Protecting Android Apps from Repackaging Using Native Code*

Simon Tanner, Ilian Vogels, and Roger Wattenhofer

ETH Zurich, Switzerland
{simtanner,ivogels,wattenhofer}@ethz.ch

Abstract. Android app repacking allows malicious actors to modify apps, bundle them with malware or steal revenue. Current detection mechanisms of app distribution services are questionable in their effectiveness, and other proposed repackaging protection schemes do not have the necessary protection against circumvention. We propose a repackaging protection architecture that verifies the app's integrity at runtime. We make use of encrypted sections of bytecode that can be decrypted with a key derived at runtime. The method partially relies on native code, and as such is difficult to circumvent. We show that our implementation provides a practical integration in the workflow of an app developer.

Keywords: Repackaging protection · Android · App security

1 Introduction

Android is the most common mobile operating system. Due to its popularity, Android also attracts malware developers. Apps submitted to Google Play are screened before they are published[1] and Android's Google Play Protect regularly scans apps that are installed on devices, regardless from which source the apps originate.[2] Even though these security mechanisms are in place, their effectiveness is debatable as they do not detect every malicious app submitted to the service. Several malicious apps were published on Google Play in 2017 and 2018, impacting millions of users.[3,4] Additionally, many third party app stores have weaker security checks. In China, the majority of Android users do not have access to Google Play and have to use third-party app stores, many of which are not trustworthy and distribute modified versions of popular apps [9].

Attackers can unpack apps contained in APK files, modify their content and then repackage and distribute them. It is common that malware is hidden within

* The authors of this paper are alphabetically ordered.

[1] https://source.android.com/security/reports/Android_WhitePaper_Final_02092016.pdf

[2] https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf

[3] https://blog.checkpoint.com/2017/04/24/falaseguide-misleads-users-googleplay/

[4] https://blog.cloudflare.com/the-wirex-botnet/

copies of existing popular apps. Attackers may also repackage popular apps to divert ad revenue. Therefore, making repackaging of Android apps harder or even impossible is of interest to both users and developers of apps.

In this work we propose and implement[5] a mechanism to protect apps from being repackaged by transforming them after compilation, before they are distributed to the public. The app is transformed without any impact on the user experience, except for a small slowdown. However, the transformation prevents normal execution if the app has been tampered with. To make the protection difficult to circumvent, it relies on encrypted integrity checks in native code. This protection prevents an attacker from modifying the app, and potentially bundling malware with it.

2 Related Work

Several solutions have been proposed to prevent the distribution of repackaged Android applications. The methods can be divided into centralized and decentralized approaches.

Centralized Approaches The centralized approaches typically analyze features from the collection of apps hosted by a central distribution platform; they identify similar features between apps, such as the instruction sequence patterns [3,20], their call graphs [4], the trace of system calls [15], the layout of the Android Activities [12,13,18], or the presence of software watermarks [19]. In this scenario, the application distribution service monitors the apps submitted by developers and removes suspected repackaged apps. This requires removing the infringing apps in a timely manner, before the offending apps are distributed to a significant amount of users.

Decentralized Approaches With a decentralized approach, the repackaged app is detected at runtime on the user's device. This way, the app verifies its own integrity during runtime. This has the advantage that it distributes the repackaging detection workload. As soon as tampering has been detected, an appropriate response mechanism can be executed during which the app may abruptly stop executing and inform the user about the tampering.

Several works about software protections that assert the software's integrity at runtime have been proposed by the research community. For instance, [2] proposes a generic integrity checking and tamper prevention scheme which involves inserting multiple pieces of code, called *guards*, that protect a specified region of code. These pieces of code typically compute a checksum over the region of machine code instructions. Droidmarking [10] proposes a non-stealthy repackaging detection approach which sends watermarking information to a separate standalone app that is responsible for validating the integrity of protected apps.

[5] The source code is provided at https://github.com/ilian/repackaging-protection

However, Droidmarking also relies on the distribution platform to statically scan the apps for tampering.

The Stochastic Stealthy Network (SSN) [6] is another dynamic repackaging protection method. It validates the public key of the developer by comparing substrings of the public key obtained at runtime with obfuscated hard-coded substrings. The public key of the developer is obtained at runtime using Java reflection. The different function names that are called using reflection are obfuscated to make static analysis more difficult for an attacker trying to circumvent the applied protections. Zeng et al. [17] propose *BombDroid*, a repackaging protection scheme based on logic bombs and encrypted code blocks. A logic bomb is a piece of code that executes when specified conditions are met. The code responsible for detecting tampering is stored within an encrypted code block. The proposed approach is designed in such a way that the decryption key is unknown to the attacker without running the application.

