draft paper ilham

by Elly Warni

Submission date: 09-Aug-2023 01:21PM (UTC+0700)
Submission ID: 2117092385
File name: ilham-2023-final2.docx (3.63M)
Word count: 2710
Character count: 14714

Signature Verification Based on Dex CRC and Blake2 Algorithm to Prevent Reverse Engineering Attack in Android Application

Abstract— The rapid growth of Android applications has led to more cyber-crime cases, specifically Reverse Engineering attacks on Android apps. One of the most common cases of reverse engineering is Application repackaging, where the application is downloaded via the Play Store or the official website and then repackaged with various additions or changes. One of the ways to avoid Application Repackaging attacks is to check the signature of an application. However, hackers can manipulate the application by adding a hook, i.e., replacing the original function for getting signatures with a new modified function in the application. In this research, the development of a verification method for Android applications is carried out by utilizing Dex CRC and the Blake2 algorithm, which will be written in C using the Java Native Interface (JNI). The results of this study indicate that the verification method using Dex CRC and the Blake2 algorithm can effectively protect Android applications from Application Repackaging attacks without burdening application performance.

Keywords— reverse engineering, application repackaging, blake2, Android protection

1 Introduction

Android is a Linux kernel-based operating system developed by Google which is widely used on smartphones and tablets today. Android smartphone sales are predicted to take around 68% of total smartphone sales. This causes the development of Android applications very rapidly [1]. App repackaging is a reverse engineering attack technique that is used to modify or insert various kinds of code into applications. In application development, there are always hackers who try to exploit / attack applications developed in the form of reverse engineering, including Application Repackaging, which is a common and severe threat in the world of Android application development. Hackers can use reverse engineering tools to disassemble an app and change, insert, modify the source code or make fake purchases [2].

Attackers can abuse the app repackaging to commit crimes such as modifying, pirating or inserting malware and then share the application through third-party Market applications or websites [3]. According to [4], 80% of malware samples are implemented via app repackaging. Code obfuscation, stub Dex, VMP, and MD5 Signature verification is commonly used to protect applications from reverse engineering attacks, where the code is difficult for attackers to understand the smali code and they can be bypassed by using some debugging tools, such as DexDump, ARM Pro, NP Manager, Ultima (used for analyzing and extracting Android applications and getting the original code

and then repacking the app again) but consume a lot of time to hack. All anti reverse methods mentioned above have vulnerabilities including :

- Ad Removal / addition or modification.
  In some applications such as games, there are advertisements that are sometimes annoying so that users try to disassemble the application and then delete/add/modify the existing ad providers in AndroidManifest.xml.
- Cloning.
  When hackers want to duplicate the same application on a Smartphone, this can be done by changing the application package.
- Cheating game.
  Game cheats can be inserted into the application with Reverse Engineering.

To address these vulnerabilities and enhance security measures, we propose a novel signature verification technique for applications. It involves calculating the CRC of the Dex file and encrypting it using the Blake2 algorithm to generate a hash signature, which is then used for integrity checks. Even the slightest change in the application will result in a different hash value. Our signature method is independent of the default Android signature, making it difficult for third-party tools to detect or manipulate the signature value. As a result, any repackaged application can be detected and appropriate actions can be taken.

2 Related Work

There are many studies about the method of preventing reverse engineering in Android applications. [5] discussed obfuscation techniques to deceive and delay hacker time to reverse engineer, [6] also introduced obfuscation techniques by adding useless code and encrypting strings on dex, then [7] discussed an advanced technique, namely control flow obfuscation where the obfuscation process is made more complicated and more effective, [8] combined obfuscation and native code to make the code more difficult to reverse, in the same year also [9][10][11][12][13] improves the obfuscation technique by using similarity analysis to detect repackaged apps. [14] used an obfuscation technique in the Kotlin programming language which is a new language in Android application development, and in 2019, [15] used an obfuscated logic bomb which will be triggered when the application has been modified.

