DexPro: A Bytecode Level Code Protection System for Android Applications

Beibei Zhao†, Zhanyong Tang†*, Zhen Li†, Lina Song†, Xiaoqing Gong†,Dingyi Fang†, Fangyuan Liu†, Zheng Wang‡
†School of Information Science and Technology, Northwest University, P.R. China.
‡Metalab, School of Computing and Communications, Lancaster University, UK.

***Abstract*—Unauthorized code modification through reverse engineering is a major concern for Android application developers. Code reverse engineering is often used by adversaries to remove the copyright protection or advertisements from the app, or to inject malicious code into the program. By making the program difficult to analyze, code obfuscation is a potential solution to the problem. However, there is currently little work on applying code obfuscation to compiled Android bytecode. This paper presents DEXPRO, a novel bytecode level code obfuscation system for Android applications. Unlike prior approaches, our method performs on the Android Dex bytecode and does not require access to high-level program source or modification of the compiler or the VM. Our approach leverages the fact all except floating operands in Dex are stored in a 32-bit register to pack two 32-bit operands into a 64-bit operand. In this way, any attempt to decompile the bytecode will result in incorrect information. Meanwhile, our approach obfuscates the program control flow by inserting opaque predicates before the return instruction of a function call, which makes it harder for the attacker to trace calls to protected functions. Experimental results show that our approach can deter sophisticate reverse engineering and code analysis tools, and the overhead of runtime and memory footprint is comparable to existing code obfuscation methods.**

***Index Terms*—Code obfuscation, Reverse engineering, Decompile, Opaque predicates**

I. INTRODUCTION

Unauthorized code reverse engineering is a major concern for Android application developers. This technique is widely used by adversaries to perform various attacks, including removing copyright protection to obtain an illegal copy of the software, taking out advertisement from the app, or injecting malicious code into the program. By making the program harder to be traced and analyzed, code obfuscation is a viable means to protect applications from unauthorized code modification.

A number of code obfuscation approaches have been proposed to protect applications against reverse engineering [1], [2], [3], [4]. Most of the prior work perform code obfuscation on high-level programming languages such as Java and require access to the program source code. However, this requirement has two major drawbacks: (1) source code level code obfuscation provides limited protection as the obfuscated code can removed or optimized out by the compiler; (2) many developers are not willing to disclose their source code. As such, a code protection technique performing on the compiled bytecodes or binary with stronger protection is highly attractive.

The first effort in this direction is SMOG [5] that performs code obfuscation by permuting the instruction opcodes from the compiled Dex bytecode[1]. The permutated opcodes are then interpreted at runtime through a modified VM interpreter. While promising, there is a significant shortcoming of this approach. Programs protected by SMOG must run in a dedicated VM other than the native Android runtime environment, which limits the application of SMOG at larger scale.

In this paper, we present DEXPRO, a novel bytecode level code obfuscation system for Android applications. Unlike prior approaches, our method performs on the Android Dex bytecode and does not require access to high-level program source or modification of the compiler or VM. DEXPRO advances prior work in the following ways. Firstly, DEXPRO performs code obfuscation on the bytecode level, so it does not require accessing to the source code and as a result the obfuscated code will not be optimized out by the compiler. Secondly, it requires no modification to the compiler and runtime environment. Hence the obfuscated code can run on any environment that supports the standard Android bytecode format. DEXPRO exploit two key structures of the Android Dex bytecode definition to protect the program against dynamic and static code analysis, which is explained in Section 6.2: (1) all except for floating operands are based on a 32-bit register and (2) the instruction follows a function call is always used to retrieve the return value of the function. Our approach utilizes the register structure of Dex to pack two 32-bit operands into a single 64-bit data item, so that any attempt in decoding the protected operands will receive incorrect information. We leverage the calling conversion of Dex, to insert opaque predicates[6], [7], [8] (i.e. code with complex logic but does not get executed) between instructions of the function call and return value retrieval. Doing so not only makes it harder for the attacker to obtain the return value, but also obfuscates the dynamic program behavior. By combining these two techniques, DEXPRO provides stronger protection when compared to existing code obfuscation techniques that

*Corresponding author. Email address: zytang@nwu.edu.cn

[1]The Dalvik executable format (Dex) is the executable binary format for Android applications. It was originally designed for the Dalvik VM. It remains to be used as a standard bytecode format for Android applications after the Dalvik VM is replaced by the Android runtime (ART).

target at the source code or bytecode level, with little extra overhead.

The core concept of DEXPRO is obfuscating the access procedure of variables in the registers. But the verifier component of Android runtime system(*CodeVerify.app; DexVerify.cpp*)[9], [10] is referred to as VFY by Dalvik VM, which verifies the type of registers when loading the class. So our obfuscated application is not normally running. It is the challenge in this paper. In order to solve the problem, we use the Dex dynamic loading technology and Dalvik runtime tampering technology, which is explained in Section 4.3.

We have evaluated our approach by using DEXPRO to protect a number of representative Android application operations. Experimental results show that our approach can protect software against sophisticated reverse engineering tools, including *Jeb*[11], *dexdump*[12], *IDA pro*[13] and *Dex2jar*[14] with less than 5% increment in code size and runtime overhead. This paper makes two specific contributions:

- It is the first work to exploit the register structure and calling convention of Android Dex for code protection;
- It is the first byte-code level code obfuscation scheme that protects software against static and dynamic code analysis.
- The obfuscation method is evaluated from potency, resilience, availability and cost.