Issues with SSN and BombDroid While the authors of SSN consider different dynamic attacks, a much simpler and more practical attack can be performed instead: as reflection is used to invoke a method to obtain the developer's public key, an attacker can rewrite the method invocations to invocations of a method that is injected by the attacker. The attacker can pass the instance and arguments of the original reflective method invocation to the newly injected method. This injected method can then return the original public key. This attack scenario can be performed without any dynamic analysis and can disable the repackaging detection for all apps protected with SSN. As most method calls in typical Java applications are not invoked using reflection, there is not a lot of runtime overhead when performing such an attack.

There are also some potential issues with BombDroid. We consider an attacker who can inject code that allows the modified app to perform code analysis and modification at runtime. Such a dynamic attacker can modify the decrypted instructions before they are executed. Similar to the previously discussed problem, the attacker can modify the instructions that obtain information from the environment, which is being relied on for the repackaging detection such as the public key of the developer and file descriptors to the original APK. An attacker can divert invocations of methods that provide environment information to injected methods that return simulated information to spoof the environment. When such an attack is performed, the repackaging detection mechanism is circumvented. Such attacks can be performed in practice by modifying and generating Dalvik bytecode at runtime.

3 Design Overview

We propose a resilient repackaging protection scheme that is based on encrypting code blocks. It partially relies on native code to address the attacks possible on previous work as discussed in Section 2. The system modifies the application

provided by the app developer and adds integrity checking in a robust manner. If repackaging is detected, an appropriate response mechanism is invoked.

The app to protect can be provided as an APK containing compiled bytecode and native code. After compiling the Android app to Dalvik bytecode and native code, the binary files are bundled together with resources and assets (images, audio, video, raw files, etc.) into an APK file. Our proposed repackaging protection scheme can be applied to this APK file to generate the protected app. The repackaging protection can thus be easily integrated into the workflow of app developers as an additional step in the compilation pipeline. As the protection is embedded within the application itself, we can distribute the protected APK to many users and distribution platforms regardless of their enforcement of repackaging protection. The transformation could also be performed by an app store to protect all apps it serves.

We aim to insert integrity checks that take action whenever repackaging has been detected. These code snippets are inserted in many locations in the app, such that the integrity is asserted at different locations throughout the execution. The challenge is to make the applied protection tamper-resistant such that it is difficult for an attacker to bypass or remove the added protection measures. Once the repackaging has been detected, response code can be executed. Many different responses are possible such as informing the user, informing the developer or crashing the app. Our repackaging protection scheme is not only robust against static analysis, but also against attackers that perform dynamic analysis.

Static Analysis The proposed system adds code to the apps to perform the repackaging detection and countermeasures. This added code has to be protected from static analysis to make removal by the attacker difficult. We make use of a technique called *conditional code obfuscation* [11] to prevent static analysis of code blocks. The condition `v == C` for a variable `v` and constant `C` is transformed to the semantically equivalent (assuming no hash collisions) `H(v) == H_c` where `H` is a one-way hash function and `H_c` is a constant equivalent to `H(C)`. We can exploit the fact that the value of the variable `v` is not known to an attacker, unless a brute force attack is performed. A key k is derived from the constant value `C` to encrypt all the bytecode within the true branch of such if-bodies during the transformation phase. The protection scheme is depicted in Fig. 1 in which these encrypted blocks in the bytecode are represented by the instructions surrounded by a red border.

Dynamic Analysis Repackaging detection and response code is not inserted in Dalvik bytecode because it is not difficult for an attacker to replay any environment information as if the original application is running instead of a repackaged one, as discussed in Section 2. Instead, we perform the checks in native code, which is also encrypted to prevent static analysis. Using native code makes dynamic analysis more difficult since it has to be performed on a machine-instruction level instead of the much simpler bytecode-instruction level. A native method is called from within an encrypted bytecode block using the Java Native

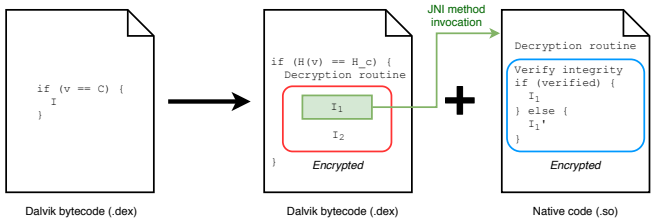

Fig. 1. Transformation of candidate code blocks

Interface (JNI). A key is passed to it to decrypt a native code block. At runtime, the native code is decrypted and the integrity of the application is verified. The use of both bytecode and native code encryption thus increases the difficulty of inspecting the behavior that is performed at runtime.