Another technique is Stub Dex which was discussed by [16], this technique moves classes.dex to another place in the APK then makes Stub Dex the first to be called when running the application and dynamically loads resources/classes.dex which will run. He also add a rooted/debugging environment and evasion attack.

Furthermore, there is a virtualization technique introduced by [17][18][19] where this technique secures the code by extracting the ARM instruction key and then mapping the instruction into virtual instructions which are then encoded into the SO file. [20] added mapped key protection to the virtualization process to make it difficult to restore so that the code is very difficult to crack. [2] implemented this virtualization

method at the binary level making it more difficult to crack and extending the hacking time.

In addition, [Parvez Faruki, 1] used the Robust Feature Signature technique where this technique detects malware or applications that have been repackaged using a database of around 1260 application samples and studies the META.INF and classes.dex of each application then calculate the value of its similarity. Furthermore, [21] used the tree structure of the AndroidManifest.xml file to detect cloned Android applications.

Author in [22] compared the original signature of existing applications on the Android market with the signatures of third-party applications and then calculated the similarity values of the two signatures.

3 Proposed Method

Default Android signature verification using MD5 can be easily obtained from third-party tools and then entered into the hook function which makes the application signature appear as if it has not changed, an example of the default Android signature hook is as shown in Figure 1.

```
static void c(Context context) {
    try {
        //MD5 DATA
        String data = "5F745C3E85992E6A87B38A5EE429A62C";
        DataInputStream is = new DataInputStream(data);
        final byte[][] originalSigns = new byte[is.read() & 0xFF][];
        for (int i = 0; i < originalSigns.length; i++) {
            originalSigns[i] = new byte[is.readInt()];
            is.readFully(originalSigns[i]);
        }

        Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
        Method currentActivityThreadMethod = activityThreadClass.getDeclaredMethod("currentActivityThread");
        Object currentActivityThread = currentActivityThreadMethod.invoke(null);

        Field sPackageManagerField = activityThreadClass.getDeclaredField("sPackageManager");
```

Fig. 1. Code snippet of hooking get signature function to bypass signature verification

Figure 1 shows that the default Android signature that has been obtained using third-party tools will be saved on the hook source code

```
String data = "5F745C3E85992E6A87B38A5EE329A62C"
```

and then force getSignature function to return the original signature every time the application is opened using the hook function, so that it appears that the application has not changed. An overview of the MD5 verification bypass using hooking method (Figure 1) can be seen in Figure 2.

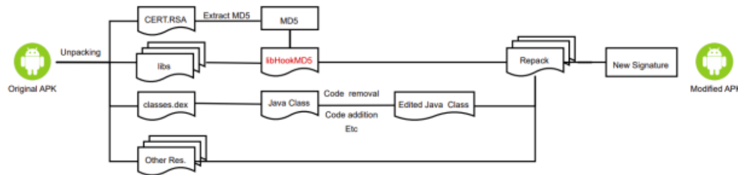

Fig. 2. MD5 verification bypass flow using hook

Hackers use third-party apps or manually bypass signature hook for example "libHookMD5". This hook will manipulate the getSignature function to always return the original md5 signature.

Therefore, we proposed a novel signature verification technique using Dex CRC and Blake2 Algorithm. This technique is written in C using the Java Native Interface (JNI) so that the source code is better preserved from decompilation and can run the blake2 algorithm effectively, this technique works by taking the CRC from class.dex and then encrypting it using the Blake2 algorithm to get a secure signature hash, then this hash will be verified every time the application run. If the application undergoes the slightest change, such as changing the application name, changing the package, changing the string, editing the XML layout, or editing the class.dex, the hash signature will change and will still be detectable even though a signature verification bypass has been carried out from several third-party tools. The Dex CRC signature verification architecture can be seen in Figure 3.

