2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing

Android App Protection via Interpretation Obfuscation

Junliang Shu, Juanru Li, Yuanyuan Zhang and Dawu Gu
Lab of Cryptology and Computer Security
Shanghai Jiao Tong University
Shanghai, China,

***Abstract*—To protect Android app from malicious reproduction or tampering, code obfuscation techniques are introduced to increase the difficulty of reverse engineering and program understanding. Current obfuscation schemes focus more on the protection of the meta information over the executable code which contains valuable or patented algorithms. Therefore, a more sophisticated obfuscator is needed to improve the protection on the executable code.**

In this paper we propose SMOG, a comprehensive executable code obfuscation system to protect Android app. SMOG is composed of two parts, an obfuscation engine and an execution environment. The obfuscation engine is at software vendor's side to conduct the obfuscation on the app's executable code, and then release the obfuscated app to the end-user along with an execution token. The execution environment is setup by integrating the received execution token, which endows the Android Dalvik VM the capability to execute the obfuscated app.

SMOG is an easily deployed system which proves fine-grained level protection. The obfuscated app generated by SMOG could resist static and dynamic reverse engineering. Moreover, the benchmark result shows SMOG only costs about 5% more performance in dispatching the incoming bytecode to the proper interpreter.

I. Introduction

Android apps face numerous threats once published. A paid app might be stolen, illegally copied, and made available for download in alternative locations. A free app might be exploited, causing damage to a brand and leveraging valuable product resources like server bandwidth. All apps are vulnerable to tampering, resulting in rogue downloads with embedded malware, or using knowledge learned to hack a server or other underlying service. Many factors help the attacker reverse engineer Android app for theft of intellectual property via software piracy: First, the Android devices are relatively open to end-users. User can easily unlock a device, apply for high privilege, acquire and analyze the app. Second, the management of Android app installation is loose. Unlike iOS which only allows the installation from the AppStore, Android allows app installation via third party app market or downloaded executable files. So pirate or modified apps can be distributed by malicious user and attacker. Third, most of the Android apps are programmed using Java and then compiled into bytecode which is a relatively semantic-rich form compared with x86 binaries, therefore Android apps are susceptible to decompilation attacks.

Since Android app is an easily reproduced product, its protection must rely on some form of active schemes such as code obfuscation. The obfuscation work for Android app faces challenging issues. Existing obfuscation schemes for Android app aim at information hiding. They either hide the meta information like identifiers and strings or transform the original control flow to a more complex one. But if an attacker has the capability of dynamically analyzing the app on Android, the interpretation of the app is exposed and the code is disassembled anyway. In this situation, existing schemes cannot protect the executable code of the app and essential information for program understanding is leaked out.

In this paper we propose **SMOG**, a comprehensive protection system based on obfuscated interpretation. SMOG consists of an *obfuscation engine* and an *execution environment* provided for both vendors and the end-users. The obfuscation engine of SMOG obfuscates the executable code of an app, then releases an obfuscated app with an *execution token*. The execution token is an Android update package that contains an enhanced SMOG interpreter. This token will be used by end-user to build a specific execution environment to execute the obfuscated app.

The core concept of SMOG is obfuscated interpretation. In detail, SMOG obfuscates one Android app into a proprietary form with opcode re-mapping and provides an *enhanced interpreter* for program interpretation. The app's bytecode is permuted according to a randomly chosen permutation matrix and thus is well protected. The permutation matrix is then used to generate an enhanced interpreter which is the core of the execution token. When the app is executed, the original interpretation process is also permuted and is hard to understand. The variant of the interpreter is numerous according to the space of the permutation matrix. An attacker can hardly employ brute force attack to find the correct matrix and reveal the protected opcode.

Our proposed system has the following advantages compared with existing obfuscation schemes:

- SMOG is the first system that attempts to implement opcode re-mapping method for Android's interpretation obfuscation. Opcode re-mapping technique has been adopted by existing x86 softwares and is proven to be a simple and effective obfuscation method. SMOG obfuscates *dex* bytecode of Android apps directly, other than obfuscates the original source code.
- SMOG aims at comprehensive protection of Android apps. Current Android obfuscation schemes are mainly designed for resisting static analysis but leaving out the bytecode semantics. SMOG could resist not only

static analysis but also dynamic analysis because the obfuscated bytecode cannot be interpreted by normal interpreter. Without the knowledge of the opcode permutation, the execution process is obfuscated and makes attacker hard to employ dynamic analysis and extract information.