Structure of the paper: We provide background in Section 2. Section 3 presents the overview of our system and Section 4 introduce the process of system implementation in detail. In Section 5, we discuss the potency, resilience and cost of the obfuscation method, and in Section 6 we deliberate evaluation of code obfuscation against current popular reverse tools and the overhead. Most relevant work in Android code obfuscation is discussed in Section7. Finally, the concluding remarks are given in Section 8.

II. BACKGROUND AND ATTACK SCENARIO

In this section we give some detailed background information and then describe the process of decompiled.

A. Dalvik Virtual Machine

Dalvik virtual machine is one of the core parts of the Android mobile device platform. Android applications are written mostly in Java, but run in the DVM.The DVM run-time environment is introduced as follows.

Application Structure. Android applications are shipped as a single Zip archive named with the .apk extension by convention. The following mainly contents can be found within common APKs:1)The classes.dex file holds all Dalvik executable bytecode. 2)The AndroidManifest.xml file requested permissions and the application components. Java applications are composed of one or more .class files, one file per file. Then the Dalvik dx compiler consumes the .class files, recompiles them to Dalvik bytecode, and writes the resulting application into a single .dex file. A.dex file contains multiple Class Definitions each containing one or more Method definition each of those being linked to Dalvik bytecode instructions present int

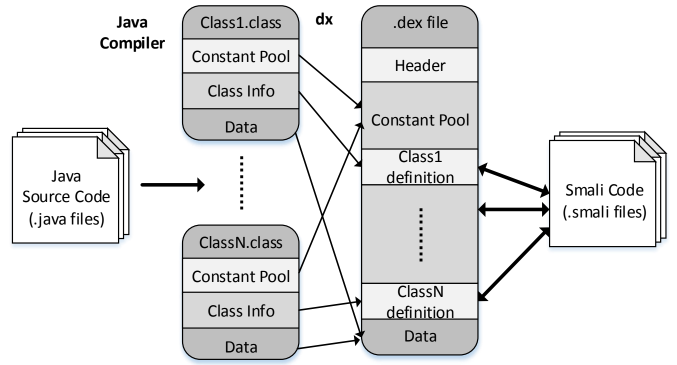

Figure 1: Compilation process for DVM applications.

the Data secion. Figure 1 provides a conceptual view of the compilation process for DVM application, meanwhile, each class within the .dex file corresponds to a smali file.

Register architecture. The DVM is register-based, whereas existing JVMs are stack-based. Dalvik bytecode assigns local variables to any of the 2 16 avalible register. The Dalvik opcodes directly manipulate registers, rather than accessing elements on a program stack.

Instruction set. The Dalvik bytecode instruction set is substantially different than that of Java. There are 237 opcodes present in the Dalvik opcode constant list[15]. The set of instructions can be divided between instructions which provide the type of the registers they manipulate.

Constant pool structure. A.dex file contains four homogeneous constants pools: for Strings,Class,Fields and Methods.It is shared by all the class and has been modified to only use 32-bit index in order to simplify the interpreter.

The process of the DVM execution. When an app installed, every installed application gets its own unique user ID by default. This means that every application will be executed as a separate system user. When an Android application is executed by the DVM, the process[3] is interpreted as follows. In detail, the specific process is as follows:

- **Load a class**. when the DVM loads a class by the function loadClassFromDex(), the class will have a ClassObject type of data structure in the runtime environment. DVM stores all loaded classes by using gDvm.loadedClassed global hash table;
- **Verify Bytecode**. Bytecode verifier verifies the loaded class by using function dvmVerifyCodeFlow();
- **Find the main class**. The DVM searches the main class in gDvm.loadedClassed global hash table by using function FindClass( ). If it can't find the needed class, it will go back to load the class;
- **Execute bytecode flow**. Interpreter is initialized by invoking function dvmInterpret( ) and then the Dalvik bytecode flow will be executed.

B. Reverse tools

Reverse engineering is a complex and lengthy task. Use the right tools for the job and cut down on expensive man-

hours. In the following, we present a selection of tools that can be used to disassemble Dalvik bytecode. Futhermore, meta information, e.g. method identifier and string constants, about the program structure can be gathered, which helps to identify interesting parts of an application.

At present, There are many reverse tools including *dexdump, IDA pro, Dex2jar, Jeb, and AndroidKiller e.g.* The first three are only static analysis tools. Obviously IDA PRO can also make experiment about dynamic analysis, but for the native local mainly. The last can statically and dynamically analyze Android applications, goodware or badware, small or large. They will be discussed: *dexdump* is a tool that is directly included with Android SDK. It is a basic dex file dissector and can also disassemble Dalvik bytecode, which is stored in the dex file. *dexdump* uses a trivial and straight forward approach in order to disassemble a dex file. It uses linear sweep to find instructions, this means that *dexdump* expects a new valid instruction behind the last byte of the currently analyzed instructions. *IDA pro* is well known as a powerful tool for reverse engineering. It also supports many architectures as well as Dalvik bytecode. A very helpful feature of *IDA pro* is presentation of the Dalvik code as a graph. This makes it much easier for an analyst to follow the control flow within a program. *Dex2jar* will turn .dex file into .class files. It makes analysis clearer and celerity. *Jeb* for Android, its extensible nature allows reverse engineers to perform disassembly, decompilation, debugging, and analysis of code and document files, manually or as part of an analysis pipeline. *AndroidKiller* is an integrated tool for decompiling and dex2jar and phone signatures re-packaging, which is very convenient for reverse analysts.