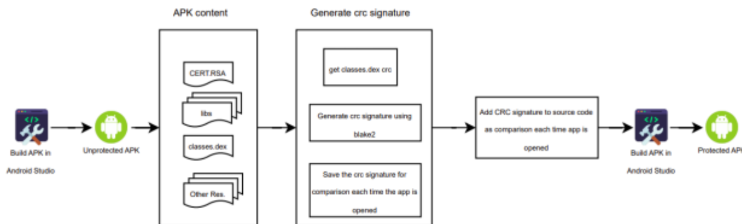

Fig. 3. Dex CRC and Blake2 Signature verification architecture

The developed signature verification method can be resistant to signature bypass attacks because it does not use the default Android signature using MD5. Unlike the method we propose, the signature is obtained from the application classes.dex and then encrypted using Blake2 with a secret key so that the hash signature cannot be obtained by even third-party tools. Hackers is impossible or very difficult to get Dex CRC signature to be hooked, see Figure 4.

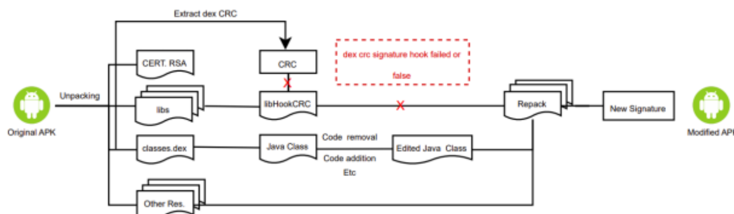

Fig. 4. Dex CRC and Blake2 signature verification cannot hooked / bypassed.

Pseudocode 1 : Dump / Log the original signature

```
1  dumpOrigHash()
2      crc_orig_hash = null
3      Crc = getCRC()
4      crc_hash = blake2(crc, “key”)
5      log(crc_hash)
6  end
```

First, we run the application and get generated Dex CRC and Blake2 signature using Android log and we save it into the source code as the original signature

Pseudocode 2 : Check Signature function

```
1  CheckSignature()
2      crc_orig_hash = “ABCD” // this is the hash obtained in step 1
3      crc = getCRC()
4      crc_hash = blake2(crc, “key”)
5      if (crc_hash != crc_orig_hash)
6          //do some stuff
7          Exit()
8      endif
9  end
```

After the original signature is obtained and added to the source code, now it will run every time the app opened and compare the current running Android application signature with the saved original signature to make sure that the application is repackaged or not, if the application is detected as repackaged then the application will force close.

4 Performance Evaluation

We have evaluated the performance of our proposed signature verification based on Dex CRC and Blake2 algorithm to mitigate reverse engineering attacks on applications. The evaluation was carried out by performing some modification/repackaging on Android applications and performing signature bypass using third-party tools to test the

robustness of the proposed signature verification. In addition, we also evaluate application performance at startup by comparing CPU, Memory usage, and load time in the original application and the application that has been added proposed method. To carry out this evaluation we use a private application because the code must be added to the source code so that the evaluation is carried out on the application we have made.

In this evaluation, we used the Xiaomi 11T Pro smartphone device with Android version 13. We used Android Studio to evaluate RAM and CPU and used the MT Manager and NP Manager tools to bypass signature verification where these two tools in total have 4 of the most frequent signature bypass methods. used by hackers.

Table 1. Smartphone specification

Brand / Type	Xiaomi 11T Pro
Android Version	13
RAM	12 GB
CPU	Snapdragon 888

As can be seen in Table 2, we made several modifications to the application and the Dex CRC and Blake2 signature has changed which indicates that the application is no longer original or has changes

Table 2. Application modification attack test

No	Attack type	Original signature	Signature after modifying
1	Changing Application Name	D260C66BDD6860658 25251732B1A...	964A182212F156288 12925AC8B…
2	Changing some xml	D260C66BDD6860658 25251732B1A...	964A182212F156288 12925AC8B…
3	Modify classes.dex	D260C66BDD6860658 25251732B1A...	D1ED8A27415CAF87 67F23D7D…

```
601 .method private startFloater()V
602     .registers 3
603
604     .line 538
605     invoke-direct {p0}, Lcom/kzm/e
606
607     move-result v0
608
609     if-nez v0, :cond_11
610
611     .line 539
612     new-instance v0, Landroid/con
```

