Securing Your App Against Repackaging Attacks

Shubham Raj, Simon Tanner, and Roger Wattenhofer

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

Abstract. Android app repackaging attacks allow an attacker to modify applications and bundle them with malware. In our evaluation we have observed that currently only very few apps are in any way protected against repackaging attacks. We propose an app repackaging protection system that can be easily applied to any Android app after compilation and is therefore easy to adapt by app developers. With an in-built profiling system, we can further reduce the performance impact of the protection system on the app. We show that the proposed system is effective against repackaging attacks.

Keywords: Repackaging protection, App security, Profiling

1 Introduction

Android is used on the majority of mobile phones and users nowadays use the smartphone for everything from browsing the web to shopping and online banking. Therefore, Android phones are an interesting target for malware developers.

A possibility for attackers to distribute malware, is to modify an existing Android app to embed malicious code and distribute it to users. Android uses the APK file format for distribution and installation of applications, which is a Zip archive that includes Dalvik bytecode, resources, and a manifest including essential information about the app such as requested permissions. This format makes it easy for an adversary to unpack an app, modify its contents and repack it. The resulting APK file can then again be distributed to users. This process is known as repackaging. Thus, it is common that malware is hidden within copies of existing popular apps. Previous studies have reported that around 73% of malware samples use some sort of repackaging for their distribution [33].

Apps submitted to Google Play are analyzed for malicious behavior using various techniques before they are made available to the public to keep the end-user safe [2]. However, there are various other app distribution platforms with often weaker to no security measures against harmful apps. In China, one of the largest phone markets in the world, most users do not have access to Google Play and have to use other third-party platforms, many of which are not trustworthy and often distribute repackaged versions of apps [21].

Depending on the modified app, the damage caused by repackaging can be large, even if only few users install it. If a banking app is modified and users install it from an untrustworthy source, the attacker can easily gain access to bank accounts and for example modify banking transfers initiated by the user. Therefore, as the developer of an app, we want to embed the repackaging detection into the app itself. The app can then detect the modification and react appropriately without relying on the detection capabilities of the app stores. Our proposed system can embed the repackaging detection capability with an almost negligible impact on the performance of the app.

During the evaluation of our protection system, we noticed that only very few applications (apps developed by Google, a European bank and an antivirus company) out of more than 400 tested apps have any kind of protection against repackaging built into them. This shows either a lack of awareness about the risk, or the interest of developers to protect against such attacks is limited due to the effort required to implement it robustly. We hope an open-source and easy to use tool will change that. Our system can create a protected version of the application within a few minutes.[1]

2 Related Work

Many different ways to detect repackaging of Android apps have been proposed. Many approaches rely on a central authority that detects the repackaging, while decentralized approaches rely on detection on the user devices.

In a centralized approach, the application distribution platform or another centralized entity is responsible for monitoring the mobile apps submitted by developers. The centralized approaches typically analyze features from the collection of apps to find repackaged apps. These mechanisms try to identify similar features between apps such as their call graphs [15], instruction sequences [32], view graphs [29], etc. Another possibility is to embed software watermarks into the app [31]. These approaches require the maintainers of the distribution platforms to remove the repackaged apps promptly. The main challenge with this approach is to come up with a reliable and scalable solution. Also, these mechanisms are only implemented by some distribution platforms, possibly due to the resources and efforts requirements, as many platforms have been found to distribute repackaged versions of apps [21].

In decentralized systems, the idea is to bundle the repackaging protection mechanism with the application itself. The app can therefore detect repackaging at runtime. If it detects some kind of tampering, a response mechanism is executed which either disrupts the normal behavior of the app or informs the user. This has the advantage of distributing the repackaging detection workload. A simple way of implementing this is by calculating the hash of the certificate used to sign the application at runtime and compare it with a hard-coded hash value. During our evaluation, we found that most apps developed by Google implement certificate hash comparison to detect repackaging. However, without

[1] The source code of our system is available at `https://drive.google.com/drive/folders/16N5KRLmlPTFAz5sZ_I3YMbx_ojPIENWY?usp=sharing` (will be published on GitHub for the final version of this paper)

protecting the hard-coded hash value and the detection code from tampering, such protections can easily be removed by an attacker.

Several mechanisms that assert the software integrity at runtime have been proposed. For instance, Droidmarking [23] proposes a non-stealthy repackaging detection mechanism, in which a standalone Droidmarking app installed on the end-user device is responsible for validating the integrity of the applications protected by Droidmarking. This approach also requires the support of the distribution platform to statically scan the submitted apps for tampering. Another work [12] proposes a mechanism that is based on a distributed scheme in which multiple pieces of code called *guards* are inserted at various points of a program to assert the integrity by computing checksums of code segments and comparing them to hard-coded precomputed values.

Stochastic Stealthy Network (SSN) [19] is a mechanism where several detection and response nodes are inserted into the app. The detection nodes are responsible for verifying sub-strings of the public key of the certificate used to sign the application. Once repackaging is detected, one of the response nodes injects logical malfunctions leading to negative user experiences to discourage users from continuing to use the app. Zeng et al. proposed *BombDroid* [28], a decentralized repacking protection mechanism based on encrypted code blocks and logic bombs. The code responsible for detecting repackaging is stored within an encrypted code block in bytecode. Encrypted blocks are decrypted using a key derived at runtime when trigger conditions are satisfied. As some statements of the original application code are woven into the encrypted block, together with repackaging detection and response code, the attacker cannot simply strip the protection without corrupting the app execution. In [26] it was shown that the approaches of SSN and BombDroid can be circumvented by an attacker using bytecode instrumentation and a mechanism based on repackaging detection in native code was proposed.

