2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering

DroidPro: An AOTC-based Bytecode-Hiding Scheme for Packing the Android Applications

1st Judong Bao
Peking University
Beijing, China
baojudong@pku.edu.cn

2nd Yongqiang He
Ardsec Company
Beijing, China
heyongqiang@ardsec.com

3rdWeiping Wen*
Peking University
Beijing, China
weipingwen@ss.pku.edu.cn

***Abstract*—Android is an open source mobile operating system represented by the Open Handset Alliance (OHA), developed by Google and other organizations since 2007, which has taken up most of the market share of smart devices. However, the applications on the platform are facing the increasingly serious security threat. Although the Android system itself provides a set of security mechanism to protect the safety of the system and applications, there are still many security risks. In order to hide the vulnerability of the applications and prevent the malicious users from tampering the apps, multiple anti-analysis methods have been applied by many Android packers to consolidate the apps. Bytecode-hiding is one of the most effective anti-analysis method, which can extract some bytecode from the Dex files and hide them from the vision of malicious analysts. Mostly, the hidden bytecode was encrypted, which can be recovered in runtime. But the conventional bytecode-hiding methods are always low-efficient and unsafe on some occasions, where the hidden bytecode can be recovered by the malicious analysts in some way. In this paper, we propose a bytecode-hiding scheme based on Ahead-Of-Time (AOT) compilation, called DroidPro, which can compile some chosen bytecode of Dex files of apps to native code in ahead-of-time that will be much harder to reverse. In our experiments, the apps packed by the packer associated with our bytecode-hiding scheme are more efficient and safer than other packers that use other bytecode-hiding schemes.**

***Index Terms*—Ahead-of-time compiler, reverse engineering, packer, Dalvik bytecode, code auditing, code profiling.**

I. Introduction

Android has occupied about 85.9% market share in the market of smartphone operating system in 2017 [3], [9]. Low barriers for entry of application developers increase the security risk for end users [7]. To protect apps from being tampered and reverse engineered, a number of app packing services (or packers) emerge [8], which prevent others from obtaining real code by concealing and obfuscating the real code (i.e., Dex bytecode) [25]. Currently, these packing services have successfully helped countless app developers protect their apps from maliciously reverse engineering and tampering. However, the efforts to crack the packed apps never stop. The unpacking tools to thwart the performance of various app packers also never stop evolving. Some novel unpacking approaches have been proposed, which can efficiently facilitate the analysis towards apps for reverse engineering or recovering the original Dex bytecode from packed apps [25], [26], [27]. Both the packers and the unpackers are evolving rapidly. Therefore, the arms race between packers and unpackers has become fiercer and fiercer and never ends.

Current packers usually involve a variety of defense measures to block analysis for reverse-engineering. Generally, the anti-analysis measures those packers employ can be classified into three categories. The first category of anti-analysis defense measures involve functions that check the static and dynamic integrity of the app (i.e., whether the app is patched or injected with debugging routines). These measures can be easily circumvented if analysts know the tricks beforehand. The second category of anti-analysis measures involve source code level obfuscation, which requires the source code to employ the protection. The third category, which is most complex and most reliable currently, involves bytecode hiding. These measures try to conceal the information of bytecode in the Dex files [6].

Considering the difficulty of using Dex code to crack down on Android, hiding the information of bytecode of Dex files has become the main measure of most app packers to hindering reverse engineering. Metadata modification and Dex encryption are currently the main measures to implement bytecode hiding for Dex files. Generally, as a defense measure modifying metadata is really tricky, which does not actually conceal the bytecode information. However, these defenses won't be available anymore, due to the stricter and stricter verification for Dex format. In fact, Dex encryption has just been the mainly effective measure.

Packers prefer to employ Dex encryption techniques, which need release the original bytecode in runtime. Packers perform a full-code releasing or incremental code releasing for recovering original bytecode. Similar to classic code packers on commodity desktop platforms, Dex encryption scheme generally depends on a decrypting stub that is responsible for decryption work at runtime. Packers always place the decrypting stub in native code part of a protected app as an initializer. The encrypted bytecode is first recovered by the decrypting stub, and then the Android VM will load and execute the decrypted bytecode.

Generally, the current bytecode hiding schemes are mostly based on encrypting bytecode of Dex files and decrypting it to original states at runtime. In these schemes the encrypted bytecode of the Dex files will always be recovered to the original state at some moment, which is easily applied by

2324-9013/18/31.00 ©2018 IEEE
DOI 10.1109/TrustCom/BigDataSE.2018.00093

the malicious users to reverse-engineer the applications. In fact, as the limited difficulty of cracking down Dex bytecode on Android, more and more developers are turning to C/C++ code for core code writing. Moreover, several ahead-of-time compilers have been proposed to make the improvement of the performance for the execution of Android devices or Java language on embedded platforms [14], [18], [19], [21]. In view of this, we got the idea that we can design an ahead-of-time compiler (AOTC) to compile the bytecode selectively extracted from Dex files to implement an AOTC based bytecode-hiding scheme.