- The obfuscation technique of SMOG is efficient and stable based on the observation of how Dalvik VM works[1][2]. SMOG provides an independent execution environment for efficiently executing obfuscated apps. Its enhanced interpreter replaces the Dalvik VM's interpreter to support permutation based obfuscated interpretation. This design simplifies the generation of the obfuscated app and the interpreter, and only incurs reasonable overhead. What's more, while many obfuscation tools need first transfers the *dex* file into intermediate code format to obfuscate it, our permutation based obfuscation directly operates on the Dalvik bytecode and reduces potential bug and compatibility problem.

II. The SMOG Obfuscation System

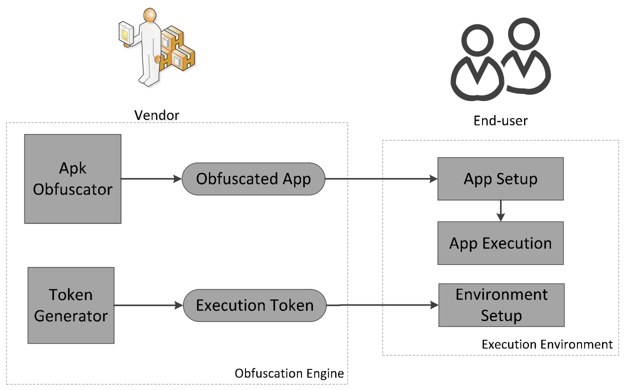


Fig. 1. SMOG Architecture

A. SMOG Overview

As depicted in Figure 1, the SMOG system is proposed as a software protection framework within which an *obfuscation engine* and an *execution environment* are provided for both software vendors and the end-users. We first focus on the high level description of SMOG's architecture and how the environments are used by apps. Then, the obfuscation mechanism is discussed in more details.

B. Obfuscation Engine

The obfuscation engine consists of two modules, an *apk obfuscator* and a *token generator*. The apk obfuscator is one of the core modules in SMOG which transforms an existing normal app into an obfuscated file through a specific *permutation handler*. The obfuscated engine receives a source program in the form of an *apk* file. Then the *apk* file is unpacked and the *dex* part is obfuscated according to the chosen matrix by the permutation handler. However, the other resource files, such as XML and image files, will remain unmodified in general. After obfuscation, the engine will re-pack and re-sign the files into an obfuscated *apk* file.

The obfuscation engine contains two major modules, an apk obfuscator and a token generator. As the kernel of SMOG, the function of the obfuscation engine is to handle source apps to provide the resistance to static analysis and generate an execution token which helps end-users to build an execution environment for obfuscated apps.

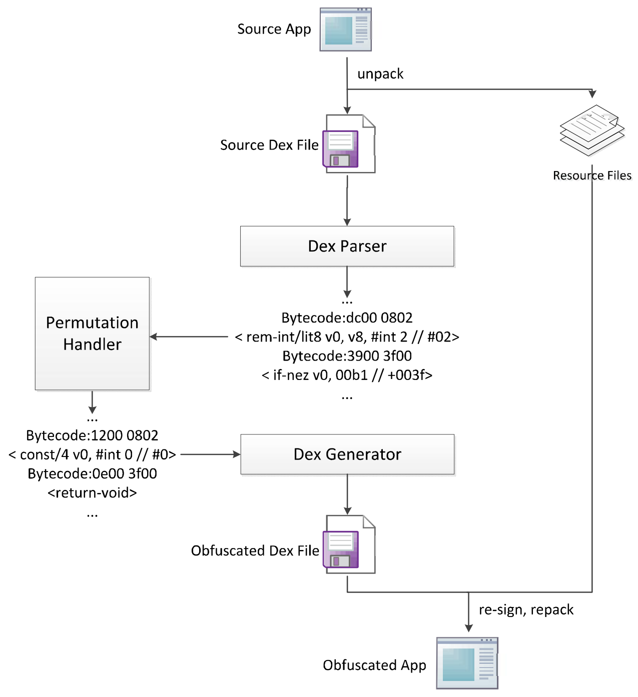


Fig. 2. Overviews of the Apk Obfuscator module

1) Apk Obfuscation: The task of apk obfuscator is to obfuscate source app. The process is shown in Figure 2. The first step is to choose the obfuscation target *apk* file. An *apk* file contains an *dex* file and resource files such as XML, images, etc. Because the resource parts do not carry vital program logic, so they are not to be obfuscated by SMOG. Vendors however could encrypt these parts in app's code.