Related to these approaches are obfuscation and other ways to prevent an attacker from modifying an app in a useful way. DIVILAR [30] converts apps to a randomized virtual instruction set and embeds an interpreter into the app. This increases the effort for an attacker as the app is not in the expected instruction set.

3 System

We propose a repackaging protection system that embeds the detection into the apps themselves. Therefore, a protected app can detect repackaging attacks at runtime. The added protection should not impact the user experience and be easy to embed in the normal workflow of building and publishing an app. The system takes an Android app as an APK file containing compiled bytecode and resources as an input and outputs a protected app in an APK file with embedded security measures against repackaging. The protected app can be distributed to users and distribution platforms regardless of whether or not they enforce security policies to protect against repackaging attacks.

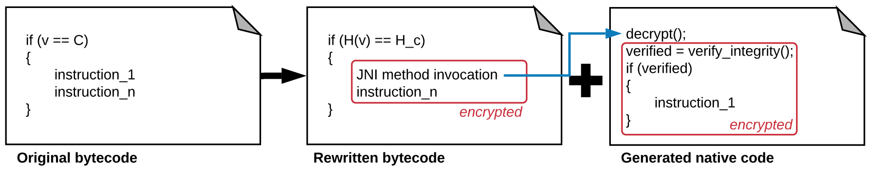


Fig. 1: Transformation of candidate code blocks. The if-condition is rewritten using conditional code obfuscation, a integrity check is added in native code and bytecode and native code get encrypted.

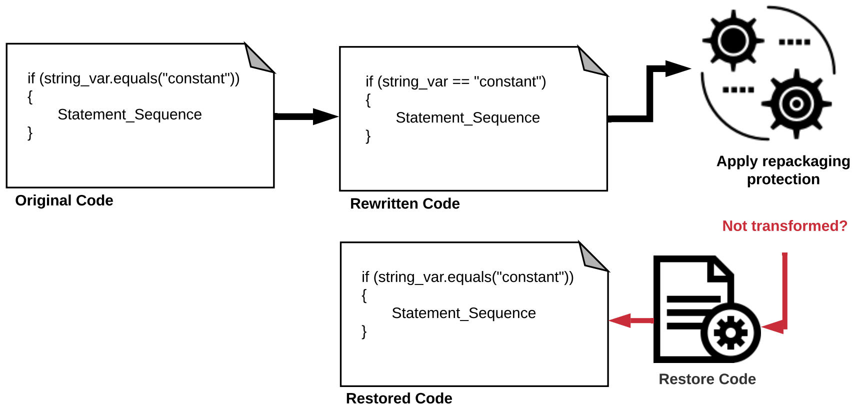
The system partly relies on encrypted native code to make it hard to circumvent the added protections as in [26]. The system instruments the compiled application and adds integrity checks at various locations in a robust manner, which at runtime detect whether the application being run is repackaged. If detected, a response mechanism is activated. An application is instrumented on a simplified bytecode level (known as *Jimple* [27]) to add encrypted code blocks. Integrity checks are added in encrypted native code. Since the system embeds the protection on the bytecode level after the application has been compiled as opposed to the source code level, it can be applied to any application that targets the Java Virtual Machine (JVM) or Android Runtime (ART), such as Kotlin or Scala.

The protection system uses a technique called *Conditional Code Obfuscation* [25]. The condition `v == C` for a variable `v` and integer constant `C` is transformed to the semantically equivalent `H(v) == H_C` where `H()` is a one-way hash function and `H_C` is the constant equivalent to `H(C)`. It is therefore hard to find out for which value of `v` this modified condition is true. Using this technique, we can derive a key from the constant `C` to encrypt the true branch of if-bodies. Figure 1 illustrates the protection scheme. With this technique, not all parts of the bytecode are directly visible to the reverse engineer.

At runtime when the condition is true and the program flow enters the if-body, the key can be derived from variable `v`, since it contains the value `C` at this point. We can therefore decrypt the bytecode, load it as a new class using `dalvik.system.InMemoryDexClassLoader` and execute it.

The code responsible for repackaging detection and response is inserted in low-level native code, which is also encrypted for further protection against static analysis. In the encrypted bytecode, a call to the native decryption function is added. At runtime, the native code is decrypted and the integrity of the application is checked. The symmetric key used to encrypt the native code is stored inside the encrypted bytecode block. To detect repackaging, the digests of various segments of Dalvik Executable (.dex) files are compared to precalculated hash values of the original application.

To prevent an attacker from stripping the code responsible for integrity assertion, a method call already present in the original if-body is moved to the



```
if (string_var.equals("constant"))
{
      Statement_Sequence
}
```

```
if (string_var == "constant")
{
      Statement_Sequence
}
```

```
if (string_var.equals("constant"))
{
      Statement_Sequence
}
```

Fig. 2: Transformation of string equality. If the encrypted code block cannot be created, the code gets restored.

encrypted native code section. The moved method call is rewritten to an equivalent JNI method call, which has the same effect as being executed directly from the bytecode. Thus, not executing the native method would prevent the normal execution of the app.