C. Attack scenario

There are a lot of dex opcodes within a Android application and they are very difficult to understand. So attacker required access to source code rather than operate on the Dex opcodes to identify key-positives resulting from automated code analysis, e.g., perform manual confirmation.

The initial stage of decompilation retargets the application .dex file to Java classes[16]. Figure 2 shows this process. The first and most important step is recovering typing information. However, the Dalvik bytecode has two generalized cases where variable types are ambiguous:1)constant and variable declaration only specifies the variable width(e.g., 32 or 64 bits), but not whether it is a float, integer; and 2)comparison operators do not distinguish between integer and object reference comparison. Reverse tools determine unknown types by observing how variables are used in operations with known type operands. they also infer register types by observing how they are used in subsequent operations with known type operands. Dalvik registers loosely correspond to Java variables. because Dalvik bytecode reuses registers whose variables are no longer in scope, we must evaluate the register type within its context of the method control flow, i.e., inference must be path-sensitive.

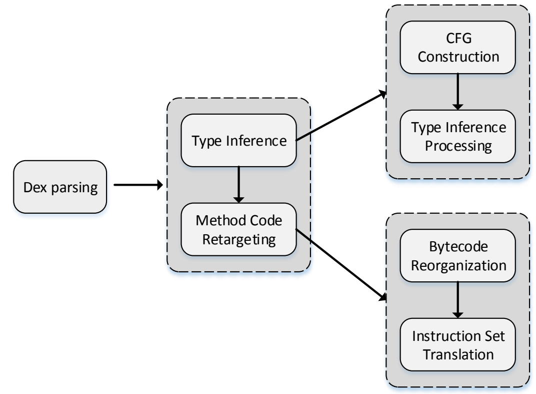

Figure 2: Dalvik bytecode retargeting.

III. Overview of Our Approach

There are two Android virtual machine and the DM was replaced by the Android runtime(ART)[17]. The difference between them is that ART virtual machine will translate the dex bytecode into native machine code when loading the class, so both virtual machines execute the same unmodified Dalvik bytecode file structure, instruction formats(smali), and constrains. Hence, our proposed obfuscation applies to apps run by both DVM and ART. But The difference between them is that the ART is not dex bytecode after loading the apps, and we can't bypass the verifier through backfilling the obfuscated bytecode after it. We should think of an approach to prevent the verifier and decrease the performance overhead as well.

As depicted in Figure 3, the system is proposed as a software protection system within which an obfuscation engine and solving the conflict problem. It mainly obtain three part.

At the first part, we are mainly introduce to how to obfuscate the smali code. The basic idea is confusing the data-flow for the access procedure of register data, and combining opaque predicates technology to confuse the control-flow.

The next step, because our obfuscation method would cause problems of the register-type conflict, we should make the executable .dex file normally execute by the Dalvik VM. So we need to modify the .dex file ,which is explained in Section 4.3.

Lastly, we will utilize the dex dynamic loading technology to load the above .dex file, but some bytecode is nop after the class being loaded, so Dalvik runtime tampering technology is used to solve the problem. Meanwhile, To illustrate this, we firstly analyzed the exact nature of the problem, and then describe the details process that is to implement normal running.

IV. Implementation details

Our proposed schemes obfuscate Android apps at Dalvik bytecode level to achieve an expressive control-flow and data-flow obfuscation.Within this paper, our approach combines obfuscation techniques against static and dynamic reverse engineering, which is discussed next.

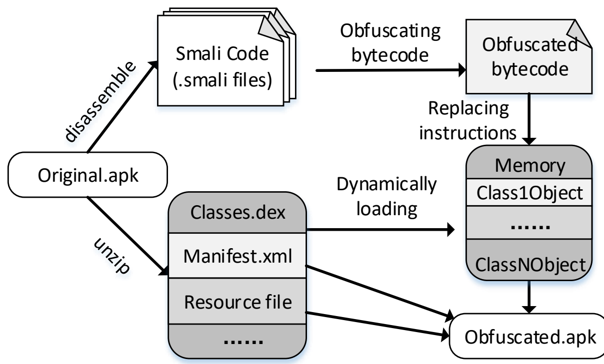

Figure 3: Workflow of the DEXPRO system. we can see three part, obfuscate the smali code ,nop bytecode within the .dex file, and dynamic load and runtime tamper, which is the background of the light-green.

A. The process of obfuscation

Dalvik VM's instructions based on the architecture of register and the constant pool is modified to use the index of 32 bits, which can simplify the interpreter. Therefore the constants within smali code are stored in the 32-bit registers. As shown in the Figure 2, Analyzing CFG and register type inference processing is very important to retarget the application .dex file to Java classes. According to this characteristic we can confuse the type of registers and control flow, they are data flow obfuscation and control flow obfuscation.