Code Removal An attacker could still remove or bypass the native code and the integrity checks contained in the native code would not get executed. To prevent this, we enforce that the native code has to be executed to preserve the functionality of the app. We therefore move a sequence of bytecode instructions, $I_1 \subseteq I$, to the native code as can be seen in Fig. 1. These instructions are rewritten to equivalent JNI function calls, which have the same effect as executing the bytecode instructions directly. The attacker cannot remove the native method invocation because the normal application behavior depends on it.

Detection of Repackaging To detect repackaging, we compare parts of files within the APK using a one-way hash function. These are typically the Dalvik Executable files, but our protection mechanism can also be configured to protect assets such as images and audio files. Integrity validation is performed in the native code when an encrypted block is encountered. Each encrypted block computes the hash of a different part of the files to reduce the impact on the performance when reading the files from storage. Alternatively, a digest of the developer's public key (located under the `META-INF` directory) can be validated to further increase performance. These computed hash values are compared to known hash values of the original app. Once the application has verified that it has not been repackaged, the instructions that have been moved to the native code (I_1) are executed. Otherwise, the response behavior is executed (I_1').

4 Implementation

The protection system transforms the bytecode of the app to an intermediate representation, without having access to the source code, and then performs the analysis to find suitable locations for bytecode encryption. The native method invocations are added such that the integrity checks are called at runtime. These

checks must depend on a reference state of all files. Therefore, the native code is generated after the Dalvik bytecode has been transformed.

Performing static analysis and modifications directly on Dalvik bytecode can be error-prone and cumbersome. In our implementation, we use the Jimple [14] intermediate representation for analysis and modifications. We use the Java API of the Soot framework[6] to perform modifications on the Jimple intermediate representation. After performing the necessary modifications for the repackaging protection in the Jimple code of the app, the Soot framework allows us to transform the Jimple code back to Dalvik bytecode and a new APK file can be generated.

Finding Candidate Blocks Our goal is to find a list of statements I, also referred to as a block, that is executed after a condition of the form

```
if (v == C) {
    I
}
```

has been evaluated to be true, where `v` is a variable and `C` is a constant. We iterate through all if-statements within each method body. A key to encrypt the code block can then be derived from `C`. In our implementation SHA-1 is used to transform the if condition. We must ensure that all statements within the if-body are only executed after the condition has been evaluated to true. Unconditional jump statements could jump to any statement in I without having satisfied the condition of the if-statement. In this case we could not compute the encryption key that is derived from the constant `C`. Therefore, if such a jump statement is encountered the code block does not get transformed and encrypted.

Transformation of a Block We derive the decryption key for a block from `v` when the true branch has been taken, and want to decrypt the bytecode corresponding to the statements within the if-body. Thus, we need a facility to decrypt and execute bytecode at runtime. Unfortunately, redefining classes containing the encrypted bytecode with ones that contain the decrypted bytecode is not possible with the Android Runtime (ART), as the instrumentation package `java.lang.instrument` that provides the `redefineClasses` method is not available on Android. We instead load a *new* class for each encountered encrypted block by making use of the `dalvik.system.InMemoryDexClassLoader` class (present since the release of Android 8.0) which can load classes in the form of Dalvik bytecode from memory. This new class has a method containing the extracted code block. The bytecode of this class gets encrypted in the transformation phase. This method is then invoked at runtime after loading the class containing the decrypted bytecode. We invoke this method containing the original statements to preserve the semantics of the app. Because we have extracted statements from within the if-body to a separate class, we hereafter

[6] https://github.com/Sable/soot

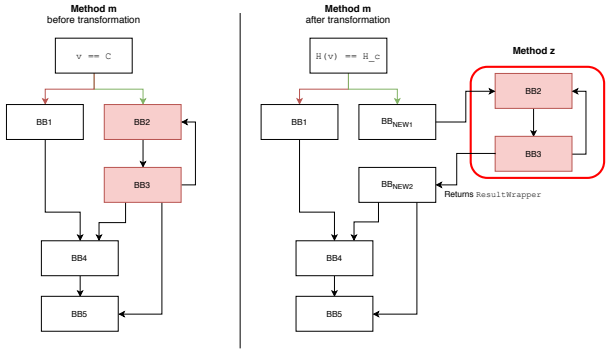

Fig. 2. Comparison of an example of a control flow graph before and after transformation. Basic blocks (BB) with a red background represent the basic blocks whose instructions can be extracted to a separate class and encrypted.

call the newly generated class, the *extracted class*. Since the statements are executed in the context of a different class and method than in the original if-body, workarounds need to be put in place in order to preserve semantic equivalence between the original and transformed app.