In this paper, an Ahead-Of-Time compiler (AOTC) based bytecode-hiding scheme called **DroidPro** is proposed, which can compile directly the Dex bytecode to native code that can run smoothly on devices without recovering the original Dex bytecode that need run through the interpretation and Just-in-Time Compilation (JITC) of the Android virtual machine before. In DroidPro, we translate the bytecode of some filtered methods used by apps to intermediate representation (IR) of LLVM [12], which then can be directly translated to native code (i.e., so library). The generated native code (i.e., so library) and new Dex files (i.e., remaining Dex file after being extracted) are repackaged and packed to a new Apk file. Certainly the generated native code and new Dex files can be specially packed further. Compared with current main bytecode-hiding scheme (i.e., Dex encryption), our DroidPro can make the native code run directly in devices without recovering. Meanwhile the Android system can eliminate the overhead for recovering the original Dex, and also can make part of Dex bytecode disappear all the runtime, which means that the malicious users cannot get the original bytecode through traditional unpacking approaches against Dex file.

The rest of this paper is organized as follows. In Section II, the overview for the DroidPro framework is presented. The method filter mechanism is given in Section III. The detail of Dex-to-IR translation and what additional post-processing measures are involved in the DroidPro framework are described in Section IV. Experimental processes and results are given in Section V. In Section VI, the related work is given. At last the paper is concluded in Section VII.

II. Overview

A. DroidPro Architecture

The DroidPro framework consists of three main components, a method filter module, a Ahead-of-Time compiler(AOTC), and a bridge module which is native library. The method filter module is applied to filter the methods that are suitable for extraction to be hidden or be kept original state. The AOTC module is used to translate qualified methods for hiding into LLVM IR, compile the IR to native code, and link the generated native code with the bridge module, which is a library implemented with Java Native Interface (JNI) and is used to guarantee the contact between native side and the remaining bytecode in Dex.

B. The Execution Flow of DroidPro

The execution flow of DroidPro is illustrated in Fig. 1. At the beginning, the method filter module is designed to help extracting bytecode of methods within Dex files. Methods within an application will be divided into two categories and respectively put into two lists(hiding and remaining list). For methods in the hiding list, the bytecodes of them are fed into the AOTC module as input. After the compilation is finished, DroidPro will modify the bytecode of the original Dex files by evacuating the original methods' body and appending a native modifier to the methods' headers, so that any callers that invoke these methods can call the generated native code generated. At last procedure, the modified Dex files and the generated native code are built into a new application package (i.e., apk). Moreover, the DroidPro can coexist with most other pack technologies, which means that the newly generated application can be packed continuously with other software protection technologies including other bytecode-hiding themes. The applications packed by multiple defence measures certainly will be much safer.

III. The Method Filter Module

Because if the Dex bytecode was compiled to native code entirely, the generated new Apk package will extremely expanded in size, DroidPro does not try to compile the entire Dex bytecode. To balance the performances of security, execution and space, DroidPro instead just compile partial bytecode of Dex file. By exploiting the capability of Java Native Interface (JNI) and Java reflection techniques to support the mutual invocation between the Dex side and native side within one application, the newly application generated by DroidPro can not only keep the functions of the original, but also can possess better safety performance and moderate space performance. Therefore, the objective of the method filter module within DroidPro framework is to classify methods of an app into hiding list or remaining list, which contains the hidden or the left methods respectively.

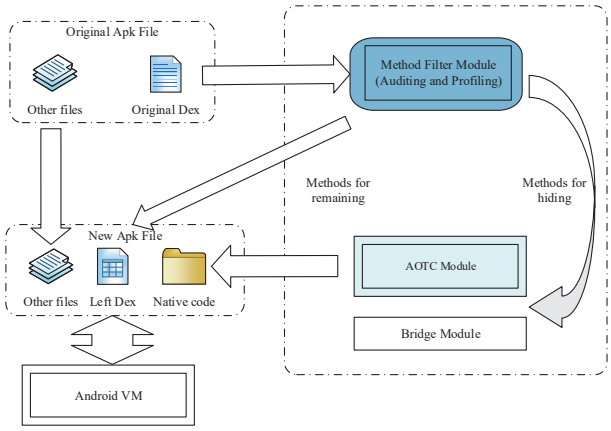

Fig. 1. The DroidPro Architecture.

Since the system or external dependency libraries don't need be packed, the opportunity of compilation resides in the bytecode that represents user defined methods. To filter proper methods, a method filter mechanism must be applied to provide the above two lists. But because of the difference of the application scenarios in practice, the principles and strategies of filtering methods can be various. Meanwhile the current defense techniques of packers always have some defects that make against the comprehensive performance of packed applications. Even our DroidPro accompanies a size issue that the size of the native code generated from Dex bytecode will expand several dozen times than original bytecode. Considering the uncertainty of method significance in practice, we are going to reference current technologies about the security detection of applications to help filtering appropriate methods. Meanwhile, we also take the execution performance and robustness of the packed applications into consideration in the method filter stage of packing.

A variety of code detection technologies have been proposed, including code auditing like [24] and malware detection [22]. Our case is that we want to hide the methods, which may cause vulnerability of the applications. So we used code auditing tools as the main part of the filter module of DroidPro, and also referenced the method filter mechanism of several existent AOTCs [20], [13], [15] for Android applications. At last we designed our current method filtering mechanism based on the code auditing and profiling of applications, which not only cares the security performance, also cares other performances including execution, space,etc.