Performing modifications directly on Dalvik bytecode level can be error-prone and cumbersome, therefore, our system uses the Jimple [27] intermediate representation and the Java API of the Soot framework [24] to assist static analysis and perform modifications.

So far, the described system transforms if statements with integer condition `v == C` to generate encrypted code blocks. To increase the number of encrypted blocks and find constants that are stronger against brute-force attacks, we adapt the method to also support different types of constants such as float, double, long and strings. We also extend the support to switch statements, not-equal (`!=`) statements and primitive wrapper object equality tests.

3.1 String equality

In Java, the `equals()` method is used to compare the content of a string object to another string and `==` is used for reference comparison. We are interested in occurrences of `equals()` used for comparison of a string object to a constant, which can be used to generate encrypted code blocks. The code labeled as *Original Code* in Figure 2 is rewritten into the form represented in *Rewritten Code*. The rewritten code block is then passed to the core module to create an encrypted code block using conditional code obfuscation as explained before. The hash of the string constant and the hash of the string variable will be compared. This ensures the semantic equivalence between the original and transformed application. We perform additional steps to ensure the encrypted code can also work where `equalsIgnoreCase()` is used to compare a string variable to a string constant without case considerations.

Even though the intermediate rewritten code is not correct nor semantically equivalent to the original code, after an encrypted code block has been generated for the rewritten code, it will be equivalent again. This allows us to use the same implementation of the repackaging protection for different data types. In some cases the repackaging protection cannot transform a particular block and we restore the original code.

The transformation of string comparisons leads to the generation of many new encrypted blocks with very strong constants.

3.2 Float, double, long equality

For comparing a float, double or long constant to a variable for equality, Jimple uses the instructions `cmp` and `cmp-long` [10]. Therefore, we find all statements where the result of such a comparison is used in an if-condition. The Jimple code is then rewritten such that the creation of the encrypted block can be performed the same way as for integer constants. Similar to the string transformation, rewritten Jimple code is restored to original Jimple code in case the encrypted code block cannot be created.

The 64-bit long and double data types lead to increased entropy of constants in comparison to only integers and can have very large values. However, note that the distribution of constant values might not be very uniform in practice as noticed during our evaluation.

3.3 Primitive wrapper object equality

Apart from direct equality check for primitive data types using double equal sign (`==`), Java also knows wrapper classes that wrap primitive data types. An object of type Double contains a single field whose type is double. All of these wrapper classes extend `java.lang.Number` and override `equals()`. The transformation for these comparisons works the same as in the string case.

3.4 Switch statement

We also partially support the generation of encrypted code blocks for `Switch` statements found in the application code. We have implemented the transformation to support both *TableSwitch* and *LookupSwitch* statements. The TableSwitch instruction is used when the cases of the switch can be efficiently represented as indices into a table of target offsets, while if the cases of the switch are sparse, the table representation of the TableSwitch instruction becomes inefficient in terms of space and hence, the LookupSwitch instruction may be used instead [8].

We rewrite the switch statement into a series of nested if-else statements which are then used to generate encrypted code blocks. However, not all switch statements are rewritten to limit the performance overhead. Switch statements with many cases, small constants or with `or` cases, where a single statement sequence has multiple case labels, are not rewritten. Switch statements can have different representations on the Jimple level which makes it almost impossible to detect the end of a switch block in some cases [6]. Therefore, we cannot rewrite all encountered switch statements.

The ability to create encrypted blocks out of the converted switch statements helps to further strengthen the security by increasing the number of encrypted blocks.

3.5 StatsAnalytics

We want to ensure that our protection method does not introduce a large performance overhead. Therefore, we have implemented a profiling system, which we call `StatsAnalytics`. Using this system, we can profile apps and detect the most frequently invoked methods. The app can be transformed again while excluding the often executed methods from transformation.

During the transformation process, the application developer can choose to embed the StatsAnalytics system into the application along with the protection scheme. The analytics system then records the execution count of encrypted blocks at runtime and submits the collected statistics to a server every minute. We have implemented a PHP based server to receive and store the submitted data. The application with the embedded StatsAnalytics can be distributed to the end user or can be given to alpha/beta testers. In this way, the system can collect real usage data or one can use monkey [9] to send a pseudo-random stream of inputs to the application and collect statistics.

Once sufficient data has been collected, the developer can use provided scripts to aggregate the statistics and find the most frequently executed methods. Two different statistics are calculated; the minute impact represents encrypted methods which have been executed most during any minute and the total impact methods that have been executed the most overall. A list of hot methods can then be created and passed to the repackaging protection system to transform the application again while ignoring the hot methods. Therefore, the performance overhead can be significantly reduced based on real usage pattern.

4 Security Analysis

The goal of the system is to prevent an attacker from circumventing the added protections and successfully repackage the app. We assume the attacker can get the APK file of the transformed application and is able to decompile the bytecode using available tools. However, the adversary does not have access to the unprotected APK file or to the source code of the unprotected application. An attacker can then perform static analysis or instrument the bytecode to try to remove the encrypted blocks. The attacker can also instrument the native library responsible for performing integrity checks.

The attacker is also able to perform a brute force attack on the decryption key with resources that are reasonable in practice. An attacker can additionally perform dynamic analysis (including debugging, hooking method calls, dynamic monitoring, sandboxing, etc.) on both bytecode and native code to inspect the execution of the application.