The *dex parser* is one of the core components of the apk obfuscator, it is designed to parse and retrieve the necessary information from the *dex* file. While an Android app is released, the *class* files are converted into a single *dex* file. That means all the instructions of an Android app can be found in its *dex* file. The *dex* file represents the core part of a program, so the main function of dex parser is to retrieve the instructions(Dalvik bytecode) in the *dex* file, and forward them to the permutation handler.

The *permutation handler* is the calculation unit of the apk obfuscator. Each Dalvik instruction has two parts: *opcode* and *operands*. The opcode part specifies the operation to be performed. For each instruction delivered from the dex parser, the permutation handler permutes it by calculation through a permutation matrix. A permutation matrix is an $n \times 2$ matrix on Dalvik opcode set *I*, where n refers to total types of Dalvik opcodes. Each matrix decides a permutation on *I*. It helps to build a bijection (bijective mapping) from Dalvik opcode collection to itself. Calculated through the permutation matrix, each opcode is permuted into another one.

A *permutation matrix* is essentially a rearrangement of the Dalvik opcode collection. The number of the available permutation matrices is determined by the number of Dalvik VM opcode collection n. For instance, in Android 4.1.1, Dalvik's ISA contains about 300 different opcodes, so the number of alternative permutation matrices is more than `300!`. Therefore, the complexity of guessing the matrix is `n!`, and the complexity on reversing obfuscated *dex* file is also `n!`.

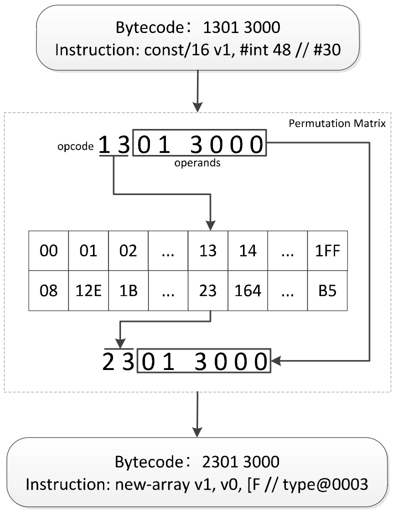

Bytecode: 1301 3000
Instruction: const/16 v1, #int 48 // #30
Permutation Matrix
opcode
1 3 0 1 3 0 0 0
operands
00 01 02 ... 13 14 ... 1FF
08 12E 1B ... 23 164 ... B5
2 3 0 1 3 0 0 0
Bytecode: 2301 3000
Instruction: new-array v1, v0, [F // type@0003

Fig. 3. An example on a bytecode permutation

Figure 3 gives an example of the bytecode permutation process. The Bytecode `1301 3000` which represents the instruction

```
const/16 v1,\#int 48 // \#30
```

turns into another bytecode `2301 3000` after the permutation, which represents instruction of

```
new-array v1, v0 [F // type@0003
```

and the combination of permuted opcodes cannot be interpreted by normal interpreter of original Dalvik VM but the interpreter released by SMOG. Therefore, when an attacker attempts to reverse the obfuscated *dex* file, she could only acquire the permuted *dex* file without any semantic information.

The *dex generator* will be invoked to create the new *dex* file after the permutation is finished. Additional information is appended to the *dex* file to help execute the code, which includes, 1) some specific information to identify if the app is obfuscated, and 2)a checksum value, calculated from the permutation matrix, which helps assign different *dex* files to corresponding interpreters.

