform chopped symbolic execution to address the path explosion challenge. To perform flow-sensitive symbolic execution, we use a dynamic exchanging model to maximize the context inheritance and rebuild the original control flow.

A. Instruction Substitution Deobfuscation

We apply taint analysis to determine the combination of obfuscating instructions and locate instructions needed to be rewritten. During our analysis, we found that sometimes obfuscated instructions may be mixed with normal instructions.

2574-1934/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-Companion.2019.00135

For example, normal instructions use the registers which obfuscated instructions modified. The taint analysis we use here guarantees the integrity of the deobfuscation analysis. The content and address of the deobfuscated instructions that need to be modified will be saved temporarily and wait until other deobfuscating work finished, before rewritten to the binary.

B. Bogus Control Flow Deobfuscation

We rely on assembly features to detect whether a binary is obfuscated by BCF. When the opaque predicate operation is detected in a Basic Block (BB), taint analysis first sets the BB as a predicate block and sets operated register as a tainted register. Then it will force the conditional jump in the parent block reaching the predicate block. After that, because only one condition jump of the predicate block can actually be accessed, taint analysis will head to the "alive" branch and label jump instruction at the end of predicate block as to be modified. Due to the optimization on ARM instructions during compilation, sometimes BCF obfuscation can occur in a BB that relay on a constraint, which is a tainted register compared to zero (there are some cases that compare with 1), our system will also automatically set it as a predicate block. After taint analysis, all deobfuscated information about instructions that needs to be modified will be used to rebuild the binary.

C. Control Flow Flattening Deobfuscation

The CFF obfuscation rebuild the control flow to a SWITCH construct. To deobfuscate the control flow, we need to identify basic blocks that maintain original operations and rebuild the control flow. Since the analysis of CFF obfuscated code is flow sensitive, and also symbolic execution has been proved as an effective program analysis technique that can systematically explore multiple program paths. It is a means of using symbolic input to analyze a program to determine what inputs cause each part of a program to execute.

In our approach, our system first determines whether the code is obfuscated by CFF. Then static analysis is applied to identify basic blocks that contain original operations. After that, we proposed a dynamic exchanging algorithm to promise our system to perform flow sensitive symbolic execution on these basic blocks, and to reconstruct the original control flow. The purpose of dynamically exchanging basic blocks in the analysis sequence is to ensure that most of the basic blocks inherit the state from the previous analysis before execution, while not starting the analysis of a basic block from a blank state. Also, during the symbolic execution, it is possible to enter other function's space through the function call. Our system hooks all possible call instructions. When the symbol execution encounters hooked addresses, it will automatically skip the call instruction and continue the analysis in the space of the current function. After being optimized, the final deobfuscation result will be written to the output.

III. Evaluation

We have applied our system to three different datasets. We first evaluate our system on a widely used C/C++ obfuscation benchmark [2]. To evaluate our system on real-world Android apps, we have also identified 5 Android projects that contain open source C/C++ code, which contain 56 functions in total. We further apply our system to towelroot exploit [1], a binary tool used to obtain the root privilege by exploiting privilege escalation vulnerabilities. In this case, we could evaluate the effectiveness of our approach on deobfuscating real threats in the wild.

InsSub: All the obfuscated operations could be successfully recovered in our experiment. The Euclidean distances between obfuscated function and original one are in the range from 14 to 44 during experiment. After deobfuscation, the distances of the recovered functions are all below 10.

BCF: We evaluate our system of BCF deobfuscation on the C/C++ obfuscation benchmark and real-world Android apps. At default, the average CFG similarity between the original program and the obfuscated program is below 0.4, while it could reach up to 0.870 after deobfuscation.

CFF: We evaluate our system of CFF deobfuscation on three different datasets. At the default obfuscation level, the average CFG similarity score is roughly 0.2 after obfuscation, while our deobfuscation results could achieve a similarity score of 0.807 on average. After activating basic block splitting, similarity scores between the obfuscated and unobfuscated function are almost all negative correlation, which indicates that the obfuscated CFG is completely different from the original one. The deobfuscating result of the enhanced CFF is ranging from 0.72 to 1, and on average is 0.84.

Full Obfuscation (All 3 Techniques): We further evaluate the effectiveness of our approach on recovering the samples obfuscated by all three obfuscation techniques at the same time. The CFG similarity score after obfuscation is -0.144 on average. However, our system could achieve very good results compared with single pass evaluation, with a CFG similarity score above 0.834 after deobfuscation.

IV. Conclusion

We have presented a novel approach for deobfuscation of Android native code based on taint analysis and flow-sensitive symbolic execution. Experiments suggested that we could successfully reverse obfuscations performed by O-LLVM with high accuracy. We believe that our system could become a useful tool for security analysts and researchers to conduct studies including malware detection and program analysis.

Acknowledgement

This work is supported by the National Key Research and Development Program of China (grant No.2017YFB0801903) and the National NSF of China (grants No.61702045).

References

[1] CVE-2014-3153. https://github.com/orenl/CVE-2014-3153, 2014.

[2] Benchmarks. https://github.com/tum-i22/obfuscation-benchmarks, 2016.

[3] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *ACM Sigsac Conference on Computer and Communications Security*, pages 343–355, 2016.

[4] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm–software protection for the masses. In *International Workshop on Software Protection (SPRO)*, pages 3–9. IEEE, 2015.