4.1 Static Attacks

An attacker may try to remove the protection by statically analyzing the protected application. The encrypted code blocks on both bytecode and native code level can easily be detected, however, removing them is not a sensible strategy as parts of the code from the true branch are in the encrypted bytecode and encrypted native code. Removal of these blocks would lead to an inconsistent state at runtime.

Symbolic execution [11] or path exploration techniques [17] are not helpful either as we depend on conditional code obfuscation. The key to decrypt the encrypted block is derived at runtime. For each encrypted block, the decryption key can be derived from the replaced if-condition: `H(v) == H_C` where `H` is a cryptographic one-way hash function and `H_C` is the hashed constant. The decryption key is derived from `v` at runtime so the attacker has to brute force the value `C` such that `H(C) == H_C`. This process needs to be performed for every encrypted code block in an application.

We support the transformation of different data types. Variables can be either of primitive data types (i.e., int, long, float, double) or non-primitive data types (i.e., string) which tend to be even further resistant to a brute force approach. During the evaluation we found constants with an entropy of up to 8000 bits. However, in practice the constants are not uniformly random distributed. We also found that our transformation process creates an average of 540 encrypted blocks during the evaluation of more than 400 applications, each of which has to be brute forced to completely strip away the protection from a single application.

4.2 Dynamic Attacks

An alternative way to attack is to dynamically execute and transform the protected application at runtime. This allows the attacker to perform more complex attacks such as memory dumping and runtime debugging. The attacker might not need to perform a brute-force attack to find the decryption key and instead hook the virtual method invocation that passes the decrypted contents to the class loader. The attacker could analyze and modify the behavior of the decrypted bytecode at runtime. Additionally, the attacker can determine the decryption key of the current encrypted block. However, given that there are a number of encrypted blocks at various locations throughout the complete application, the attacker has to find enough execution traces that together lead to the decryption of all code blocks to completely remove the applied protections. Also as the integrity checks are inserted into native code, woven together with parts of the original code of the application, it is not sufficient to only analyze the bytecode to strip the protection.

We have applied several checks to resist against dynamic code analysis and debugging at runtime. We monitor `TracerPid` of the process status file to detect the debugging of the process using the `ptrace` system call which is commonly used by code analyzers, debuggers, etc. We have also applied a mechanism in which the main process forks a child process to trace the parent process as a debugger, as an additional layer of resistance against the native code debugging. This mechanism exploits the fact that a process can only be debugged by a single process using `ptrace` at a time. These checks can easily be extended with root or debugging environment detection methods and detection of common instrumentation frameworks or method hooking techniques on the native code level. Moreover, encrypting these checks along with integrity assertion requires additional efforts in order to analyze the implemented checks.

By forcing an attacker to dynamically analyze native code, we can utilize native code obfuscation frameworks [16] which use instruction substitutions, bogus control-flow insertion, control-flow flattening, etc. to obfuscate the native code instead of relying on bytecode based obfuscation which significantly increases the complexity of the analysis. The usage of self-modifying machine code [20], complex machine instructions or even undocumented instructions [14] can increase the complexity of successful attack even further.

The attacker could also use vtable hijacking, GOT/PLT, LD_PRELOAD or other dynamic instrumentation techniques to circumvent the protection offered by our transformation using tools like ARTDroid [13], Frida [4], etc. However, to instrument the application, many of these tools either require injecting some code into the application or root access on the device to properly work. In our case, instrumenting the application code will activate integrity based repackaging detection response, until those are taken care of separately and our system defenses can easily be extended with root detection measures. We could also extend the system to detect Frida or other similar frameworks by checking the environment for related artifacts such as relevant binaries, process, package files, etc. [22].

It is widely believed that a software-based protection mechanism can eventually be bypassed as long as the adversary is determined enough. The purpose is to make it as hard as possible to perform those attacks to discourage the attacker as it is assumed that an adversary would only be interested in repackaging an application if it is cost-effective. Given the open-source nature of our developed security tool, the security of the tool can be reviewed and further improved by the wider community as well.

5 Evaluation

We evaluate our repackaging protection system using a large set of apps. With statistics about apps on Google Play from App Annie [1], we downloaded all top free apps from Google Play using the package IDs of the applications. We

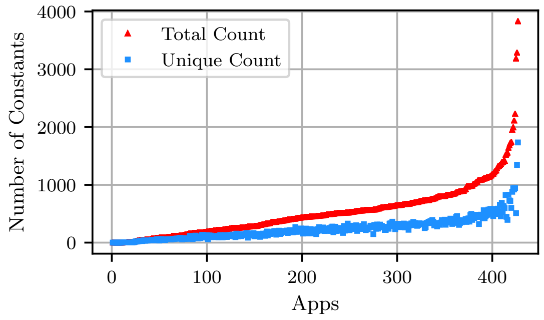


Fig. 3: Number of total and unique constants used during transformation for each application. The apps are ordered by the total number constants.

separated the games from the downloaded applications using a list of games from App Annie as during the evaluation we found that many of games only contain little bytecode, as they mostly use native code. We test the rest of the applications. We also downloaded around 30 applications of top banks for testing. In total we tested around 500 applications, out of which few were ignored during the transformation process due to the use of outdated and deprecated ABIs (i.e., armeabi, mips, mips64) which are no longer supported by Android NDK or because of issues (e.g., [7,3]) in the Soot framework while processing the applications. Hence, in total, more than 400 top applications were evaluated.