A method z is added to the new class with the sequence of statements I, local variables, and trap handlers of the original block. For every local variable used in a statement in I, a parameter is added to the method signature of z. We refer to the method that originally contains I as m. The original method m is transformed to remove the sequence of instructions I and add a call to z passing the corresponding local variables. The condition before executing I is replaced in m by the hashed comparison. The original constant C is therefore not present anymore. When the local variables get modified in method z, these changes are not visible outside of z. To preserve semantic equivalence between the original and transformed method m, z returns a list of the modified primitives and references to m, allowing these changes to be reflected in the context of m.

Jump targets in Dalvik bytecode, for both conditional and unconditional goto statements, must be contained within the same method as the jump statement. As we have copied the jump statements from the original method to the extracted class, the jump target is potentially not contained within the same method anymore. To resolve this, an identifier for the jump target is returned to the caller of z whenever a jump to a statement outside of z needs to be performed. The jump is subsequently performed in the transformed method m, after z has returned.

To pass all the discussed information from z back to m, a helper class `ResultWrapper` is used.

Fig. 2 shows an example of the transformation of the control flow graph. The nodes BB2 and BB3 in the figure are the basic blocks corresponding to the instructions I. In the transformed method, $\mathrm{BB}_{\mathrm{NEW1}}$ is responsible for decryption and calling method z. The inserted instructions responsible for restoring the

modified primitives and handling return values are omitted in this illustration. We can see that jumps from BB3 to BB2 persist after transformation, whereas jumps from BB3 to BB4 and BB5 are removed from BB3 after transformation, and instead return the corresponding jump identifier to BB_{NEW2} using the returned `ResultWrapper` instance.

Some class members that were accessible in the original method are not accessible anymore when called from the method z outside of the original class. Java access modifiers are not only enforced during compile time, but also during runtime. In the Dalvik Executable format, access modifier bit fields are stored for classes, inner-classes, fields and methods in the `access_flags` bit field.[7] As we are potentially accessing or modifying a field, or calling a method that the newly generated class does not have access to, we need to statically set the access modifier of the relevant classes and class members to public.

Adding Native Integrity Checks Integrity checking is performed in native code which is called via the Java Native Interface (JNI). More precisely, the extracted method z will call a native method to perform integrity checking. Our implementation generates C++ code to be compiled to a shared object file. For every native method added to the Jimple intermediate representation, we write its corresponding method signature to the generated C++ source file. To prevent an attacker from ignoring method calls to the shared library, we move a statement from the extracted block I to the native code by emulating the execution of the statement using the C++ JNI. The native program code is then encrypted with a randomly chosen symmetric key stored in the encrypted bytecode of z.

A statement w from the extracted block is chosen to be woven into the native code. A static native method signature is added to a new Java class that contains all the added native methods. Note that this Java class only contains a list of method names, return type and parameters it accepts without providing any direct implementation. This injected class is responsible for loading the compiled shared object file that contains the native code for all blocks in its class initialization method. The parameters of the native methods are set such that all local variables needed to execute statement w in the native code and a decryption key to decrypt the native code can be passed down from z to the native method. The statement w is then removed from the extracted block, and is instead substituted by a static invocation of the native method, passing the local variables referenced by w and a randomly chosen decryption key k used to decrypt the native code. This key gets decrypted at runtime together with the bytecode of the extracted class. The generated native code consists of code that asserts the integrity of the running app and the JNI method calls needed to emulate the effect of the woven code w.

The execution of the generated native methods consist of three parts:

1. Decryption of the native code using the supplied key, passed as an argument to the native method.

[7] https://source.android.com/devices/tech/dalvik/dex-format

2. Validating the integrity of the installed Android application.
3. Executing statement w using JNI to preserve semantic equivalence between the transformed and original application.

We write the code to be encrypted (all but the decryption procedure) to a dedicated function that will be decrypted at runtime. We can use a function pointer to obtain the memory location of the function that needs to be decrypted in the context of the decryption routine with the supplied key that is passed as an argument. Before decrypting the instructions that are located within the text segment, we must ensure that the memory pages where the decrypted instructions will be written to are marked as writable. Note that as the decryption of the instructions are performed in-place, the memory locations of the encrypted and decrypted instructions are the same. As specified by the `p_flags` field in the ELF header of the compiled shared object file, the text segment is by default marked as readable and executable. Thus, the memory pages that contain the encrypted instructions are first marked as writable using the `mprotect` system call. We mark the relevant memory pages as writable in preparation for decrypting the instructions in-place. Encryption and decryption is performed with a stream cipher, as we want to preserve the length of instructions in the ELF binary so we can statically encrypt during the transformation phase in-place, and decrypt during runtime in-place. We decided to use AES-128 in CTR mode, which turns the block cipher into a usable stream cipher for our purpose. Each byte to be decrypted is XORed with the byte generated by the stream cipher based on the chosen key.

After decrypting the instructions, the permissions of the memory pages can be restored to readable and executable for security reasons. A static boolean within each generated method is set to indicate that the code has already been decrypted such that the instructions are only decrypted once.