A. The Auditing and Profiling Tools

In DroidPro, we employ auditing tool and profiling tool, which are applied to collect the information about the vulnerability and the execution performance of an application respectively, to help generating the profile data of Dex files. By the generated profile data we create the hiding list and remaining list.

The auditing tool is used to collect the information about code defects in Dex especially some weaknesses about data leakage. Our main intention is to get the positions of code defects in Dex, and by concealing the key points of code in control and data flow, the difficulty for reverse engineering to the applications packed by DroidPro will increase greatly. To counter software vulnerabilities, the patching in source code level is the best way, but that is not the duty of packing service. However, with the help of auditing tools, we can hide the methods associated with vulnerabilities and conceal the logic of vulnerabilities towards malicious users to protect the applications better.The profiling tool is used to collect the runtime execution information of methods in an application, including the execution time of each method, the call graph, the frequency of invocation to every method, and the execution time percentage of child methods, which will be recursively accumulated and synthetically considered to calculate the time-consuming values of methods. By comparing all the values about execution information of methods with the predefined thresholds, we can get a list of the methods with high execution overhead.

By filtering the methods associated with vulnerability and the methods with high execution overhead to hiding list and compiling them to native code, we can make the packed apps with superior security and execution performance to the original state. Currently, we employ AppAudit [23] and Traceview [5] as the auditing tool and the profiling tool respectively.

B. Invocation Overhead Avoidance

Through the preliminary processes of auditing tool and profiling tool, we can get the basic profiled data, which can generate roughly hiding list and remaining list. That means some methods may be compiled to native code or hidden to native mode. Simultaneously, the number of JNI invocations of every method can also be determined. If the number of JNI invocation is too high in one method, the compilation for that method will greatly increase the execution overhead of that method. So if a method with very large number of JNI invocations has been put into hiding list for its high execution overhead, it should be taken out from hiding list and put into the remaining list. The threshold is currently set based on the feedback from a set of performance experiments. This filter mechanism can take effect and helps minimizing the number of methods with frequent JNI invocation in hiding list. However, if the methods that are put into hiding list for security reasons, they should be remained there, because protecting application is the main purpose of DroidPro.

C. Method Filter Mechanism

The main idea of the current method filter mechanism is to first filter the methods associated with some vulnerabilities into the hiding list for hiding the bytecode through compilation, and then filter some methods for trying best to improve the execution performance. The main processes of the filter module is to collect the user-defined methods correlated with some vulnerabilities and with high time consuming values in the entire execution-flow and to put corresponding methods to the hiding list. The checking flow is organized into the following steps:

- Step 1: Through the vulnerability information collected by the auditing tool, we directly select the methods in Dex files which may take effect in the process of exploiting the vulnerabilities to the hiding list, especially some vulnerabilities about data transferring, database operation and some check logic in code. Go to step 2.
- Step 2: Calculate the result of the self-code execution rate multiply the invocation frequency of every method in Dex file according to the profiled data from the profiling tool to get the time-consuming value of that method. Check the time-consuming values of all user-defined methods and get the time-consuming values of every top-level method by accumulating the time-consuming values of their child methods recursively. If the time-consuming

values of a specific method are over predefined values, put the method into the candidate hiding list. Go to step 3.
- Step 3: Adjust the hiding list and the remaining list to avoid compiling the methods that may cause extra vulnerabilities or extra overhead due to their compilation. After that, we get the final hiding list and remaining list.

Through this method filter mechanism, a method which may cause vulnerability and spend a lot of time on its self code execution will be put into the hiding list.

D. Classifying Methods

Based on the description above, we have built a pretty filter module to classify the methods that are suitable for hiding list or remaining list respectively. Fig. 2 shows how the filter module works. First, based on security auditing result, we filter some methods into the hiding list. Then, to hide as much bytecode of Dex files to native code as possible and improve performance as much as possible, we analyze the rest of methods in Dex files with the profiling tools and select some other methods to the hiding list considering those methods can bring beneficial effect for overall performance improvement. Based on the two lists generated from the filter module, we can continue to the latter compilation and some post-processing works. In the future, we will involve multiple filter process, which means we will do the filter work several times with different auditing tools and profiling tools. By this we can avoid the emergence of new security issues and performance issues that may appear after one round of filter and compilation work.

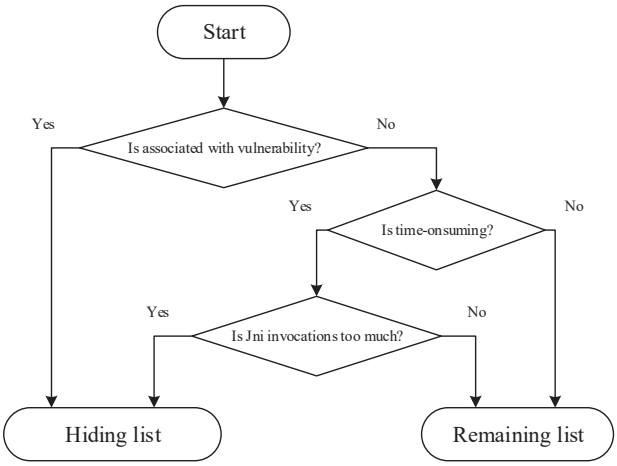

Fig. 2. The mechanism to classify methods

IV. Compilation And Optimization

In this section, we will detail how to extract methods from Dex files and translate the extracted bytecode to native style. In addition, some optimizations and post-processings we conduct in DroidPro are also given.