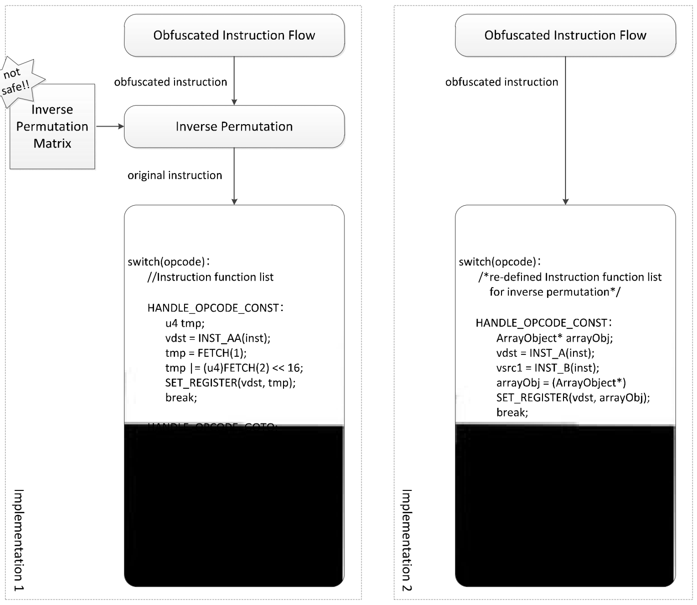
After the parsing, permutation and generation on the *dex* file, the obfuscation engine re-signs and re-packs the *dex* file along with other unmodified resource files, and then releases the obfuscated app.

2) Execution Token: The *execution token* is released by SMOG for target device of the end-user to update the execution environment, and uniquely matches the obfuscated apps. An execution token contains a signed enhanced interpreter and system patches for adding execution policies of the obfuscated app. The mission of the execution token is to install an enhanced interpreter to the target Android Dalvik VM to execute the obfuscated apps. Along with this *apk* file, an execution token is distributed to the end-user to setup an execution environment to ensure the code can only be executed on the target device. After the execution token is installed on the target Android device, end-user could enable the execution environment for the obfuscated app.

The core concept of SMOG is interpretation obfuscation. Instead of downloading and executing a normal app, the SMOG system would allow software vendors pre-process its *dex* file to generate an obfuscated *apk* file to tackle malicious reverse engineering attempts. A normal Dalvik VM cannot comprehend this obfuscated *apk* file. Therefore, the execution token together with the obfuscated app decide the unique execution environment for an end-user.

The core part of the token is the enhanced SMOG interpreter that conducts the mapping from the permutated instructions to the original ones. It is designed to execute obfuscated apps. Therefore, the instructions of obfuscated apps that have been shuffled by the permutation matrix are capable of running by the interpreter. The enhanced interpreter provides a mapping implementation of each Dalvik VM opcode according to the specific permutation we did on instructions. As introduced in section2.1, Dalvik VM uses a single function with many handlers to interpret different opcodes. It would be easier to inverse permutation using the implementation in Figure 4(left), using a reversed permutation matrix before each instruction has been dispatched to their handler. So, the obfuscated instruction flow will revert back to the original flow. However, this method is likely to reveal the token by the reverse engineering means. Therefore, an improved implementation is proposed as in Figure 4(right). Instead of explicitly adding the matrix into the interpreter, we implement the process of the interpretation by redefining the opcode handlers. The content of each opcode handler has been changed according to the permutation matrix. In this case, the attacker have to investigate the whole decompiled code to find out the corresponding opcode, which requires a thorough comprehension of Dalvik VM interpreter and skillful reverse engineering techniques. This to some extent ensures the security of the permutation matrix.

In an typical Android app setup procedure, there are three tasks: installation, optimization, and pre-verification, as depicted in Figure 5. On a normal Dalvik VM, an optimization procedure on the dex file is carried out right after the apps installation. That might cause the obfuscated app unable to be executed. Therefore, to support our obfuscated apps and interpreter to work on an Android device, the original app installation process is to be modified. We patched the normal dex optimization function to work with the obfuscation process. As elaborated in previous sections, the dex generator adds some specific marks and checksum to the dex file. To use these extra information, we make some changes to the Android



```
switch(opcode):
    //Instruction function list

    HANDLE_OPCODE_CONST:
        u4 tmp;
        vdst = INST_AA(inst);
        tmp = FETCH(1);
        tmp |= (u4)FETCH(2) << 16;
        SET_REGISTER(vdst, tmp);
        break;
```