After the native code has been decrypted, the instruction pointer enters the instructions that were previously encrypted. We validate the integrity of the running application by obtaining the path to the APK file that contains the code of the currently running app. We can obtain this path by invoking the Android PackageManager binary, located at `/system/bin/pm` when supplying it with the package name of the currently running application. We compute hashes of parts of the files within the APK archive to verify its integrity.

Each generated native method computes and compares the hash of a section of a file with the precomputed hash that was determined statically during the transformation procedure. At this point in the transformation process, we are generating C/C++ code for the native library. This implies that we can statically compute the hash of every file that will be bundled with the APK, except for the shared library file itself and files added after compilation such as the developer's code signature. Note that we can also assert integrity of the developer's certificate file, which might be a good choice when all applications signed by this certificate are trusted by the developer. If the hashes match, the woven statement w is executed to ensure correctness of the transformed application. If the hashes do not match, we execute our response mechanism. In the current

implementation a null-pointer is dereferenced, such that a segmentation fault is raised. An alternative to this would be to also execute a JNI call, but instead of emulating the same effect as w, we could randomly invoke a Java method with random parameters or launch a new activity that is not supposed to be launched under normal circumstances in order to interrupt the normal behavior of the application.

As the symmetric key used to encrypt and decrypt the native code is known during the transformation phase, we statically encrypt the relevant code sections after compilation using the offsets of the functions to be encrypted. Non-static functions have external linkage, and are contained in the symbol table. The symbol table contains a mapping between the symbolic names and their offsets within the compiled binary. We encode and store the symmetric key in the name of the function before compilation (with C linkage to prevent function name mangling) such that both the offset to start encrypting at and key to encrypt are known when reading the symbol table of the compiled binary. After compilation, we can thus easily encrypt the relevant sections by reading the keys and offsets in the ELF header with the `nm` utility. After encrypting the instructions, we strip the keys from the symbol table with the `strip` utility so we do not leak the key used to encrypt the native code to the attacker.

5 Security Analysis

The goal of the system is to make it as difficult as possible for the attacker to circumvent the added repackaging protection. In general, removing all encrypted blocks and restoring the woven code or bypassing the detection code is sufficient to thwart the protection scheme. We aim to make it harder for an attacker to circumvent this scheme than to reimplement the app from scratch.

Static Attacks An attacker may try to remove the protection by statically examining and modifying the code of the protected Android app. The encrypted blocks in the bytecode and native code can easily be detected. However, removal of these sections would lead to an inconsistent state at runtime, and most likely crash the app. The attacker is forced to decrypt the encrypted code. For each block, the decryption key can be derived from the replaced if-condition `if(H(v) == H_c) {...}` by inverting the one-way hash function.

When a brute force approach is used, the attacker has to try each value from the domain of the variable `v`. For primitives in Dalvik bytecode, the domain can be as large as 2^{64} in the case for `long` or `double` variables. Therefore, 2^{63} hash operations and comparisons have to be computed on average to obtain the correct key. Note that in practice, the distribution of such constant values are not uniformly random (see Fig. 3), which can be advantageous for the attacker.

Dynamic Attacks The attacker can not only analyze the protected application statically, but can also transform the protected application. This enables the

attacker to perform more complex attacks. Code can be injected that can perform code analysis or change the behavior of the application at runtime.

The challenge of finding the correct encryption key can be solved without a lot of effort when considering a dynamic attack in which the virtual method invocation that passes the decrypted contents to the class loader is hooked: the method invocation can be modified by an attacker such that its arguments are passed to an arbitrary method. In this newly injected method, the attacker can tamper with the decrypted Dalvik bytecode at runtime. Additionally, the attacker can determine the decryption key for the current encrypted block. Since these encrypted blocks are present in different locations in the bytecode, the attacker has to find enough execution traces to decrypt all code blocks to completely remove the applied protections.

As discussed in Section 3, the repackaging protection code is inserted into the native shared libraries, woven together with parts of the original code of the application. The proposed protection is specifically designed in such a way that bytecode analysis is not sufficient to bypass the protections. Combining application code and integrity checks in the native code requires the attacker to also analyze native code at runtime. Dalvik bytecode allows the attacker to deduce a lot of semantic information due to the verbosity of the Dalvik instructions. Useful information such as variable types or which type of method invocation is performed (private method invocation, interface invocation, class initializer invocation, ...) can be extracted. Machine code, on the other hand, is more difficult to analyze. The usage of pointer arithmetic, self-modifying machine code [7], complex machine instructions or even undocumented instructions[8] can significantly increase the complexity of the analysis. With our approach of forcing attackers to analyze native code dynamically, code obfuscation frameworks operating on native code [5] can be used instead of being limited to bytecode-level transformations.