A. Dex-to-IR Convertion

The main work of AOTC module is to parse Dex files to intermediate represent (IR) of Low-Level Virtual Machine (LLVM) [12], and then translate the IR to native code. This work is based on the ANTLR [16] that can generate the parser to translate the preprocessed Dex bytecode to the LLVM IR, Clang [1] and LLVM that can cooperate to translate the IR and C code to native code. The translation processes are shown in Fig. 3.

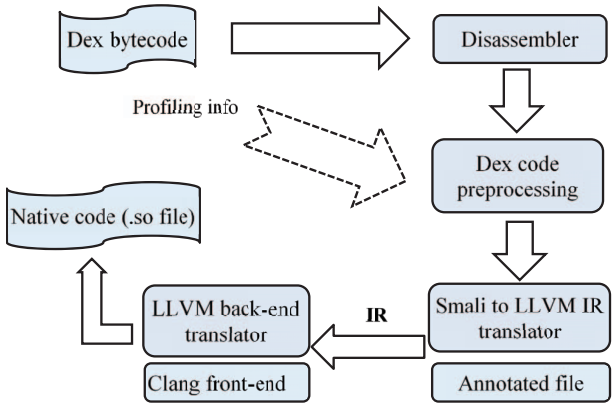

Fig. 3. The Dex-to-IR compilation flow of DroidPro.

1) Dex Code Preprocessing: Through the preprocessing, the Dex files are disassembled into a human-readable text file (i.e., smali file) by backsmali [11] and Dexdump. We applied both tools to collect all the necessary information for the conversion of Dex bytecode. The JNI based bridge module, which is library responsible for bridging Android VM (Dalvik or ART) and native code, and can be generated according to above information. The object of this phase is to better preprocess the Dex file to parse it into the LLVM IR. There are another two works included in this step. The first is to build function call cycles of the methods to compile together by modifying the return instructions of these methods. This can reduce the overhead of procedure calls among native methods that transfer parameters and return results by special stacks conventionally. The second is to generate auxiliary information for the next-step compilation by inserting annotations.

By chaining the native methods we can make the some parameter passing and result returning of native methods carry on in pure native mode. We can manipulate the invocation and return instructions of the caller and callee methods. By analyzing the invocation relation of the methods in the hiding list, we get all the call pairs of parent methods and child methods in hiding list. In the native code of parent method, we can modify the indirect invocation instructions for child methods to direct invocation instructions to the native code of child methods, which means that the invocation doesn't need traverse the VM mode. This measure can help avoid frequent switching back and forth between the native mode and Android VM mode. More details will be provided in the later section.

Gathering and recording information about every instruction is beneficial to the latter compilation work, especially useful to the semantic analysis of the bytecode. We can create annotations for fields, methods, and instructions of the bytecode in text files. With the help of the annotations we can then generate the IR from preprocessed bytecode and create the bridge module of the DroidPro framework. Using dexdump we can get all main information of class, field, method and baksmali can give the detail of every instruction. Table I gives some examples of annotations we use. In Table I, the FunctionType means a method's parameter types and return type to help constructing a method's parameters and return value from virtual registers of Dex instructions to local variables of LLVM IR. For example, FunctionType(DD)Ljava/lang/Double; indicates that the a function which is going to be compiled has two parameters of double type and returns a Ljava/lang/Double;type value. The item "FieldType" informs the type of one field, the type is "Ljava/lang/String;" in this example. The Lineitem records one bytecode's line number. The Instruction represents that it is to invoke a static function which have two int type parameters and return a int type result.

TABLE I. Examples of annotation

Item	Value
FunctionType	(DD)Ljava/lang/Double;
FieldType	Ljava/lang/String;
Line	.line 543
Instruction	invoke-static {v2,v3}, Lcom/examples/Test;->mul(II)I;

2) Handling Additional Issues: For normal execution instructions we take the way of translating the bytecode of Dex executables to LLVM IR, by which we can get much benefit from LLVM's mature compiler infrastructure. However, LLVM IR cannot directly invoke Java methods and fields, but they can invoke C code smoothly, so the logic in native level in DroidPro to process mutual invocations between native code and Android VM is realized in C code by using JNI. Based on above, there exist several issues in the translation process between native mode and Android VM mode.