5.1 Testing System

Monkey [9] is often used for testing systems, it sends a pseudo-random stream of input events to the application. However it does not systematically try out all the possibilities of the user interface of an app. Therefore, we used DroidBot [18] for the testing. It can generate UI-guided test inputs. We send input events to both original and transformed application and record detailed information including system memory log, transformation logs, screenshot after each event, system logs, etc.

We compare the logs collected for both original and transformed applications to find new exceptions, determine success rate, etc. We also use perceptual hashing along with manual tests to compare the screenshots collected for each event in both original and transformed application. Due to the model-based deterministic nature of inputs generated by DroidBot, the input events generated and the functionality path traversed in both original and transformed application remain the same. The manual comparison is necessary for cases where dynamic content is loaded, where content is dependent on the time, etc. In these cases the screenshots differ but we can only determine manually whether the functionality is still correct. The screenshot-based comparison also allows looking for traces of mechanisms suggested in literature which distort the user interface at runtime on repackaging detection to discourage users from using the application.

5.2 Results

We evaluate our system using more than 400 applications downloaded from Google Play. As noted in Section 4 uniqueness, distribution, and entropy of the constants used during transformation is critical for the resilience of the system against brute-force attacks. We also noted that the total number of encrypted blocks is important for the security promises, as an attacker using the brute-force approach needs to find the constant for each of the encrypted code blocks in an application.

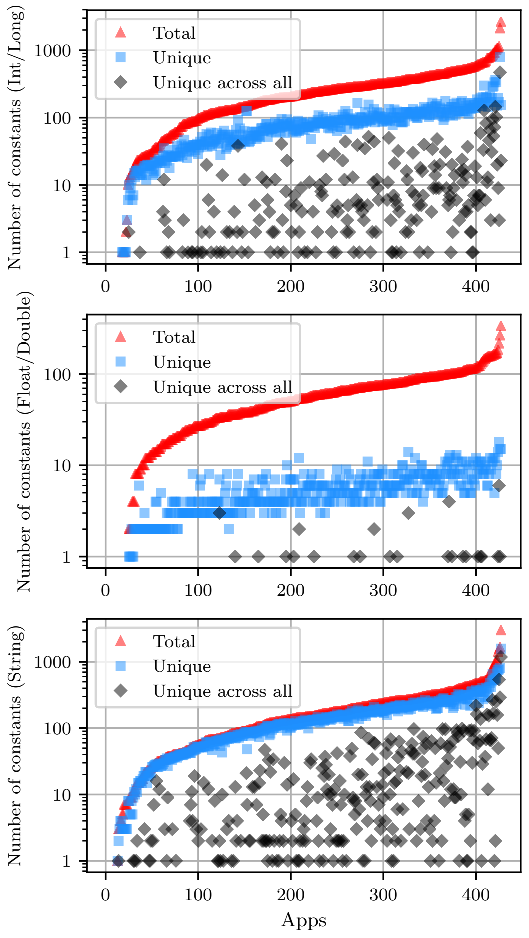
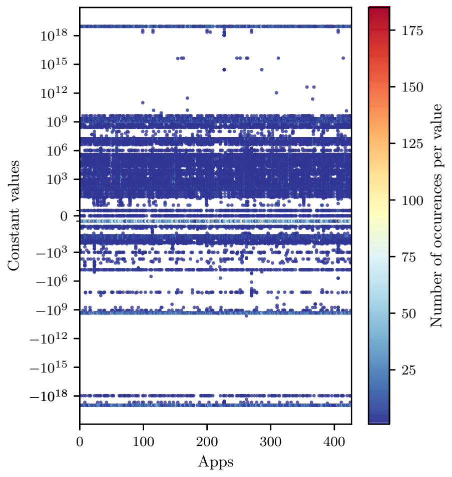
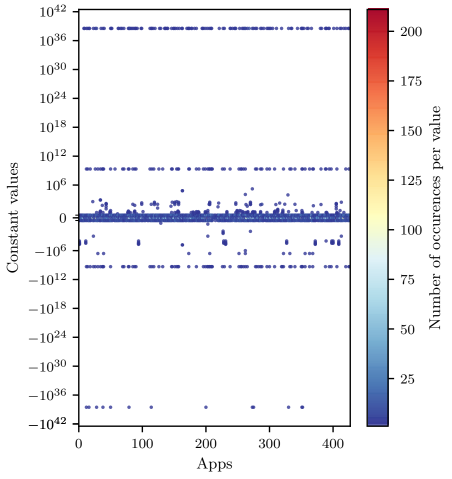


Fig. 4: Number of total constants, unique constants in the same application and unique constants across constants of all other applications in the evaluation.



(a) Integer and long values



(b) Float and double values

Fig. 5: Distribution of integer and long values and float and double values used for transformation. The plots have a symmetric logarithmic y-axis.

We recorded the value of each constant used in the creation of encrypted code blocks during the transformation phase. Figure 3 shows the total number of constants and the number of unique constants for each application used during the transformation. As we can see from this plot, apps with more constants also contain more unique constants. If an app has many duplicate constants, this makes brute-force attacks easier, as the attacker can first try already found constants against an encountered encrypted block.

To give a better idea of the usefulness of the constants of different data types, we additionally separate the constants by the data type (int/long, float/double and string). The number of total constants, unique constants in the same application and the unique constants across constants of all other tested applications separated by the data type is shown in Figure 4. This figure shows that strings, in general, have more unique constants in each of the application, while float and double in general have fewer unique constants in comparison to the other data types.