The integrity of the running application is verified by making use of the C standard library to obtain a file handle of the APK file that corresponds to the running application. The methods that provide the file handle and read its content are provided by Bionic, Android's C library. An attacker might try to hook these library functions by statically overwriting pointers in the Global Offset Table (GOT) or Procedure Linkage Table (PLT) of the ELF binary, which are tables filled by the dynamic linker that contains offsets to procedures of shared libraries. Modifying the relocation information for dynamically linked Bionic functions gives the attacker control over the libraries that are loaded by the dynamic linker. In this way, `fopen()` and `read()` calls can be hooked. This attack could easily be mitigated by not relying on any shared library, but instead only on system calls by passing interrupt vectors to the kernel using an interrupt signal.

Another possible attack scenario involves sandboxing of the protected app in a host app, where the attacker can potentially modify the behavior of the

[8] https://github.com/xoreaxeaxeax/sandsifter/blob/master/references/domas_breaking_the_x86_isa_wp.pdf

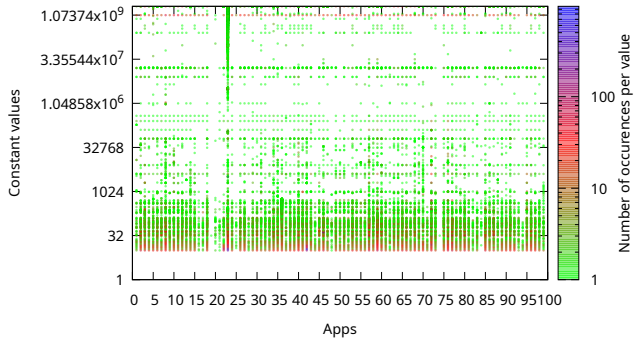

Fig. 3. Distribution of 32-bit constant values encountered during transformation. Apps are taken from the list of most popular applications on Google Play. Constants within the interval $[-9, 9]$ are ignored.

sandboxed application by modifying its memory at runtime, or by running malicious code in a separate thread unbeknownst to the sandboxed app. To be resilient against such attacks, the repackaging protection scheme should detect whether the application is being executed with sandboxing or debugging. This can often be achieved by comparing the result of system calls at runtime with expected values. Another work [1] achieves sandboxing by making use of the `ptrace` system call, often used by debugging tools. The application that is being sandboxed can detect that it is being traced using several techniques such as spawning a subprocess that traces itself, preventing itself from being traced by other processes because a process may only be traced by a single process.

These countermeasures raise the difficulty of the attack, but are no permanent solution. This problem can be reduced to solving the anti-debugging or anti-sandboxing problem. Solving this problem lies outside of the scope of this work, but is actively being researched [8].

6 Evaluation

We evaluate the proposed repackaging protection system using a collection of free apps downloaded from Google Play and F-Droid.[9]

As mentioned in Section 5, the distribution of the constant values in the transformed conditions is important for the resilience of the system against brute-force attacks. Fig. 3 shows the observed distribution of the constants. This serves as an indication of how much effort an attacker needs to invest to statically attack the protection scheme. We can observe that some apps have the same constant values as others, which might help an attacker guess the encryption keys. Note that if a library is being utilized by an app, the constants appearing in the bytecode of that library will be shared amongst all apps which include it.

[9] https://f-droid.org/en/

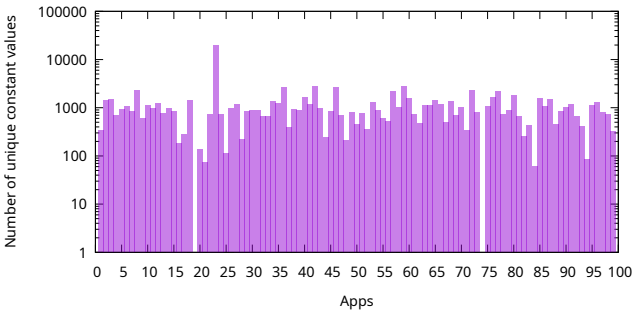

Fig. 4. Number of unique constant values encountered during transformation. Apps are taken from the same sample of apps as Fig. 3.

The unique number of constants within an app is shown in Fig. 4. This figure shows how many blocks can be utilized for transformation. We also observe that for a few apps, no constants have been found. This might be due to Android packers interfering with the static analysis process of finding candidate blocks because the original bytecode has been compressed or encrypted [16]. Additionally, some code obfuscators might artificially increase the number of constants in if-statements which we suspect to be the case for Facebook Messenger (`com.facebook.orca`, app number 23 in Fig. 3 and Fig. 4).