One issue is that the generated native code must keep the language features of Java. Java, as an object-oriented language, has many object-oriented features, such as instance creation, virtual method, function overloading, etc. When converting to C or IR, these features should be kept. Fortunately, most these features are implemented originally by JNI in Java system. So that is easy to implement them with JNI by ourselves, we can implement these feature in translated IR and C code. For the function loading mechanism in Java, different from Java, in C language, functions can be distinguish by name or scope. So for function overloading, we can implement that by identifying different overloading functions with different names. For example, in Java class "com.test.DemoClazz", we have overloading functions, "void func(int)" and "void func(int, int)", in C level we can implement them by giving them different names.

```
sput-object v3, Lcom/test/DemoClazz;->value:Ljava/lang/Integer;
……
add-int/lit8 v3, v3, 0x1
……
sget-object v3, Ljava/lang/System;->out:Ljava/io/PrintStream;
```

Fig. 4. Virtual registers of Dex bytecode

```
%basicType = type { i32 }
%objectType = type { i32 }
……
%v5 = alloca %basicType, align 4
……
%v5_1 = bitcast %type* %v5 to i32*
store i32 666, i32* %v5_1, align 4
……
%v5_2 = bitcast %i32* %v5_1 to objectType*
store objectType %18, objectType* %v5_2, align 4
```

Fig. 5. Example code of variable renaming in IR code

The other one issue is that the bytecode instructions of Dex are based on virtual register, which is basically non-typed like the general purpose register in assembly language, it can store data of any type. From Fig. 4, In different parts of one piece of Dex bytecode, the register v3 can store different data types, it is first used to store a java.lang.Integer object, then holds a integer value, and last it is loaded a java.io.PrintStream object value. Similar to the bytecode of Dex, the LLVM IR conduct its local variables as virtual register, but the variables of LLVM IR are strongly-typed. So we must carefully handle the type conversion of the variable that is mapped to a virtual register of Dex bytecode, when that virtual register is referenced as another type.

LLVM IR is strong typed, which is very different from the Dex byte code. We must carefully handle the type change of variables representing a virtual register of Dex bytecode in its every use. With the help of type information annotations from the preprocessing of Dex code, we can use new temporary variables with new types to help operating the memory space that is allocated to represent the virtual register. In Fig. 5, the piece generated code of IR shows that how the memory space representing the virtual register v5 hold different types of data, just like it in Dex bytecode. Obviously, in every the virtual register accessing with different types in Dex bytecode, the memory space representing the virtual register is labeled by different name in LLVM IR.

3) Code Generation: After the preprocessing shown in Fig. 3, the Dex files are then translated to the human-readable text form intermediate representation (IR) (i.e., .ll files) of the LLVM compiler infrastructure, and the text form intermediate representation is finally translated into a shared library with the clang [1] front-end and LLVM back-end. Moreover, we have developed some special interfaces to handle the accesses

from the native side to the Android VM's resources, and packaged them into the bridge module as a library, which will be linked with the generated native code finally. This library will guarantee the basic operations of apps such as method invocation, instance creation, field accessing and so on.

B. Optimization

Current packers are all the composite style with a bunch of measures for anti-analysis defenses that all will bring additional overhead for the execution of the applications. So the overhead issue must be cared about discreetly and the optimization must be done. With JNI, the native mode and the Android VM can interact with each other. However, JNI suffers from time and space overhead just like other mechanisms of supporting interoperability. Generally, we can classify the reasons of JNI overhead to two categories, one is call-out operation and the other one is call-back operation. The call-out operations are adopted by Android VM to invoke the native methods. Contrarily, the native side can adopt the call-back operations to access the resources of Android system.

Compared to the call-out operations, the call-back operations involve more significant overhead, because they always need perform indirect-jump by referencing the JNI environment variable, which can cause extra overhead that call-out operations doesn't need. Besides, a call-back operation need take a large amount of time to perform constant pooling in advance for the need of context-switches. Table II shows the comparison of the average execution time between a call-back operation and a call-out operation got by performing each operation 1000 times.

As we can see, the gap between call-out and call-back operations in performance is so wide that some optimizations to call-back operations are strongly demanded. Since in packing work there is no need to modify Android's framework, we focus on reducing the call-back overhead of generated native code to improve the performance of the new applications.

TABLE II. The time comparison between 1000 call-out and call-back operations

	Native	Java
Non-static field	126.2 ms	11.2 ms
Static field	107.6 ms	5.2 ms
Non-static method	41.8 ms	2.5 ms
Static method	123.4 ms	32.1 ms

1) Resolution in Ahead of Time: Traditionally, the resources access from the native side such as field accessing and method invocation are realized with Java reflection mechanism, which are very slow usually. To avoid it, some types of instructions such as field accessing and method invocation are organized with a reference in symbolic forms. The reference is a item corresponding to offset/index of the symbolic forms. This conversion process is usually called constant pool resolution. As all the works in the execution flow of DroidPro are conducted ahead of time, we generate the constant pool resolution form ahead of time, which can eliminate the runtime overhead of constant pool resolution like Android system. For doing that work, dexopt [4] are introduced. Normally, when an app is installed, dexopt is used to generate the optimized Dex file. The Dex bytecode after optimization by this tool is called ODEX. Some constant pool referencing instructions in ODEX are replaced by a quicker reference, which reference the constant through offsets generated based on static linking. With the information got by parsing ODEX files, we can significantly speed up call-back operations of the generated native code. However, this does not contain static references because the offsets of static references are corresponding to the address from where the class is loaded.