Dalvik VM gets the operand from the corresponding register directly based on the type of operation code when performing bytecode flow, regardless of what the type of register is, so we can obfuscate it from two aspects. On the one hand, because the constants of the long and double in smali code are 64-bit, they are stored in the two adjacent 32-bit registers, so we can use a long constant instead of the two constants respectively stored in two 32-bit registers . On the other hand, we will confuse a instruction of capturing the return value of an object reference for a instruction of capturing the return value. The two methods on the above are to confuse on the data access, so that they can make the compiler tools puzzled.

Confusing the control flow is an effective way. The principle is to use various technical means to hide or modify the real control process of a program, to prevent the attacker from analyzing the control flow[18]. Inserting opaque predicates is one of the effective obfuscation.

In this paper, we propose a obfuscation method by inserting reinforced opaque predicates[2]. The first step is to judge the position where to insert, which was conducted on the basis of process analysis. Invoking the key function(*self-define*) is very important modules of program, so we can insert reinforced opaque predicates into the place where is between the instruction of invoking the key function and the accessing returned value, and the process is shown in Figure 4. For example, function of purchasing equipment function is very critical in the game and its returned value is usually analyzed

```
.method pubilc test()
    ......
    invoke-virtual {p0, v0, v1},
    Lcom/example/test/MarketLogic;
    ->payCallback(II)I

    inject opaque predicates

    move-result v4
    ......
.end method
```

Figure 4: Control obfuscation. In order to obfuscate the control-flow, we will inject the opaque predicates between two instructions that are invoking and capturing the return value.

by the attacker. If the developer takes such protection method, the attackers will not get the return value. This way increase the complexity of control flow and data flow.

B. Register type-conflict problem

This section elaborates an important problem caused by the Android runtime system, which we encountered while implementing the described obfuscation techniques. It also explains how we modified target apps prior to apply the obfuscation techniques so as to prevent the problem and satisfy the Android runtime system.

Because the Verifier module[10], [9] seems to perform a register liveness analysis in each method prior to running it, and ensure that there is no type conflict among live registers at an program point.

Firstly, at each program point, it keeps track of live registers together with the data type they hold. Should it find a register with two different types at the same program point, it will report the register-type conflict problem as shown follows and the running app will crash.

```
VFY: register1 v0 type 19, wanted 17
VFY: register1 v1 type 20, wanted 17
VFY: rejecting opcode 0x90 at 0x0008
VFY: rejected Lcom/example/.../...;
    .add()I

 Verifier rejected class Lcom/example/
    .../...;
 Class init failed in newInstance call
 (Lcom/example/.../...;)

 Shutting down VM
```

Secondly, In order to exactly get the return value of the child function which is called, the instruction used to retrieve the return value of the function does not follow a function call. Our approach insert the reinforced opaque predicates into the two instructions. So it will report the conflict information as follows.

```
VFY: copyRes1 v3<-v7 cat=1 type=0
VEY: rejecting opcode 0x0a at 0x0008
VEY: rejected Lcom/example/.../...;.Test()V
VEY: Verifier rejected class Lcom/example/
```

```
    .../...;
Verifier rejected class Lcom/example/
Class init failed in newInstance call (Lcom
/example/.../...;)

Shutting down VM
```

From the error information, we can come to the conclusion that the validation fails is reported during the static initialization of the class. It is a new solution that Obfuscated Dalvik bytecode are backfilled in the memory after initializations. However, we now know that when Dalvik VM performs the bytecode flow with in the dex file, it gets out the operand from the corresponding register in terms of the opcode, regardless of what the type of register is. Given this issue, we thus need to find a solution that will bypass the verifier of the evaluated Android runtime systems from deducing any false type conflict in otherwise correctly obfuscated apps.

C. App execution

As described in Section2.1, Dalvik VM verifies the validity of instructions when loading the class, for example, verifying the type of the registers. so executing an obfuscated app must bypass the process of the verifier.To address the problem, we can do the following three things.

Firstly, we will recompile the obfuscated smali into a .dex file. Then we will apply for a memory space to store the obfuscated bytecode, which is in the .dex file can be stored in the ObjMethod structure of showing in Figure 5. At last, we will fill the junk bytecode in the obfuscated method and get a new.dex file. When attackers statically analysis it, they will get many junk code, so this way resist static analysis successfully.

Secondly, in order to load the executable file new.dex utilizing the technology of loading dex dynamically, we develop our own customized Classloader function to load all the classes in this dex file, before which the default Classloader in the API layer should be replaced to ensure the normal execution of the real dex file.

There exists a system component called Application[14] in Android frame. it will initialize several global variables when establishing Application(i.e. before launching app), and all the Activity within the same app can access the value of these variables. Usually, system will develop an Application automatically and we don't need to develop it specifically, so through designing customized class ProxyApplication that extends Application, the default Classloader in the system will be replaced by the customized MyDexclassloder when initiating ProxyApplication. Moreover, we should configure ProxyApplication in the ApplicationManifest.xml file as follows:

```
<application
    Android:name="ProxyApplication"
</application
```

Lastly, in order to bypass verification of Dalvik VM for instructions(i.e. register types), we utilize the technology of dynamically loading. As bytecode in the obfuscated methods

```
typedef struct {
    String ClassName;
    String MethodNmae;
    Char* newCode;
}ObjMethod;
```

Figure 5: The structure of ObjMethod. It is used to store the bytecode of the obfuscated method within the .dex file. The bytecode backfill the memory after the class loaded.

in new.dex is a zero sequence after loading new.dex, we need to fill the obfuscated bytecode in the corresponding location in the memory before execution.