(a) original smali code

```
601 .method private startFloater()V
602     .registers 3
603
604     .line 538
605     invoke-direct {p0}, Lcom/kzm/e
606
607     move-result v0
608     const/4 v1, 0x1
609     if-ne v0, v1, :cond_11
610
611     .line 539
612     new-instance v0, Landroid/con
```

(b) modified smali code

Fig. 5. Smali code

```
private void startFloater() {
    if (!isServiceRunning()) {
        startService(new Intent(
    } else {
```

(a) original java code

```
private void startFloater() {
    if (isServiceRunning()) {
        startService(new Inter
    } else {
```

(b) java code after smali modification

Fig. 6. Java code

Figure 5 (a) and (b) shows the original smali code and modified smali code ,respectively hacker modified the conditional statement as shown in Figure 5 (b). Figure 6 (a) and (b) shows the original java code and java code after smali modification.

```
/com.kzm.esp I/MRX: DEX CRC with Blake2 signature invalid
/com.kzm.esp I/MRX: original signature D260C66BDD686065825251732B1AFD56121CAABC2E1
/com.kzm.esp I/MRX: current signature 964A182212F15628812925AC8BDA79C0A287998C4737
```

```
com.kzm.esp I/MRX: DEX CRC with Blake2 signature invalid
com.kzm.esp I/MRX: original signature D260C66BDD686065825251732B1AFD56121CAABC2E
com.kzm.esp I/MRX: current signature D1ED8A27415CAF8767F23D7D4446F5EA4723F25CC9E
```

Fig. 7. Signature verification result after changing application name, xml and modifying dex file

Figure 7 shows our method detects and displays a log of signature verification results after changing the app name, XML layout, and modifying dex. In Table 3, the signature verification bypass test was carried out using four signature kill/bypass tools including NP Kill Sign., NP Kill Sign. V2, SF Kill Sign., and Modex 3 Kill Sign., where we can see the results that none of the four bypass signatures can manipulate the proposed CRC signature so that the application can still be detected as an application that is not original or has been modified/repackaged.

Table 3. Signature verification kill / bypass test

No	Attack type	Original signature	Signature after modifying
1	NP Sign Killer	D260C66BDD6860658 25251732B1A...	DF2825FDC4A03EB4 5E3F11B9 …
2	SF Sign Killer	D260C66BDD6860658 25251732B1A...	D7A94A1F436F46AE 3D3159AA…
3	Modex 3 Sign Killer	D260C66BDD6860658 25251732B1A...	6C36AAA597EE268C 49D57CE4…
4	NP Sign Killer v2	D260C66BDD6860658 25251732B1A…	FAB93CDB2653643C E4F74E5D…

```
/com.kzm.esp I/MRX: current signature DF2825FDC4A03EB45E3F11B9E4DCA2EBBF629FE95B
/com.kzm.esp I/MRX: current signature D7A94A1F436F46AE3D3159AAA70989A95063
/com.kzm.esp I/MRX: current signature 6C36AAA597EE268C49D57CE4D218C8D94BC6D7072B
/com.kzm.esp I/MRX: current signature FAB93CDB2653643CE4F74E5D948019E1AD7
```

Fig. 8. Signature verification result for signature killer in Table 3.

In the signature bypass test, certain logs were deleted by the signature kill tools, but the current signature verification based on Dex CRC and Blake2 remains intact and cannot be bypassed. Furthermore, Table 4 demonstrates that our proposed method has no significant impact on CPU usage, as the highest CPU usage observed in applications using MD5 verification is 29%, while applications using Dex CRC and Blake2 signature verification show a slightly lower CPU usage of 27%. The memory usage of the application where the highest RAM usage is 173 MB for applications using MD5 signature verification and 184 MB for applications using Dex CRC and Blake2 signature verification and the average difference between the two methods in RAM usage is only 3.2 MB. In terms of loading time, MD5 signature and our method (combination of Dex CRC and Blake2) takes 5 seconds and 5.57 seconds to open MainActivity of the application, respectively.