2) Caching: For the static field accessing and static method invocation, there are no convenient measures like ODEX, like the ahead of time resolution from ODEX, which is only appropriate to the object instance. Therefore, we introduce a caching mechanism for improving the performance of static call-backs. To accelerate the speed of static field accessing and method invocation and reduce corresponding referencing overhead, the references of the fields and methods are cached at the native side. A hash table is employed to record the methods/fields based on their names. The field accessing and method invocation can be speed up by directly looking up the hash table. However, this mechanism also may come with some time and space overhead, it still can bring more than five times performance improvement over the original call-back operation on average.

V. Experiment

To illustrate and evaluate the comprehensive performance of our DroidPro framework on app protection, 10 unpacked app samples are specially made to test the security performance (i.e., compilation functionality), 1000 random unpacked apps are download to test the reliability and 10 of them are used to test space performance, and two benchmark applications, CaffeineMark 3.0 [17] and the BenchmarkPi [10] are used to test execution performance. The reason to specially make app samples is that we can design the composition of methods of Dex files, then in function test we can designate specific methods to be compiled by just tweaking the method filter module of DroidPro. Our experimental environment is based on Redmi Note 2 device running on Android 5.0.2.

A. Function Test

Hiding the extracted bytecode of the Dex file is the basic function of our tool, DroidPro. We designate some methods in advance, then extract and compile them to native code. Finally, we run the new apps in device to test whether the new app can run normally. We can decompile respectively the original and DroidPro-packed packages of apps to see some result. In Fig. 6, we can see one example that the method body of function add in Fig. 6 (a) has disappeared in Fig. 6 (b). The original apps are packed to new apps whose hidden methods are attached the reserved word native.

Any user-defined method we designated in advance in 10 app samples can be hidden (or compiled) in test and all the new

```
.method public static add(Ljava/lang/Double;Ljava/lang/Double;)Ljava/lang/Double;
    .locals 4
    .param p0, "v1"    # Ljava/lang/Double;
    .param p1, "v2"    # Ljava/lang/Double;

    .prologue
    .line 542
    new-instance v0, Ljava/math/BigDecimal;

    invoke-virtual {p0}, Ljava/lang/Double;->toString()Ljava/lang/String;

    move-result-object v2

    invoke-direct {v0, v2}, Ljava/math/BigDecimal;-><init>(Ljava/lang/String;)V
```

(a) The original method body of function add

```

.method public static native add(Ljava/lang/Double;Ljava/lang/Double;)Ljava/lang/Double;
.end method

```

(b) The method body of function add after packing

Fig. 6. The change of method body

apps can run normally. In test, the hidden (or compiled) bytecode cannot be recovered by traditional unpacking measures. We can also designate the methods to be hidden in practice by tweaking the method filter mechanism, and certainly the new apps packed by DroidPro can be continuously packed by other ant-analysis measures for anti-reverse.

1000 app samples are tested by Monkey [2] in respectively original state and packed states. Table III show that there are 973 app samples packed by DroidPro can still run normally, and 27 samples meet a crash in 1000 unpacked app samples. In addition, 1000 packed app samples are not unpacked by known unpacking tools, such as AppSpear or DexHunter etc. DroidPro can effectively sabotage the process of them. Anyway the robustness and the safety performance of DroidPro is pretty excellent.

TABLE III. The result of robustness test

	Normal	Crash
Packed	973	27
Original	1000	0

B. Performance Test

To exactly evaluate the execution performance of DroidPro, two benchmark programs, CaffeineMark 3.0 [17] and the BenchmarkPi [10], are employed as benchmarks. The two applications can be directly downloaded from their own web pages.

1) Testing of CaffeineMark 3.0: CaffeineMark 3.0 contains a series of test items, and every its score represents the count of Java instructions executed per second in that item. Table IV shows the description of the test items involved by CaffeineMark 3.0. Fig. 7 gives the test results about performances of CaffeineMark 3.0. As shown in Fig. 7, the scores by using DroidPro are much better than the original executions, in most test items except for the String test. Overall, the application packed by DroidPro has the best performance. The anomaly result in the String test is mainly related to the frequent JNI invocations.

TABLE IV. Items of CaffeineMark 3.0

Item	Description
Sieve	The classic sieve of Eratosthenes finds prime numbers.
Loop	The loop test uses sorting and sequence generation as to measure compiler optimization of loops.
Logic	Tests the speed with which the virtual machine executes decision-making instructions.
Method	The Method test executes recursive function calls to see how well the VM handles method calls.
Float	Simulates a 3D rotation of objects around a point
String	Operation of basic string

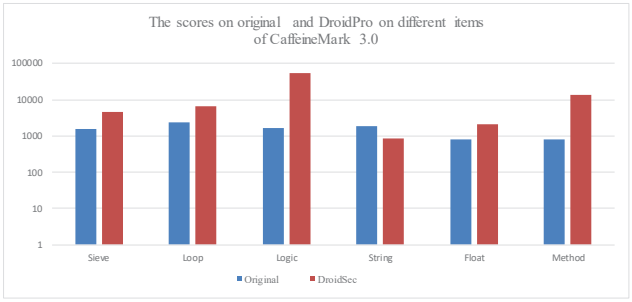

Fig. 7. CaffeineMark 3.0 scores comparison

2) Testing of BenchmarkPi: The BenchmarkPi is often used to test a device by calculating the value of Pi, and is a excellent tool to test the performance of a CPU. In this test, we use DroidPro to compile the methods that are responsible for the main workflow of BenchmarkPi. The performance results are shown in Fig. 8. We can see that the execution of BenchmarkPi packed by DroidPro is about two times faster than that is original.