[redacted]

```
switch(opcode):
    /*re-defined Instruction function list
      for inverse permutation*/

    HANDLE_OPCODE_CONST:
        ArrayObject* arrayObj;
        vdst = INST_A(inst);
        vsrc1 = INST_B(inst);
        arrayObj = (ArrayObject*)
        SET_REGISTER(vdst, arrayObj);
        break;
```

[redacted]

Fig. 4. Two Implementations on the Enhanced SMOG Interpreter

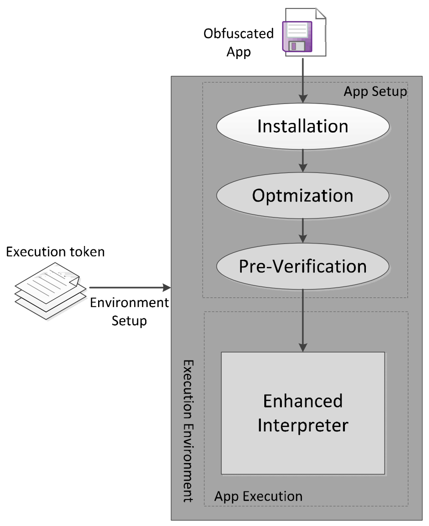


Fig. 5. Overview of the Execution Environment

pre-verification process. Then, the execution environment will choose the proper interpreter, say the SMOG interpreter, to run the obfuscated apps.

The *token generator* is utilized to generate the enhanced SMOG interpreter and modified system source code,then compile them into the final execution token. This execution token will help the end-user setup a correct execution environment to run the obfuscated apps. In this process, the enhanced SMOG interpreter and modified system source code will be compiled into a Dalvik VM core lib file named *libdvm.so*. The *so* file is compiled from Android native code, reverse engineering towards such file is quite hard and that increase the security of permutation matrix. At last, *libdvm.so* will be packaged into execution token with some other verification information.

C. App execution

App execution is conducted by the enhanced interpreter of specific execution environment, referring to Figure 5. Though an obfuscated *dex* file has a specific instruction structure which can not be comprehended by normal interpreter, the enhanced interpreter which has already permuted the opcode interpreting function can guarantee the correctness of executing the app. Given bytecode set *C*, permutation *f*, inverse permutation *f'*, the matrix *M* and the enhanced interpreter EI_M, there exists

$$F'(F(C,M),EI_M) = C$$

So, the obfuscated bytecode will be interpreted as meaningful instructions under obfuscated interpretation.

The enhanced interpreter to support obfuscated interpretation is based on the original portable interpreter. In Dalvik VM the fast and the portable interpreter can be switched dynamically. SMOG takes advantage of this mechanism. SMOG only modifies the portable interpreter of the Dalvik VM. When the obfuscated code is to be executed, the modified Dalvik VM first identifies which part of code is to be executed, and switches to the enhanced interpreter responsible for interpreting the obfuscated code.

The environment setup is a quite simple process. While end-user get execution token through vendors, she is able to use this token to patch her system automatically. After that, the files in the token will replace the same files in the target Android system. Then the setup process is completed. New system not only inherits all the functions of original system, but also has the ability to execution obfuscated apps.

III. Evaluation

In the following, we provide an analysis of security and overhead of SMOG.

A. Security Analysis

Our design contains following security considerations: 1) The obfuscation scheme can resist not only static disassembling but also dynamic reverse engineering. 2) The malicious leakage of obfuscation details hardly affects our scheme's security.

1) Resistance to Reverse Engineering: Static reverse engineering is unavailable to analyze our obfuscated apps because the semantic information is hidden by the permutation. If an adversary wants to perform instruction level analysis, and further to reconstruct the structure of original app and allows it to be executed on other devices, she must understand the permutation first and then reconstruct a *dex* file, which is a manual reverse engineering work and very time-consuming. We use the state-of-the-art reverse engineering tool IDA pro[3] to evaluate the strength of SMOG resisting to static disassembly. While IDA Pro is able to disassemble the original app's bytecode, the result for the obfuscated version shows that the disassembler fails to work. The obfuscated app can resist static analysis effectively because SMOG permutes the bytecode. We further use disassembler(including dexdump, baksmali[4], Dedexer[5], Androguard[6]) and decompiler(including jad[7], jd-gui[8] and ded[9]) to test our obfuscated app. None of these tools can deal with the obfuscated app correctly. The result shows that SMOG makes the standalone static analysis impossible.