We have measured the runtime overhead when executing a transformed block as opposed to executing the original code block. This overhead was measured on an LG Nexus 5X running the factory image of Android 8.1.0. For each encrypted block, we observed an overhead of approximately 27 ms upon first execution and approximately 3 ms for subsequent executions as shown in Fig. 5. Most of the overhead occurs when an encrypted code block is executed for the first time which involves decrypting the Dalvik bytecode, invoking the class loader to load the decrypted class and decrypting the native code.

The overhead of a sample of apps is shown in Table 1. Each of the apps were given random input for 2 minutes using Monkey.[10] The number of blocks encountered and executed at runtime for the first time are referred to as block misses. Blocks that are executed subsequently, are called block hits. The total overhead per app is computed by multiplying block misses and block hits by the average measured overhead of that type in Fig. 5. The overhead noticed by the user in practice may be lower since multiple threads may execute different blocks at the same time.

Fig. 6 shows how over time, different code paths for an app are taken when providing random input, and thus encountering new transformed blocks which need to be decrypted at runtime. We observe that initially the rate of unique blocks encountered is high, but declines over time as more blocks are already decrypted and loaded into memory.

[10] https://developer.android.com/studio/test/monkey

Table 1. Performance overhead after transformation

	# misses	# hits	total (s)	total (%)
com.aappstech.flpmaps	17	415	1.704	1.42
com.takeaway.android	34	1232	3.939	3.28
org.mozilla.firefox	33	3874	12.513	10.42
com.whatsapp	12	698	2.418	2.01

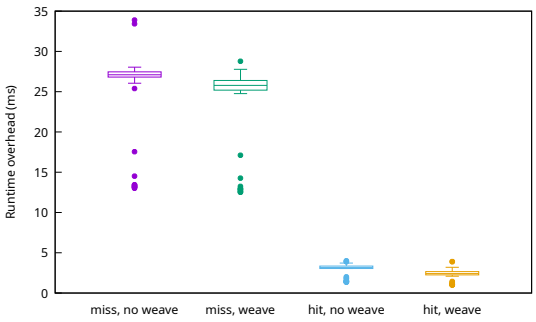

Fig. 5. Boxplot of runtime overhead per encountered encrypted block. The first two boxes illustrate the overhead when a block is first encountered. When a code block is reached that already has been decrypted (referred to as a hit), the overhead is much smaller, as shown in the last two boxes.

We have encountered some issues with the transformation of a portion of apps. An Android APK can contain native libraries for a set of ABIs. Some of the APKs are only provided with native libraries of an ABI that is deprecated in the current NDK toolchain. Some of those are not available in the current NDK version. These apps can therefore not be transformed with our implementation. The developer has to provide a set of shared object files whose ABIs are not deprecated.

We are using the third-party library Soot to perform our transformation on the Jimple intermediate representation, which has some issues[11] when transforming some of the applications from the collection.

We have built a testing workflow which compares the execution of a Java source file before and after transformation. This helped us to discover issues with the implementation and resolve them fairly quickly. Many apps on Google Play are obfuscated with ProGuard, which makes debugging issues with our implementation and Soot's time consuming.

We have evaluated how many apps from our collection are successfully transformed and do not experience unwanted side-effects such as app crashes or thrown exceptions: We have randomly selected 200 apps as a sample from the F-Droid marketplace. 2% of the sample could not be transformed (e.g. when the original APK only contains native code with deprecated ABIs). 7.5% of the

[11] https://github.com/Sable/soot/issues/969

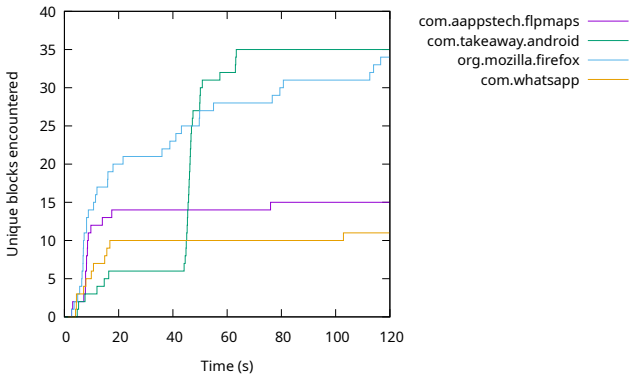

Fig. 6. Unique blocks encountered while running a few apps from Google Play while providing a random input every 100 ms for 2 minutes using Monkey.

sample introduced new exceptions at runtime after transformation. We also evaluated the 100 most popular apps from Google Play. 12% of the sample could not be transformed, and 41% of the sample introduced new exceptions at runtime after transformation. We compared the exceptions thrown by the original app to those thrown by the app after transformation. We made use of Monkey to send a pseudo-random stream of input events to each original and transformed app.