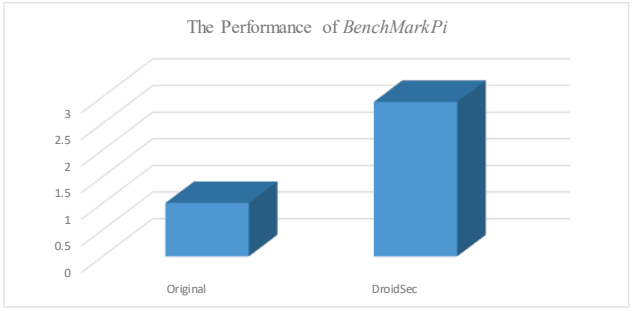

Fig. 8. Performance comparison of BenchmarkPi

3) Size Test: From 973 app samples that can run normally after packing we randomly choose 10 samples per 20MB interval, whose overall size scope is 0-200MB. The tests are conducted with the method filter module in Section 3. By the Table V and Fig. 9, we can see the size change of packages between the original and packed applications. The data is given in MB except last column that show that the weighted value of average increased size of every range. The weighted value of average increase size is about 0.38, which means the size

of apps packed by DroidPro is 0.38 larger than the original.

TABLE V. The result of size test

	Average Size		Average Increased Value	
Range	Original	Packed	Value	Weighted
0-20	15.62	21.47	5.85	0.37
20-40	19.65	27.10	7.45	0.38
40-60	52.67	72.66	19.99	0.38
60-80	72.28	98.99	26.71	0.37
80-100	93.53	129.06	35.53	0.39
100-120	111.24	152.36	41.12	0.37
120-140	122.88	171.48	48.60	0.41
140-160	161.20	219.00	57.80	0.36
160-180	185.50	249.91	64.41	0.35
180-200	169.60	241.27	71.67	0.42

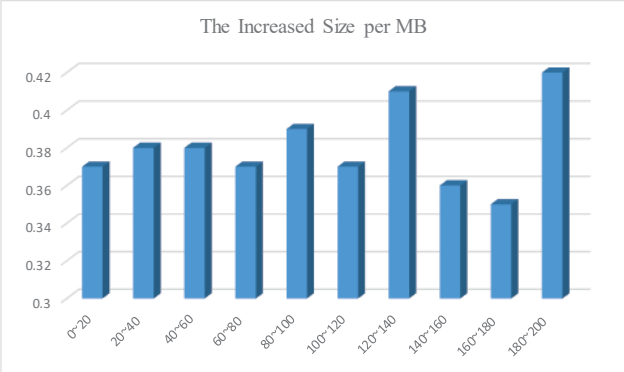

Fig. 9. The increased size per MB

Overall, the comparison described above indicate that the AOTC based bytecode-hiding tool, DroidPro, can effectively hide the extracted bytecode for Android applications. Though the experiment doesnt integrate other anti-analysis measures to pack the app, it still has the basic function of AOTC based bytecode hiding that the generated native code will execute without translating back to Dex bytecode. The filtered bytecode of the original application has just been hidden in native code style. The experiments above have demonstrated that the scheme we proposed almost has improved the comprehensive performance of the applications rather than bring the overhead in order to recover the hidden Dex bytecode, there almost is no any other extra overhead except of the normal execution of the application, it has come over the shortcomings of other bytecode-hiding schemes mentioned earlier in this paper.

VI. Related Work

There have been some researches to apply an ahead-of-time compilation technology for Android applications to improve the execution performance of apps [13], [15], [20]. Similar to other AOT compiler for Android system, DroidPro also is method-based. Unlike other AOTC schemes, DroidPro transforms the Dex bytecode to intermediate representation of LLVM rather than the C code, the mature compiler infrastructure of LLVM give a very big help to the design and the development of DroidPro. Finally, our is functionally different from other AOTC schemes, our DroidPro is mainly de-signed to hide the selected Dex bytecode to prohibit from reverse-engineering the applications. However, the improvement of the execution performance is just side product, and that bring the potential for the packers associated with DroidPro to dig more performance to consolidate apps. The bytecode hiding schemes of current app packer services are mostly based on hiding bytecode of Dex file that will be recovered to original states at runtime. In those schemes the encrypted bytecode of the Dex files will always be recovered to the original state at some moment, which is easily applied by the malicious users to get the original bytecode of Dex files. That is the main difference between DroidPro and them, and also the advantage of DroidPro over them.

VII. Summary And Conclusions

Through the above experiments the paper draw the following conclusion. the AOTC based bytecode-hiding scheme, DroidPro, can extract and compile the bytecode of Dex files to native code, therefore can protect the apps more effectively than other packers. DroidPro has carefully minimized extra overhead of JNI with several optimization measures such as resolution ahead of time and caching method/field references. In addition, DroidPro has no the overhead of decrypting that is involved by other most packers.