The number of unique constants in the same application is calculated by counting the number of constants in the set of total constants. The unique constants across constants of all other applications is calculated for an application `A` by taking the set difference between constants of `A` and constants of all applications except `A`. These constants ensure that even if an attacker gathers the list of constants from a large number of other applications (in our case more than 400 apps) and then uses this list to perform a dictionary attack on the protected application, the success rate would not be very high due to the uniqueness of constants of one application across the constants of all other applications. It also

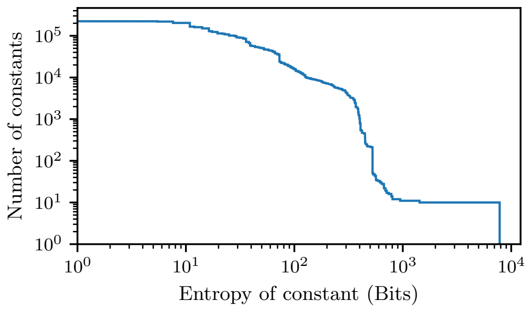


Fig. 6: Reversed cumulative histogram of entropy of constants used during transformation. The y-value for a given x-value shows the number of constants that have an entropy higher than that x-value.

shows that the higher the number of total constants (i.e., the total number of blocks), the higher is also the number of unique constants in the same application. Also the number of unique constants across all apps is generally larger for apps with more constants, however the variation between apps is much higher.

Another important property of the constants when considering a brute-force attacker is the distribution of the values across the domain of the data type. The distributions of integer/long and float/double based constants are shown in Figure 5a and Figure 5b. The integer/long values are more evenly distributed across the range of possible values than float/double. In both plots some lines of often occurring constants are visible.

We also calculated the entropy of constants which is a common practice to measure the strength of a password (i.e., the constant in our case). The entropy for each constant is calculated as `log2(sizeOf(pool))` $\times$ `length`, where `length` is the number of characters in the constant and the pool is the total character set of which a constant is made from.

Figure 6 shows the entropy of constants used during the transformation. We observe that the calculated entropy goes as high as 8000 bits, and the highest concentration is between 100-1000 bits of entropy. String constants result in the highest entropy due to use of alphanumeric characters along with symbols.

From these results we can conclude that especially string and integer/long constants are valuable for the creation of encrypted blocks.

The total number of constants in Figure 3 is also equal to the total number of created encrypted blocks in the tested applications. Our transformation system created about 540 encrypted blocks on average for the applications tested during the evaluation, while for some applications it was more than 3000 blocks.

The total number of blocks created in an application roughly correlates to the size of the bytecode as can be seen in Figure 7. The total number of encrypted blocks of an application does not only depend on the size of the code base but also depends on the complexity and whether the application has been packed.

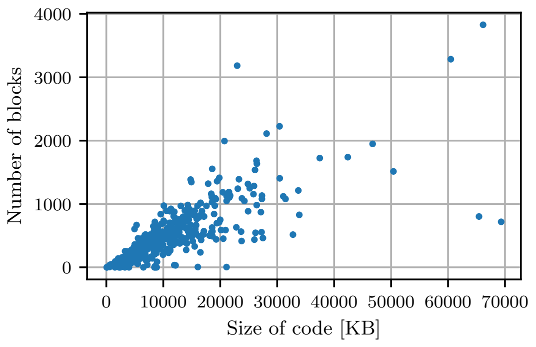


Fig. 7: Correlation between the size of bytecode (classes*.dex files) in kB and the number of generated blocks. Larger apps allow to embed more encrypted blocks.

We observed, that for very few apps, no encrypted blocks could be created. This might be due to Android packers interfering with the static analysis because the original bytecode has been compressed or encrypted.

We also calculate the increase in the size of the application. The median increase is only 12% across the set of applications.

Most important for the user experience is the impact of the repackaging protection system on the performance. We therefore measure the runtime overhead when executing an encrypted block as opposed to executing an original code block with an application specially developed for this purpose. The overhead is measured on a Xiaomi Mi A2 running the factory image of Android 9. Figure 8 shows the overhead in four different situations. Most of the overhead occurs when an encrypted block executes for the first time (called miss in the figure) which involves decrypting the bytecode, invoking the class loader to load the decrypted class and decrypting the native code. Subsequent executions (hits) of the decrypted block result in an almost negligible performance impact. The other distinction is whether a method invocation has been moved from the bytecode to the native code (weave). This is only possible if a suitable instruction was encountered in the original bytecode.

The overhead of a sample of tested apps is shown in Table 1. Each app was given random inputs for 2 minutes using Monkey. The number of blocks encountered and executed for the first time is referred to as block misses, while subsequent block executions are called block hits. The total overhead is computed by multiplying block misses and block hits with the average measured overhead of that type in Figure 8.