When running an Android application, it will load dex file and parse the dalvik bytecode that is executed by Dalvik VM. We reads or writes the byte flow when running the app with the aid of the JNI component[19], as there isn't a direct method to achieve this compared to X86 and ARM frame due to limited instruction sets of DVM. Native library and Dalvik VM owe the same priority as they are in the same process, so we can utilize JNI to call methods in native library to achieve the modification for Dalvik bytecode.

The address of dex in the memory after loading can be acquired through DexFile class in Android system. According to the address, we can acquire the memory address of the obfuscated methods via parsing the dex structure, then fill in the corresponding memory address with the corresponding bytecode stored in the ObjMethod structure to ensure correct system execution.

V. OBFUSCATION EVALUATIONS

A. Evaluations criteria

Collberg[20] gives an accurate definition on code obfuscation: The i is one of all possible inputting set *I* which is in the program *P*. If and only if it is $\forall$ *i:T(P)(i)=P(i)*, we can just think that the transition of the confusion was correct. The obfuscation method is evaluated from potency, resilience and cost by Collberg at al.

Let *T* be a behavior-conserving transformation, such that $P \xrightarrow{T} P'$ transforms a source program *P* into a target program *P'*.

$T_{pot}(P)$, the potency of *T* with respect to a program *P*, is a measure of the extent to which *T* changes the complexity of *P*. Let *E(P)* be the complexity of *P*. It is defined as

$$T_{pot}(P)=E(P')/E(P)-1 \quad (1)$$

T is a potent obfuscating transformation if $T_{pot}(P)$¿0.

$T_{res}(P)$ is the resilience of *T* with respect to a program *P*. $T_{res}(P)$=*one-way* if information is removed from *P* such that *P* cannot be reconstructed from *P'*. Otherwise,

$$T_{res}(P) \triangleq Resilience(T_{Deobfuscaoreffort}, T_{Programmereffort}) \quad (2)$$

App		Number of instructions before obfuscation	Number of instructions after obfuscation
ele_me	online ordering	24	12
fileexplorer	Mi file manager	72	36
photup	photo album	30	15
v2ex-daily	exchange community	16	8
zhihupaper	zhihu daily	16	8

Table I: Count the number of instructions of meting obfuscation criteria.

T_{cost}*(P)* is the extra execution time/space of *P'* compared to *P*. that is to say

$$T_{cost}(P)=Cost(C_{time}\ ,\ C_{size}) \quad (3)$$

B. Evaluations obfuscation method

Abouting code obfuscation techniques, Collberg put forward three evaluation indexes: the potency, resilience and cost, We will Make qualitative evaluation for the confusion in this article according to the evaluation index.

Potency. The proposed code confusion method includes both control obfuscation and data obfuscation. In data obfuscation, we can obfuscate the instructions of accessing the values from the registers and calling method that its returned value is object type. The experimental result as shown in Figure 6,first, we confuse two 32-bit constant definition instructions into a 64-bit constant definition instruction, when the attackers are using reverse tools to reverse,such as *dexdump Dex2jar Jeb*,the getting result is wrong.In control obfuscation, in order to resist attacker to access the returned values which is after calling the key function, we will insert the opaque predicate between two instructions that are calling and accessing returned value. The above two kinds of method can both resist the attacker to acquire correct program code and increase the complexity of the control flow. Thus, the proposed method has good strength.

Resilience. The obfuscated program has been analyzed in terms of reverse analysis. For example, the attacker can't access correct instructions and analyzing logical construction of program. As shown in Figure 7, decompilation process is failure and control flow becomes very complicated, so the proposed method has strong slastic.

Cost. The function's time complexity is O(1) after being obfuscated. Firstly, inserting opaque predicate and modifying the instruction format does not change polynomial of the complexity. Secondly, dex dynamic loading and backfilling the bytecode both increase little time, as shown in Figure 10. In space consumed, we will only insert opaque predicate into invoking the key function and obfuscated instructions is short. Thus this method has little effect on the time and consumed space.

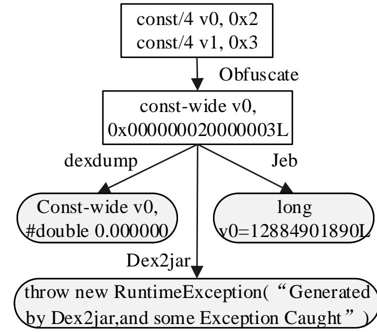

Figure 6: Two 32-bit constant definition instruction confusion into a 64 - bit constant defined instruction,get the wrong results from reverse tool *dexdump, Dex2jar, Jeb*

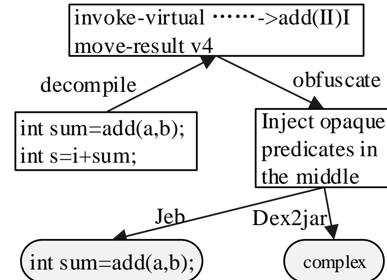

Figure 7: The case of reversing engineering. When we inject the opaque predicates after invoking the *add()* method, the attacker can't capture the return-value *sum*.