To employ dynamic analysis based reverse engineering and deduce the permutation matrix from the interpretation the attacker must overcome two obstacles: First, it is difficult to dynamically analyze the execution environment. The enhanced interpreter is integrated into Android's core runtime – Dalvik VM and is protected with the benefit of device's protection mechanism. Debugging or tampering it is forbidden. Second, even the interpreter is acquired and dynamically analyzed, the reverse engineering of a permuted interpreter is much harder than that of an app. We do not expect our system is perfect secure against advanced analysis, but the discussed defending strategy can thwart most reverse engineering attempts.

2) Resistance to Re-distribution: SMOG generates different interpreters for different devices. Malicious user who wants to re-distribute the app must transplant the execution environment as well. The execution token is dependent on the device's IMEI and pirate execution token can not be installed arbitrarily. Another way is via recovering the secret permutation and then de-obfuscating the app. A brute force searching is obviously unrealistic for the permutation matrix is random chosen by vendor and is kept secret to end-users. But to reverse engineer the execution environment and find the permutation involves first identifying more than 300 functions corresponding to each opcode of the Dalvik VM and then rebuild the permutation. One interpreter contains about 16000 lines of assembly code(ARM) and to the best of our knowledge, there's no automatic reverse engineering techniques for this function identification and permutation recovery process. Thus the manual work is time-consuming. And even the interpreter is understood and the permutation is found, attacker could only break one app's protection among 300! possible permutations. Vendor could change the permutation for another app to defeat the leakage threat.

3) Tamper Proofing: An importance aspect of protection is tamper proofing of the execution environment. SMOG takes advantage of Android's system integrity protection mechanism to prove it. First, the execution token is signed. If the attacker modifies the token, it can't be installed due to the failure of the integrity verification. Second, the enhanced interpreter is a privileged system component integrated to the Dalvik VM. It is protected by system's privilege isolation mechanism, therefore the runtime security is kept.

B. Performance

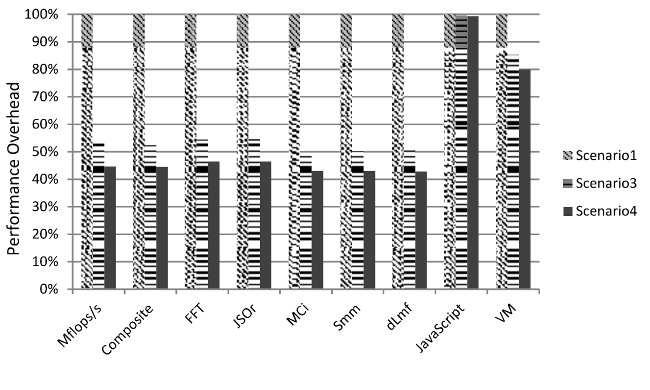
While having the ability to prevent program from being reversed by both static and dynamic analysis technology, any such system would be impractical if the approach induced high overhead. We use suite *0xbenchmark*[10] to test the performance overhead introduced by SMOG. The evaluations are carried out on two both emulator and real mobile device. We use an Android 4.1.1 emulator running on a PC of 2G RAM and a 64-bit Ubuntu (10.04). The emulator was allocated 256MB of memory and 1GB SD card. The authentic mobile device is a Nexus 7 tablet running Android 4.1.2. It has a 1.3GHz Tegra3 CPU and 1GB RAM.

In the evaluation, we carefully monitor various aspects of performance. The computing ability, including *MFlops/s*, *composite*, *fast Fourier transform*, *Jacobi Successive Over-relaxation*, *Monte Carlo integration*, *sparse matrix multiply*, *dense LU matrix factorization* are tested. The garbage collection test is to evaluate the overhead of the VM. And we run *Sun Spider* to test JavaScript performance.