Table 4. Performance test table

Attempt	Memory Usage (MB)		CPU Usage (%)		Loading Time (Second)	
	Dex CRC	MD5	Dex CRC	MD5	Dex CRC	MD5
1	168,00	160,00	26	26	4,96	4,71
2	162,00	166,70	26	26	4,54	4,47
3	166,50	161,20	26	25	4,55	4,55
4	162,90	164,30	25	25	4,83	4,84
5	163,00	160,00	25	25	5,00	4,49
6	170,50	162,00	27	26	4,68	4,37
7	167,00	164,00	26	29	4,38	4,25
8	184,20	173,00	18	17	5,57	5,40
9	167,00	162,00	26	24	4,61	5,41
10	168,00	167,40	25	26	5,11	5,19
11	166,30	159,00	25	26	5,00	4,48
12	164,20	159,70	25	25	4,92	4,93
13	162,00	167,00	26	25	4,68	4,75
14	169,50	171,20	26	25	4,67	5,21
15	170,00	165,40	25	26	5,06	5,06
16	170,10	166,50	26	25	5,08	5,08
17	168,60	166,90	26	26	5,00	4,51
18	160,00	161,40	26	24	5,00	4,56
19	170,80	161,20	26	26	4,72	5,07
20	163,70	161,00	25	26	5,54	4,79
Max	**184,20**	**173,00**	**27**	**29**	**5,57**	**5,41**
Min	**160,00**	**159,00**	**18**	**17**	**4,38**	**4,25**
Average	**167,22**	**164,00**	**25**	**25**	**4,89**	**4,81**

We also compare the other method with our proposed method in Table 5.

Table 5. Method comparison

Type of Attack	Method					
	Code Obfusca-tion	**Stub Dex**	**Virtu-ali-zation**	**Ro-bust**	**Default Signature Verifica-tion**	**Propo-sed Me-thod**
Decompile App	Yes	Yes	Yes	Yes	Yes	Yes
Directly edit decom-piled app	Yes	No	No	Yes	Yes	Yes
Recompile / repackage app	Yes	Yes	Yes	Yes	Yes	Yes
Recompiled app can run-ning wit-hout signa-ture verifi-cation by-pass	Yes	Yes	Yes	Yes	No	No
Recompiled app can run-ning after signature verification bypass	Yes	Yes	Yes	Yes	Yes	No

The evaluation shows that reversed / repackaging application using Dex CRC and Blake2 algorithm cannot be run even though hackers add the signature verification by-pass as shown in Figure 4. Our proposed method has a minimal impact on application performance because of small changes to the app, unlike the obfuscate and virtualiza-tion methods requiring many changes to the code, hence, the size of the application becomes large which will affect the performance.

5 Conclusions & Future Directions

This paper presents a novel approach called Signature Verification Based on Dex CRC and Blake2 algorithm for ensuring the integrity of Android applications. By leveraging Dex CRC and Blake2, this method enhances the security of applications against reverse engineering attacks where the repackaged application cannot run even if it has a signature verification bypass. To the best of our knowledge, Dex CRC and the Blake2 algorithm is the first technique employed to bolster the resilience of Android applications, addressing the vulnerabilities found in existing methods. The performance evaluation demonstrates that the proposed method effectively mitigates signature verification bypass techniques commonly used by attackers. It provides robust protection even against minor modifications, such as changes to application names or packages. Furthermore, the performance evaluation indicates that the use of Dex CRC verification has a minimal impact on application performance.

Our proposed method focused on preventing an application from running after being repackaged/reversed so that the changes made by hackers will be useless, however, hackers can still read the information contained in decompiled applications, especially in the java classes, to cover this issue, a combination of several methods is required so that the anti-reverse method can be improved.

draft paper ilham

ORIGINALITY REPORT

0%	0%	0%	0%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Changsung Lee, Jaewook Jung, Jong-Moon Chung. "Intelligent Dual Active Protocol Stack Handover Based on Double DQN Deep Reinforcement Learning for 5G mmWave Networks", IEEE Transactions on Vehicular Technology, 2022 Publication	<1%

Exclude quotes	On	Exclude matches	< 5 words
Exclude bibliography	On		