VI. Experiments and analysis

A. Experimental Setup

Hardware: Our approach is evaluated using a Xiaomi 4 phone, which runs the Android 4.4.4 operating system. The device has a processor: a four-core processor at 2.5 GHz. The device has 2 GB of RAM and 16 GB of internal storage.

Benchmarks: Based on the deep research inDexPro, as shown in Fig.2, this paper designed and implemented a prototype system. It can resist the static analysis and dynamic analysis of the process of the reverse engineering. Our approach is evaluated by using the developed tool to protect a set of representative operations of typical Android applications.

In order to prove the practicability of our protection method, we selected five popular apps downloaded from Android Open Source Project,switch can represent different types in the application market, Choosing different function applications can show the feature that exist instructions of defining 32-bit variables preferably in the application market, so it is different

Test case	The characteristics of the instruction
Data.apk	Defining two 32-bit variables
BubbleSor.apk	Returned value of the reference type
Hanio.apk	key function(self-defined)
Contact.apk	All the above

Table II: A list of apps we selected to test. Column 2 represent the type of instructions of four typical Android apps.

with the next experiment cases. then count the instructions of meting obfuscation criteria that is two adjacent instructions of defining 32-bit variables. So the number of the instruction of this type would decrease 50 percent. It is listed in Table I. We found that, in all tested apps,there are many instructions meting the conditions of obfuscating some application.

Performance overhead is also very important about code obfuscation and our protection method applied user-definable key functions. So measure the performance overhead and effectiveness of DexPro, We have implemented these operations in four applications (listed Table II). We chose these APP for the following reasons: Data.apk is a constant definition; BubbleSor.apk is a bubble sorting algorithm; Hanoi.apk is a typical representation of the Hanoi algorithm; Contact.apk has all of the above instruction features, Each representing a different character,switch can show effectiveness and performance overhead betterly. So these test cases can completely represent our obfuscation engineering and be tested to evaluate from the effectiveness and performance overhead.

B. Effectiveness

The main purpose of this paper is to protect the executable code of the application, and make it not be easy analyzed by attacker. The below from two aspects of resisting static and dynamic analysis analyze the effectiveness of the protection method based on confusion of smali code.

The some bytecode within executable file is a junk code sequence before executed bytecode flow by DVM. So when attackers want to statically analyze the logical structure of the program, they can't acquire correct bytecode. Before the app running the attackers dump the bytecode through dynamically analyzing, this way is the same with resisting static analysis. If running, they will dump the obfuscated bytecode, which isn't correctly decompiled, as shown in Figure 4.

In order to the effectiveness of the obfuscated method, we will also utilize reversing tools to reverse executable code of the application. The experiment results is shown in Table III.

C. Performance overhead

We measured the performance overhead of DexPro by comparing the .dex file size, memory use and launch time of applications before and after obfuscated by DexPro. We measured the launch time by capturing the timestamps of logs output by Android Logcat. The size of memory use was measured by using the command procrank, which can get the memory usage of the current system process and read

Test case	Jeb	Dex2Jar	dexdump	IDA pro
Data.apk	×	×	∗	∗
BubbleSor.apk	×	×	∗	∗
Hanio.apk	×	∗	×	∗
Contact.apk	×	×	×	∗

Table III: The results of analysis by reversing tools. × represent that the code after reversing is wrong. ∗ represent that it becomes very complicated

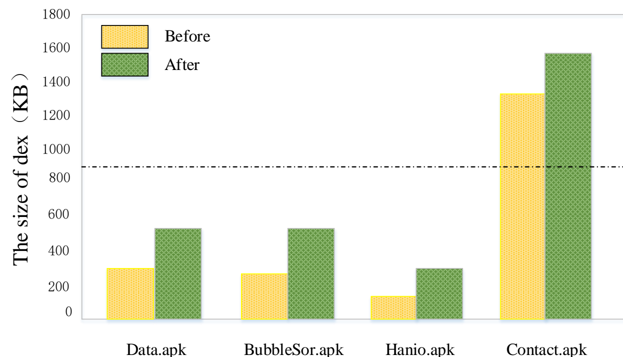

Figure 8: Describe the changes of the .dex file size. Compare to the D-value of the .dex file size before and after obfuscated.

information from the */proc/pid/maps* to count them, including PSS, USS, VSS, RSS.

We can see from Figure 8 that the dex file size increases more than 13KB. This increase in file size is mainly due to the packer dex file that is used to load really dex file and inserting the opaque predicates. The results are not obvious, because the application size we selected is relatively small and little difference.

Figure 9 illustrates memory use changes of applications before and after obfuscated. The memory usage increases by an average of 1.95M. We will describe that memory costs are mainly caused by the re-packaging dex file, storing obfuscated bytecode, and native library. But current mobile devices tend to provide more RAM, for example, Huawei Nexus has two processors: a four-core ARM Cortex-A57 processor at 1.95 GHz and a four-core ARM Cortex-A53 processor at 1.55 GHz. The device has 3 GB of RAM and 64 GB of internal storage. So you can ignore the costs.

