ACIDroid: A Practical App Cache Integrity Protection System on Android Runtime

***Abstract*—To improve the execution performance of applications, Android introduced a new optimization technique using app cache files. However, this new feature also brings a new security concern called "app cache tampering attack" that can eventually change the behavior of installed applications by modifying the executable codes in their app cache files. We address this issue with ACIDroid, an efficient app cache integrity protection solution on Android, which relies on the selective transformation of the original DEX codes into the optimized DEX codes.**

To show the feasibility of ACIDroid, we performed app cache tampering attacks on 14 popular Android apps and tried to detect the changes in app cache files using ACIDroid. With the modified app cache files, ACIDroid was able to correctly detect the (intentional) changes while having an acceptable execution time overhead less than 5% of the total execution time.

I. Introduction

Since Android 4.4 (KitKat), Google introduced an optimization technique that uses app cache—when applications are installed, their executable codes are compiled into *optimized* executable codes and stored in app cache files to improve the performance of the installed apps (e.g., speeding up the execution) [1]. However, this optimization technique also raises a security concern about the integrity of app cache files [2].

In practice, some sensitive Android apps (e.g., banking and game apps) have a self-defense mechanism to check the integrity of apps by themselves [3]—such apps typically verify the validity of its Android application package (APK) file with its signature while there is no sufficient security mechanism to protect the integrity of app cache files. Therefore, sophisticated attackers would devise a new attack to modify app cache files instead of the original APK file. Even though the Android framework uses a checksum-based integrity check to protect app cache files, we found that such checks can effectively be bypassed via the modification of checksum of the target cache file. We call this type of attack *app cache tampering attack* because executable codes in app cache files can maliciously be modified to change the program's control flow and data.

Recently, Wan et al. [2] suggested a defense mechanism with the precomputed hash values of all possible optimized Dalvik EXecutable (DEX) codes in order to check the integrity of the optimized DEX codes that would be appeared in the app cache files. However, it is quite challenging to precompute all possible hash values for optimized DEX codes and efficiently store them in the app because machine-specific DEX codes in the app cache files may vary with Android OS version and `compile-filter` option.

To provide the integrity of the optimized DEX codes in app cache files with minimum storage burden, we present "ACIDroid" (**A**pp **C**ache **I**ntegrity on An**droid**), which is a highly effective cache integrity protection system that relies on the selective transformation of the original DEX codes into the optimized DEX codes. We found that there are general rules for converting original DEX codes into the optimized DEX codes in app cache files even with variation in Android OS version and `compile-filter` option. ACIDroid applies those rules to detect the modification of the optimized DEX codes (that we are particularly interested in) in app cache files.

Our key contributions are summarized below:

- Design of a practical app cache integrity protection system on Android runtime, which uses 18 converting rules to optimized DEX codes, demonstrating that ACIDroid requires significantly less storage overhead compared with Wan et al.'s approach [2] using the database of app cache file signatures.
- Evaluation of ACIDroid's performance against app cache tampering attacks on 14 real-world Android apps, demonstrating that ACIDroid can accurately check the integrity of optimized DEX files in app cache files while having an acceptable execution time overhead less than 5% of the total execution time.

The rest of this paper is organized as follows. In Section II, we provide some background information about Android runtime (ART) execution environment and app cache files. Section III describes how app cache tampering attacks are implemented, and Section IV presents the overview of ACIDroid against app cache tampering attacks. Section V presents the evaluation results. We discuss how ACIDroid can be deployed in real-world apps and its limitations in Section VI. Our conclusions and further work are in Section VII.

II. Background

A. Android runtime (ART)

Android is an open-source project for mobile-phone operating systems. While Android is a Linux-based system, the applications for Android are usually implemented using Java and compiled to specific bytecode known as Dalvik EXecutable (DEX). During runtime, an application can be executed in either Dalvik or ART virtual machine environment depending on Android OS version.