7 Conclusion

In this work, we discussed the limitations of recent work on repackaging protection embedded in apps. We proposed a complete architecture to protect an app from being repackaged by unauthorized actors based on native code. The evaluation shows that the proposed approach only has a limited impact on the performance of the protected apps.

The strength of the system against brute-force attacks depends on the distribution of constants in the candidate blocks. Approaches to insert artificial constants which are compared at runtime to a variable could be inserted to increase the number of encrypted blocks.

This work has focused on a repackaging detection architecture, and shifted the integrity problem from the bytecode level down to the machine-code level. Integrity of the compiled native code in our implementation is important because it is responsible for triggering the response behavior upon detecting an integrity mismatch. As discussed in Section 2, integrity protection of native code has been discussed by the research community in depth. Because of the complexity and difficulty to analyze machine code, circumventing the proposed protection is more difficult than protection based on bytecode. The attacker can either brute-force all the encryption keys and then analyze the machine code or the analysis has to be performed at runtime.

References

1. Bianchi, A., Fratantonio, Y., Kruegel, C., Vigna, G.: NJAS: sandboxing unmodified applications in non-rooted devices running stock android. In: SPSM 2015, Denver, CO, USA. pp. 27–38 (2015)
2. Chang, H., Atallah, M.J.: Protecting software code by guards. In: ACM CCS-8 Workshop DRM 2001, Philadelphia, PA, USA. pp. 160–175 (2001)
3. Hanna, S., Huang, L., Wu, E.X., Li, S., Chen, C., Song, D.: Juxtapp: A scalable system for detecting code reuse among android applications. In: DIMVA 2012, Heraklion, Crete, Greece. pp. 62–81 (2012)
4. Hu, W., Tao, J., Ma, X., Zhou, W., Zhao, S., Han, T.: Migdroid: Detecting app-repackaging android malware via method invocation graph. In: ICCCN 2014, Shanghai, China. pp. 1–7 (2014)
5. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-llvm - software protection for the masses. In: SPRO 2015, Florence, Italy. pp. 3–9 (2015)
6. Luo, L., Fu, Y., Wu, D., Zhu, S., Liu, P.: Repackage-proofing android apps. In: DSN 2016, Toulouse, France. pp. 550–561 (2016)
7. Mavrogiannopoulos, N., Kisserli, N., Preneel, B.: A taxonomy of self-modifying code for obfuscation. Computers & Security **30**(8), 679–691 (2011)
8. Nevolin, I.: Advanced Techniques For Anti-Debugging. Master's dissertation, Ghent University (2017)
9. Ng, Y., Zhou, H., Ji, Z., Luo, H., Dong, Y.: Which android app store can be trusted in china? In: COMPSAC 2014, Vasteras, Sweden. pp. 509–518 (2014)
10. Ren, C., Chen, K., Liu, P.: Droidmarking: resilient software watermarking for impeding android application repackaging. In: ASE '14, Vasteras, Sweden. pp. 635–646 (2014)
11. Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Impeding Malware Analysis Using Conditional Code Obfuscation. In: NDSS 2008, San Diego, CA, USA (2008)
12. Soh, C., Tan, H.B.K., Arnatovich, Y.L., Wang, L.: Detecting clones in android applications through analyzing user interfaces. In: ICPC 2015, Florence/Firenze, Italy. pp. 163–173 (2015)
13. Sun, M., Li, M., Lui, J.C.S.: Droideagle: seamless detection of visually similar android apps. In: Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA. pp. 9:1–9:12 (2015)
14. Vallee-Rai, R., J. Hendren, L.: Jimple: Simplifying Java Bytecode for Analyses and Transformations. Tech. rep. (1998)
15. Wang, X., Jhi, Y., Zhu, S., Liu, P.: Detecting software theft via system call based birthmarks. In: ACSAC 2009, Honolulu, Hawaii, USA. pp. 149–158 (2009)
16. Yu, R.: Android packers: facing the challenges, building solutions. In: Proceedings of the 24th Virus Bulletin International Conference (2014)
17. Zeng, Q., Luo, L., Qian, Z., Du, X., Li, Z.: Resilient decentralized android application repackaging detection using logic bombs. In: CGO 2018, Vösendorf / Vienna, Austria. pp. 50–61 (2018)
18. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: Viewdroid: towards obfuscation-resilient mobile application repackaging detection. In: WiSec'14, Oxford, United Kingdom. pp. 25–36 (2014)
19. Zhou, W., Zhang, X., Jiang, X.: Appink: watermarking android apps for repackaging deterrence. In: ASIA CCS '13, Hangzhou, China. pp. 1–12 (2013)
20. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: CODASPY 2012, San Antonio, TX, USA. pp. 317–326 (2012)