We notice that for some applications, we have a high number of block hits (i.e., execution of already decrypted blocks), even though the count of block misses is not significant due to the transformation of hot methods. This implies that the transformation of hot methods should be avoided by utilizing StatsAnalytics. For the evaluation we have not used StatsAnalytics as many profiling

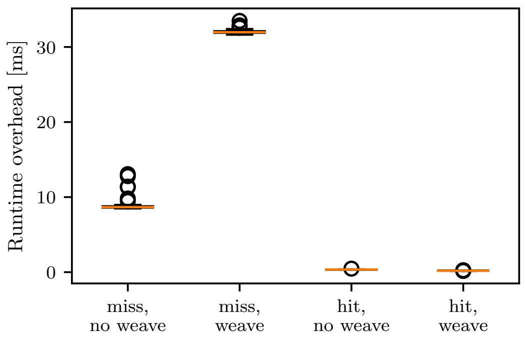


Fig. 8: Boxplot of runtime overhead per encountered encrypted block. The first two boxes illustrate the overhead when a block is first encountered. When a code block is reached that already has been decrypted (referred to as a hit), the overhead is much smaller, as shown in the last two boxes.

traces have to be collected beforehand. Table 1 shows that the overhead is only 2.91% at most even without using StatsAnalytics.

Figure 9 shows how over time, different code paths for an app are taken when providing random input using Monkey, and thus encountering new transformed blocks that need to be decrypted at runtime. We observe that initially, the rate of encountering new blocks is high, but declines over time as more blocks are already decrypted and loaded into memory.

As noted above, we ignore some applications during the transformation due to the use of deprecated ABIs for native libraries which are no longer supported in the current Android NDK version. These apps can therefore not be transformed directly, however, the developer can provide a set of shared object files whose ABIs are not deprecated to facilitate the protection of the app.

We have been able to achieve 99% success rate for transformation of the supported applications. We checked whether no new exceptions were generated due to transformation of the application and all functionalities of the transformed application work as expected by checking the runtime logs for new exceptions/errors and comparing the captured screenshots. We ignore the runtime exceptions which are due to bugs in the Soot framework (e.g., [5]) itself while calculating the success rate. The reason for the few introduced additional errors is the interference of the Soot framework during the described restoration process for some edge cases. The success rate is 100% if we only use if statements which use integer based comparison to create encrypted blocks.

During the evaluation of more than 400 applications, it also became evident that only very few applications (less than 10 apps in total developed by Google, an antivirus company and a European bank) implemented any kind of protection against repackaging attacks. Google utilizes a kind of certificate comparison to detect repackaging in most of its applications.

Table 1: Performance overhead after transformation of some sample apps. Shows the number of executions of not yet decrypted code blocks (miss), already decrypted blocks (hit) and the corresponding time overhead in seconds and in percent of the total time.

App Id	**# miss**	**# hit**	**total (s)**	**%**
com.apple.android.music	26	233	0.58	0.48
com.axs.android	14	12356	3.49	2.91
com.expedia.bookings	6	7774	2.14	1.78
com.google.android.apps.authenticator2	16	935	0.56	0.46
com.lookout	25	8196	2.63	2.19
com.twitter.android	33	2546	1.32	1.10
com.yahoo.mobile.client.android.mail	32	274	0.71	0.59
de.postbank.finanzassistent	10	702	0.38	0.31
it.feio.android.omninotes	41	1027	1.09	0.90
naman14.timber	25	3200	1.33	1.11

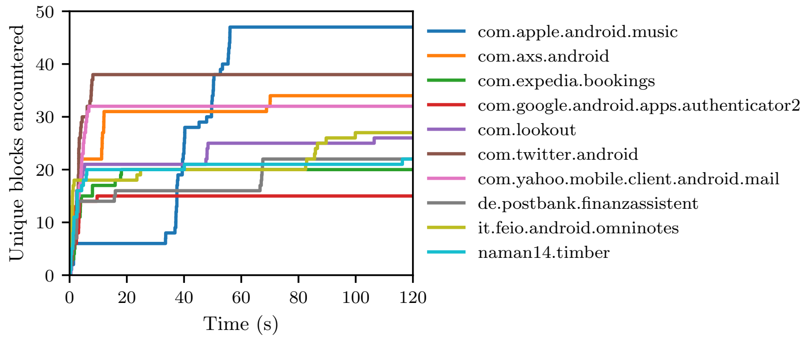


Fig. 9: Unique blocks encountered while running a few apps of the test set while providing a random input every 100 ms for 2 minutes using Monkey

6 Conclusions

Decentralized mechanisms to protect apps from repackaging attacks bring many advantages in comparison to a centralized scheme, but developing a resilient mechanism is hard.

In this work, we have proposed a security tool that can be applied to any Android app. The evaluation shows that the resulting transformation system is effective to guard against repackaging attacks and efficient for practical usage. We achieved 99% success rate for the transformation of applications.

Utilizing our testing system, we also surveyed whether any repackaging protection measures are implemented by the tested apps. Even most apps developed by big tech companies and banks do not implement any repackaging protection. Hopefully with our security tool, more developers and companies will protect their applications against repackaging attacks.