To measure the app response delay after being protected by DexPro, we compared the app launch time for the first, second, third, and fourth run before and after obfuscated. To make it more precisely, we did each experiment 20 times and used the average value as a final reference. The launch time of four test case is listed in Table IV.

Figure 10 shows the app launch time changers at each time. We can see that the time increase at the first launch, which is mainly caused by: 1) running the obfuscated application needs to replace Classloader; 2) Before running we must find

Test case		Data.apk	BubbleSor.apk	Hanio.apk	Contact.apk
First(s)	Before	2.175	2.249	2.124	2.912
	After	4.027	4.612	4.677	5.412
Second(s)	Before	0.939	0.923	0.844	0.911
	After	1.027	0.942	0.892	1.017
Third(s)	Before	0.845	0.882	0.797	0.891
	After	1.171	0.959	1.024	1.238
Fourth(s)	Before	0.823	0.862	0.781	0.887
	After	0.994	0.928	0.934	1.099

Table IV: Launch time(in seconds) of four apps for the four times launch before and after obfuscated.

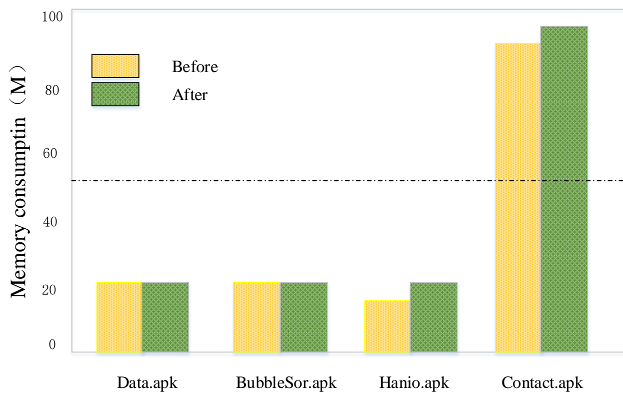

Figure 9: Describe the changes of memory use. Compare to D-value of memory use before and after obfuscated.

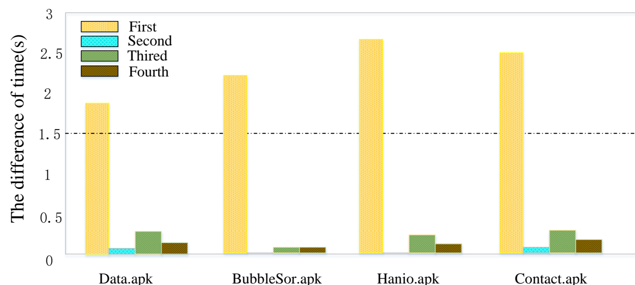

Figure 10: Describe the changes of launch time D-value. Compare to the first ,second, third ,and fourth launch time D-value before and after obfuscated. we can see only that the time significantly increase at the first launch

the obfuscated method in the corresponding memory address through parsing the structure of the dex file and then fill the bytecode that is stored in ObjMethod in the address. These classes and libraries will be loaded at the application's first execution and will be kept in the memory so long as the system resources are sufficient. Thus, they do not need to be loaded again in the app's latter launch, which is the reason why they all show a similar launch time as the unprotected apps at the latter launch.

VII. Related work

Obfuscation is a useful and cost effective technique and it does not require any special execution environment. Moreover it is believed to be more effective on Android system[21], [22]. Patrick Schulz in his work code Protection in Android[23] discusses some possible code obfuscation methods on the Android platform using identifier mangling, string obfuscation, dead code insertion, and self modifying code. Ghosh et al.[24] have discussed a code obfuscation technique on the Android platform that aims at increasing the complexity of the control flow of the application so that it becomes tough for a reverse engineer to get the business logic performed by an Android application. Kundu has also worked on some obfuscation techniques like clone methods, reordering expressions and loops, changing the arrays and loop transformations[25].

There are various Android obfuscation tools available in the market, such as Proguad[4]. But the current Android obfuscation tools seem to still lack the combination of complex control-flow and data-flow obfuscation techniques. In [1], [2], [3] authors presents confusion scheme and algorithm of Android oriented software Java code, combined with the algorithm and improved insertion branch path and flattening the excess flow of control of these two kinds of control flow obfuscation method.

Junliang Shu et al. proposed SMOG[5], a comprehensive executable code obfuscation system to protect Android app. 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. SMOG will also modify the code of DVM interpreter. Noor et al.[26] present a protection scheme based on obfuscation, code modification and cryptographic protection that can effectively counter reverse engineering.

Vivek Bala et al.[27] analyzed the need for potent control-flow based obfuscation so as to help protect Android apps. they also have described the design and implementation of three control-flow obfuscations for Android apps at the Dalvik bytecode level, which go beyond simp control-flow transformations used by exiting Android obfuscators. The register-reuse conflict problem raised by the Android runtime system has also been addressed by means of our type separation technique.

VIII. Conclusion and Future Work

To deal with the problems that Android application is easily to be tempered and repacked, this paper proposes a protection method based on obfuscating smali code. We have analyzed the related work about protecting the code of Android apps in recent year. We also have described the design and implementation of confusing the data flow for the access procedure of register data, and combining opaque predicates technology to confuse the control flow. This method can resist decompiling. Meanwhile, we evaluate the potency, resilience and cost of the code obfuscation method. Experimental result demonstrates excellent performance of the effectiveness and low cost.