Dalvik Environment Dalvik virtual machine is a concept to execute the application on Android versions later than 2.2. The

Workshop on Binary Analysis Research (BAR) 2019
24 February 2019, San Diego, CA, USA
ISBN 1-891562-58-4
https://dx.doi.org/10.14722/bar.2019.23xxx
www.ndss-symposium.org

Dalvik Just In Time (JIT) compiler dynamically translates parts of the Dalvik bytecode into machine code known as native code each time the app is run [4].

Some parts, which are not executed, are not compiled at all. Therefore, JIT compilation method has advantages of using less memory and shortening compilation time [5].

ART Environment ART virtual machine is an application runtime environment used on Android versions later than 4.4 (known as KitKat). It replaced Dalvik and also introduced AOT (Ahead Of Time) compiler which compiles the entire application's bytecode into machine code at the time of installation [6]. In other words, the entire DEX bytecode is compiled into executable machine code. In this way, ART improves overall runtime efficiency. The code compiled by AOT compiler in ART is optimized and stored in the cache files to improve performance. After compiling an application, the files generated by AOT compiler have the extensions of `.art`, `.oat`, `.odex` and `.vdex`. Moreover, AOT compiler uses the dex2oat tool to convert the original DEX stored in the APK file to optimized DEX which is stored in cache files in turn. Furthermore, `.odex` and `.vdex` include optimized DEX generated from original DEX in APK. The files which are modified to cache files after this optimization process are known as `.odex` and `.vdex`. These files can be directly executed to improve performance when the application is run.

ART also uses checksum to determine whether the cache files have been tampered. If it concludes that the cache files have not been tampered, ART directly executes the cache files which include optimized DEX. On the other hand, if ART determines that any of the cache files has been modified, the ART calls DEX in the original APK file. AOT compilation has advantages of being able to execute with high performance, shortening startup time and expanding battery lifespan [7]. However, it has still been a problem that the compilation time could be burdened and the internal storage could be lacked [8].

The improvement of hardware specification made it possible to combine JIT compilation system with AOT compilation system (Dalvik and ART). Recent smart-phone has large storage, and as a result, the ART system does not give the burden to device anymore. The burden of the storage space is eased thanks to the improvement of hardware specification. The AOT compilation method has benefits of improving startup and runtime performance.

B. App cache files

After installing an application, a variety of files which have the file extensions `.art`, `.oat`, and `.odex` are generated. The cache file having `.vdex` extension is also generated on Android versions later than 8. The cache files which end with `.odex` and `.vdex` have an OAT file format. If the cache file has not been tampered with, the ART will execute the application's cache file directly. The cache file which is directly run varies depending on the version of Android OS. OAT files are generated by a **dex2oat** or a **dexopt** tool which are included in Android OS. The **dex2oat** tool keeps the compatibility [9], and the **dexopt** tool optimizes the original DEX bytecode by pre-computing data, pruning empty methods and improving virtual method calls [10].

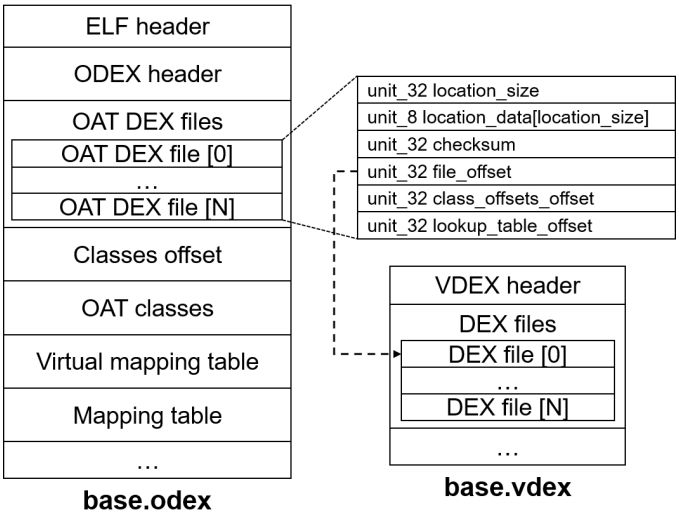

Fig. 1. Example of OAT cache file structure on Android 8.