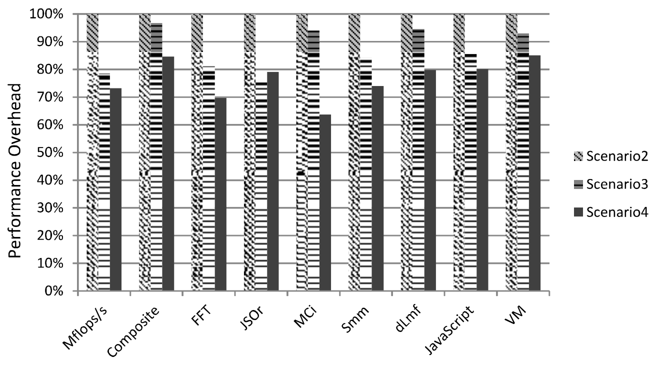
The benchmark suite is conducted to evaluate the executive overhead in three different scenarios.

- Scenario 1: an original *0xbenchmark* is running on an normal Android system with fast interpreter as the performance base.
- Scenario 2: an original *0xbenchmark* is running on an normal Android system with portable interpreter as the performance base.
- Scenario 3: an original *0xbenchmark* is running on an SMOG execution environment.
- Scenario 4: an SMOG obfuscated *0xbenchmark* is running on an SMOG execution environment.

Figure 6(a) is the benckmark result on Nexus 7 tablet, and Figure 6(b) shows the observed value on the official emulator in Android SDK. The performance value of Scenario 1 and Scenario 2 is chosen as the base value for comparison. As we can see in Figure 6(b), there is no significant performance loss on the emulator. In contrast, Figure 6(a) suffers greater performance loss due to the performance advantages of fast interpreter on real device. The performance on SMOG execution environment of mathematical operation, such as Mflops/s, Composite, FFT, JSOr, MCi, Smm and dLmt, are close to 50% slowdown on Nexus 7, while the JavaScript is less than 1% and VM less than 20% performance loss.



(a) Nexus 7



(b) Emulator

Fig. 6. Performance Overhead of SMOG

We notice that the mathematical benchmark involves dense arithmetic operations, so it goes through more permuted instructions than the JavaScript and VM benchmark tests. While running the VM test, on the contrary, invokes many external functions which is not the obfuscation target of SMOG. So the performance slowdown by SMOG depends on the proportion of these instructions in the app. The performance loss in VM and JavaScript testing results reveal that our system will not bring inconvenience to end-user experience, but only becomes noticeable while doing a large amount of computation.

Running a program on the SMOG execution environment(Scenario 3 and Scenario 4) is always slower than on an original Android system(Scenario 1 and Scenario 2). The performance slowdown is caused by SMOG execution environment's modification to system. There also exists the performance gap between Scenario 3 and Scenario 4. The SMOG execution environment costs about 5% more performance in dispatching the incoming bytecode to the proper interpreter.

IV. Conclusion

This paper proposed a novel Android app protection system SMOG, based on obfuscation interpretation, which makes it difficult for an attacker to reverse engineer the obfuscated app.

We have described the architecture of an Android apps protection system SMOG to enhance the protection on the valuable or patent algorithms inside the Android apps. It requires an integration of many mechanisms including an obfuscation engine and an installation of an execution environment. The obfuscation engine is responsible for releasing an obfuscated app and making it difficult for an attacker to conduct reverse engineering, and a corresponding execution environment on the end-user's device is provided to run the obfuscated apps.

The security analysis results proves that the obfuscation engine provides the obfuscated apps the resistance to reverse engineering. Besides, using experiments on both real device Nexus 7 tablet and Android emulator the SMOG execution environment correctly executes the obfuscated app with approximately 5% performance loss.

References

[1] D. Bornstein, "Dalvik vm internals," in *Google I/O Developer Conference*, vol. 23, 2008, pp. 17–30.

[2] "Code and documentation from android's vm team," http://code.google.com/p/dalvik/.

[3] Hex-rays, "Ida pro," http://www.hex-rays.com/.

[4] "smali - an assembler/disassembler for android's dex format," https://code.google.com/p/smali/.

[5] G. Paller, "Dedexer," http://dedexer.sourceforge.net/, 2012.

[6] A. Desnos and G. Gueguen, "Android: From reversing to decompilation," 2011.

[7] "Jad java decompiler," http://www.varaneckas.com/jad/.

[8] "Java decompiler," http://java.decompiler.free.fr/?q=jdgui.

[9] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX Security Symposium*, vol. 2011, 2011.

[10] 0xBench, "Comprehensive benchmark suite for android," http://code.google.com/p/0xbench/downloads/list.