Furthermore, as a unitary type measure for packing apps, DroidPro can be used by combining with any other anti-analysis measures of packing apps to improve the capability to resist reverse engineering like other packers that always are synthetical type with lots of anti-analysis measures. DroidPro currently introduces the existing auditing and profiling measures to determine which methods ought to be compiled (or hidden) by DroidPro or be retained. The mechanism of filtering appropriate methods can be changed according to various needs in practice, it is also our new research direction to develop an adaptive mechanism for filtering methods.

In summary, the main idea of DroidPro is to classify and compile methods associated with some vulnerabilities and high execution overhead of an Android application to native code, and make the generated native code be called by the Dalvik virtual machine or Android Runtime (i.e., ART) via JNI at run time. Generally, the apps packed by DroidPro have shown better performance than the apps packed by other packers in experiments.

References

[1] Clang: a C language family frontend for LLVM. http://clang.llvm.org/, 2018. [Online; accessed 15-June-2018].
[2] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey, 2018. [Online; accessed 15-June-2018].
[3] Android. Statistics & Facts. https://www.statista.com/topics/876/, 2018. [Online; accessed 15-March-2018].
[4] Google Android. Dexopt Dexfile Optimization Tool. https://source.android.com/devices/tech/dalvik/dex-format, 2018. [Online; accessed 15-June-2018].
[5] Google Android. Inspect trace logs with Traceview. https://developer.android.com/studio/profile/traceview, 2018. [Online; accessed 15-June-2018].

[6] Axelle Apvrille. Playing hide and seek with dalvik executables. *Hacktivity, Budapest, Hungary*, 2013.
[7] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.
[8] Gartner. Debunking Six Myths of App Wrapping. https://www.gartner.com/doc/3008117/debunking-myths-app-wrapping, 2018. [Online; accessed 15-June-2018].
[9] Google. Android Developer Website. https://developer.android.com/, 2018. [Online; accessed 15-June-2018].
[10] Benchmark International. BenchmarkPi. http://www.benchmarkpi.com/, 2018. [Online; accessed 15-June-2018].
[11] JesusFreke. Smali/baksmali is an assembler/disassembler for the dex format. https://bitbucket.org/JesusFreke/smali, 2018. [Online; accessed 15-June-2018].
[12] Chris Lattner. The LLVM Compiler Infrastructure. https://llvm.org/, 2018. [Online; accessed 15-June-2018].
[13] Yeong-Kyu Lim, Sharfudheen Parambil, Cheong-Ghil Kim, and See-Hyung Lee. A selective ahead-of-time compiler on android device. In *Information Science and Applications (ICISA), 2012 International Conference on*, pages 1–6. IEEE, 2012.
[14] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient java environment mixing bytecode and compiled code. In *COOTS*, pages 1–20, 1997.
[15] Hyeong-Seok Oh, Ji Hwan Yeo, and Soo-Mook Moon. Bytecode-to-c ahead-of-time compilation for android dalvik virtual machine. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1048–1053. EDA Consortium, 2015.
[16] Terence Parr. ANother Tool for Language Recognition. http://www.antlr.org/, 2018. [Online; accessed 15-June-2018].
[17] Pendragon. CaffeineMark 3.0. http://www.benchmarkhq.ru/cm30/, 2018. [Online; accessed 15-June-2018].
[18] Todd A Proebsting, Gregg M Townsend, Patrick G Bridges, John H Hartman, Tim Newsham, and Scott A Watterson. Toba: Java for applications-a way ahead of time (wat) compiler. In *COOTS*, pages 41–54, 1997.
[19] Ankush Varma and Shuvra S Bhattacharyya. Java-through-c compilation: An enabling technology for java in embedded systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 3, pages 161–166. IEEE, 2004.
[20] Chih-Sheng Wang, Guillermo Perez, Yeh-Ching Chung, Wei-Chung Hsu, Wei-Kuan Shih, and Hong-Rong Hsu. A method-based ahead-of-time compiler for android applications. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 15–24. ACM, 2011.
[21] Michael Weiss, François De Ferriere, Bertrand Delsart, Christian Fabre, Frederick Hirsch, E Andrew Johnson, Vania Joloboff, Fred Roy, Fridtjof Siebert, and Xavier Spengler. Turboj, a java bytecode-to-native compiler. In *Languages, Compilers, and Tools for Embedded Systems*, pages 119–130. Springer, 1998.
[22] WP Wen, Rui Mei, Ge Ning, and LL Wang. Malware detection technology analysis and applied research of android platform. *Journal on Communications*, 8:78–85, 2014.
[23] Mingyuan Xia. AppAudit: Uncover Hidden Data Leaks in Apps. http://appaudit.io/, 2018. [Online; accessed 15-June-2018].
[24] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 899–914. IEEE, 2015.
[25] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. Adaptive unpacking of android apps. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 358–369. IEEE, 2017.
[26] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 359–381. Springer, 2015.
[27] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: toward extracting hidden code from packed android applications. In *European Symposium on Research in Computer Security*, pages 293–311. Springer, 2015.