The protection method presented in this paper has some limitations as well. It can only be used on some specific instructions formant and not be used at large scale. Our proposed method does not protect the resources and native library .so of an Android app and only protects the dex file.

In the future work, we will utilize the technology of control-flow platting to confuse the control-flow of apps and protect native library .so to strengthen the protection and increase difficulty in reverse engineering. More importantly, the obfuscated method is extended to the ART VM.

Acknowledgment

This work was partially supported by the National Natural Science Foundation of China under grant agreements No. 61672427 and No. 61572402; the International Cooperation Foundation of Shaanxi Province, China under grant agreements No.2015KW-003 and No.2017KW-008; the Research Project of Shaanxi Province Department of Education under grant agreement No.15JK1734; the Service Special Foundation of Shaanxi Province Department of Education under grant agreement No.16JF028; the Research Project of NWU, China under grant agreement No.14NW28; the UK Engineering and Physical Sciences Research Council under grants EP/M01567X/1 (SANDeRs) and EP/M015793/1 (DIVIDEND); and the Royal Society International Collaboration Grant (IE161012).

References

[1] X. A. Zheng Qi, "The control flow of confusion for android mobile application," 2014.

[2] L. Jinliang, "Research and realization on android software protection technology," Ph.D. dissertation, Beijing University of Posts and Telecommunications, 2015.

[3] Z. Qi, "Research and implementation of code obfuscation algorithms for applications of and smartphone terminal," Master's thesis, Beijing University of Posts and Telecommunications, 2015.

[4] "Proguard,," http://proguard.sourceforge.net/.

[5] J. Shu, J. Li, Y. Zhang, and D. Gu, "Android app protection via interpretation obfuscation," in *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*. IEEE, 2014, pp. 63–68.

[6] Y. Yang, W. Fan, W. Huang, G. Xu, and Y. Yang, "The research of multi-point function opaque predicates obfuscation algorithm," *Appl. Math*, vol. 8, no. 6, pp. 3063–3070, 2014.

[7] Z. Yuan, Q. Wen, and M. Mao, "Constructing opaque predicates for java programs," in *2006 International Conference on Computational Intelligence and Security*, 2006.

[8] Y. Yang, W. Fan, W. Huang, G. Xu, and Y. Yang, "The research of multi-point function opaque predicates obfuscation algorithm," *Appl. Math*, vol. 8, no. 6, pp. 3063–3070, 2014.

[9] "Codeverify.cpp,," http://androidxref.com/4.2.2_r1/xref/dalvik/vm/analysis/CodeVerify.cpp.

[10] "Dexverify.cpp,," http://androidxref.com/4.2.2_r1/xref/dalvik/vm/analysis/DexVerify.cpp.

[11] "Jeb." http://securitymusings.com/article/4003/android-security-and-the-tools-i-use-jeb.

[12] "dexdump." https://play.google.com/store/apps/details?id=com.redlee90.dexdump.

[13] "Idapro,," https://www.hex-rays.com/products/ida.

[14] "Dex2jar." https://sourceforge.net/p/dex2jar/wiki/UserGuide/.

[15] A. Bartel, J. Klein, Y. L. Traon, and M. Monperrus, "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot," in *ACM Sigplan International Workshop on State of the Art in Java Program Analysis*, 2012, pp. 27–38.

[16] W. Enck, D. Octeau, P. Mcdaniel, and S. Chaudhuri, "A study of android application security," *British Medical Journal*, vol. 2, no. 3859, pp. 1175–1175, 2011.

[17] "Android runtime," https://en.wikipedia.org/wiki/Android_Runtime.

[18] M. Dalla Preda and R. Giacobazzi, "Semantic-based code obfuscation by abstract interpretation," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2005, pp. 1325–1336.

[19] "Java native interface-jni,," http://developer.android.com/intl/zh-cn/training/articles/perf-jni.html.

[20] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[21] A. Venkatesan, "Code obfuscation and virus detection," Ph.D. dissertation, San Jose State University, 2008.

[22] S. Schrittwieser and S. Katzenbeisser, "Code obfuscation against static and dynamic reverse engineering," in *International Workshop on Information Hiding*. Springer, 2011, pp. 270–284.

[23] P. Schulz, "Code protection in android," *Insititute of Computer Science, Rheinische Friedrich-Wilhelms-Universitgt Bonn, Germany*, vol. 110, 2012.

[24] S. Ghosh, S. Tandan, and K. Lahre, "Shielding android application against reverse engineering," in *International Journal of Engineering Research and Technology*, vol. 2, no. 6 (June-2013). ESRSA Publications, 2013.

[25] D. Kundu, "Jshield: A java anti-reversing tool," Ph.D. dissertation, San José State University, 2011.

[26] M. Shoaib, N. Yasin, and A. G. Abbassi, "Smart card based protection for dalvik bytecode–dynamically loadable component of an android apk," *International Journal of Computer Theory and Engineering*, vol. 8, no. 2, p. 156, 2016.

[27] V. Balachandran, Sufatrio, D. J. J. Tan, and V. L. L. Thing, "Control flow obfuscation for android applications," *Computers & Security*, vol. 61, pp. 72–93, 2016.