References

1. App Annie — The App Analytics and App Data Industry Standard', `https://www.appannie.com/en/`, accessed 2021-05-12
2. Cloud-based protections — Play Protect, `https://developers.google.com/android/play-protect/cloud-based-protections`, accessed 2021-05-12
3. ConcurrentModificationException in SootClass.getMethodUnsafe, `https://github.com/soot-oss/soot/pull/1116`, accessed 2021-05-12
4. Frida - dynamic instrumentation framework, `https://frida.re/`, accessed 2021-05-12
5. Instrumented app crash with com.google.android.gms, `https://github.com/soot-oss/soot/issues/1171`, accessed 2021-05-12
6. [soot-list] end of a switch block - mailing list, `https://mailman.cs.mcgill.ca/pipermail/soot-list/2009-June/002327.html`, accessed 2021-05-12
7. Soot parsing many app failed for the same reason : Invalid value type for primitibe operation, `https://github.com/soot-oss/soot/issues/1168`, accessed 2021-05-12
8. Switches: Compiling for the java virtual machine, `https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-3.html#jvms-3.10`, accessed 2021-05-12
9. UI/Application Exerciser Monkey — Android Developers, `https://developer.android.com/studio/test/monkey`, accessed 2021-05-12
10. Bartel, A., Klein, J., Le Traon, Y., Monperrus, M.: Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis. pp. 27–38. SOAP '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2259051.2259056
11. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically Identifying Trigger-based Behavior in Malware, pp. 65–88. Springer US, Boston, MA (2008)
12. Chang, H., Atallah, M.J.: Protecting software code by guards. In: Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001, Philadelphia, PA, USA, November 5, 2001, Revised Papers. Lecture Notes in Computer Science, vol. 2320, pp. 160–175. Springer (2001). https://doi.org/10.1007/3-540-47870-1_10
13. Costamagna, V., Zheng, C.: Artdroid: A virtual-method hooking framework on android art runtime. In: IMPS@ESSoS (2016)
14. Domas, C.: Breaking the x86 isa (2017), `https://github.com/xoreaxeaxeax/sandsifter/blob/master/references/domas_breaking_the_x86_isa_wp.pdf`
15. Hu, W., Tao, J., Ma, X., Zhou, W., Zhao, S., Han, T.: Migdroid: Detecting app-repackaging android malware via method invocation graph. In: 2014 23rd International Conference on Computer Communication and Networks (ICCCN). pp. 1–7 (Aug 2014). https://doi.org/10.1109/ICCCN.2014.6911805
16. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM – software protection for the masses. In: Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015. pp. 3–9. IEEE (2015). https://doi.org/10.1109/SPRO.2015.10
17. Kang, B., Yang, J., So, J., Kim, C.Y.: Detecting trigger-based behaviors in botnet malware. In: Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems. pp. 274–279. RACS, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2811411.2811485

18. Li, Y., Yang, Z., Guo, Y., Chen, X.: DroidBot: A Lightweight UI-guided Test Input Generator for Android. In: Proceedings of the 39th International Conference on Software Engineering Companion. pp. 23–26. ICSE-C '17, IEEE Press, Piscataway, NJ, USA (2017). https://doi.org/10.1109/ICSE-C.2017.8
19. Luo, L., Fu, Y., Wu, D., Zhu, S., Liu, P.: Repackage-proofing android apps. In: 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016. pp. 550–561. IEEE Computer Society (2016). https://doi.org/10.1109/DSN.2016.56
20. Mavrogiannopoulos, N., Kisserli, N., Preneel, B.: A taxonomy of self-modifying code for obfuscation. Comput. Secur. **30**(8), 679–691 (Nov 2011). https://doi.org/10.1016/j.cose.2011.08.007
21. Ng, Y.Y., Zhou, H., Ji, Z., Luo, H., Dong, Y.: Which android app store can be trusted in china? In: Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference. pp. 509–518. COMPSAC '14, IEEE Computer Society, Washington, DC, USA (2014). https://doi.org/10.1109/COMPSAC.2014.95
22. Owasp: Owasp/owasp-mstg - reverse engineering tools detection, `https://github.com/OWASP/owasp-mstg/blob/master/Document/0x05j-Testing-Resiliency-Against-Reverse-Engineering.md#testing-reverse-engineering-tools-detection-mstg-resilience-4`, accessed 2021-05-12
23. Ren, C., Chen, K., Liu, P.: Droidmarking: resilient software watermarking for impeding android application repackaging. In: ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. pp. 635–646. ACM (2014). https://doi.org/10.1145/2642937.2642977
24. Sable: Soot - a java optimization framework, `https://github.com/soot-oss/soot`, accessed 2021-05-12
25. Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: NDSS (2008)
26. Tanner, S., Vogels, I., Wattenhofer, R.: Protecting Android Apps from Repackaging Using Native Code. In: 12th International Symposium on Foundations & Practice of Security (FPS), Toulouse, France (November 2019)
27. Vallée-Rai, R., Hendren, L.J.: Jimple: Simplifying java bytecode for analyses and transformations (1998)
28. Zeng, Q., Luo, L., Qian, Z., Du, X., Li, Z.: Resilient decentralized android application repackaging detection using logic bombs. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018. pp. 50–61. ACM (2018). https://doi.org/10.1145/3168820
29. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: Viewdroid: towards obfuscation-resilient mobile application repackaging detection. In: 7th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec'14, Oxford, United Kingdom, July 23-25, 2014. pp. 25–36. ACM (2014). https://doi.org/10.1145/2627393.2627395
30. Zhou, W., Wang, Z., Zhou, Y., Jiang, X.: DIVILAR: diversifying intermediate language for anti-repackaging on android platform. In: Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA - March 03 - 05, 2014. pp. 199–210. ACM (2014). https://doi.org/10.1145/2557547.2557558
31. Zhou, W., Zhang, X., Jiang, X.: Appink: watermarking android apps for repackaging deterrence. In: 8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013. pp. 1–12. ACM (2013). https://doi.org/10.1145/2484313.2484315

32. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the Second ACM Conference on Data and Application Security and Privacy. pp. 317–326. CODASPY '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2133601.2133640
33. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA. pp. 95–109. IEEE Computer Society (2012). https://doi.org/10.1109/SP.2012.16