Among the OAT files, the files that end with `.odex` are called cache files which are special Executable and Linkable Format (ELF) files. Such cache files are depicted in Fig. 1. In addition, the `.odex` file is sandboxed and securely protected in the application folder (the application's data folder[1]) [2]. When the application is installed, Android checks if all of the security parameters (e.g., checksum and code signing) are valid. However, after the application's installation, when the user tries to run the application, ART only checks the checksum value of `.odex` file's header. ART compares the checksum of `.odex` file against the checksum value of `base.art` file or `base.apk` file. Therefore, if `.odex` file is changed in a malicious manner, ART cannot prevent the modified `.odex` file from being run [2].

III. App cache tampering attack

Sabanal [11] introduced the app cache tampering attack on Android ART runtime environment. To implement this attack, an attacker needs to have a system privilege to disable the attacked application sandbox. While this assumption may seem rather strong, it is not uncommon in practice and should be considered for app protection. There exist several survey results showing that a large number of Android devices were rooted by device owners for the purpose of getting rid of unnecessary built-in apps or updating to the latest version of Android. According to the official report from Google in 2016 [12], about 5.6% of all Android users were using rooted devices–either intentionally or due to security bugs. Furthermore, the survey results in [13] showed that about 7% of all respondents were using rooted Android devices.

To implement app cache tampering attacks for experiments, we modified the OAT files complied from the base APK file by the AOT compiler. Fig. 2 shows an overview of the app cache tampering attack on ART. The app cache tampering attack can be launched in the following steps:

1) We install the target original APK files on the device and automatically generate the OAT cache files using

[1]/data/app/[APK package]-[random hash]/oat/[instruction sets]/

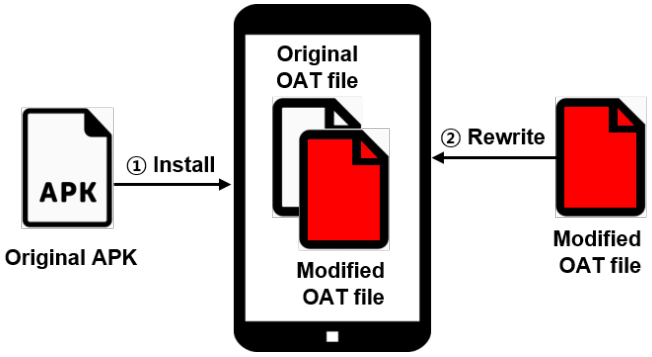

Fig. 2. Overview of app cache tampering attack on ART.

```
const-string v6, “Tampered”
const/4 v7, 0x0
invoke-static {p0, v6, v7}, Landroid/widget/Toast;->
        makeText (Landroid/content/Context;Ljava/lang/
        CharSequence;I) Landroid/widget/Toast;
move-result-object v6
invoke-virtual {v6}, Landroid/widget/Toast;->show()V
```

Fig. 3. Example of the injected smali code.

the **dex2oat** and **dexopt** programs. In this step, if Android version is lower than or equal to 7, the OAT cache files are generated as `base.odex` file. If Android version is equal to 8, the OAT cache files are generated as `base.odex` and `base.vdex` files.

2) We extract the generated OAT files from the device. Since this step is solely for making modification of the cache file easy, it is not necessary to extract the OAT files. If the OAT files can be tampered directly, this step can be skipped.
3) We analyze the extracted OAT cache files, modify the control flow of application and inject some malicious codes we want to be executed. Further, the modified OAT files can be generated using an easy method since we can generate the modified OAT files from the modified APK file. For instance, we can decompile the original APK file using the open source decompilation tool such as apktool [14], modify the existing smali code and inject malicious smali code such as the one shown in Fig. 3. We can then obtain the modified OAT file from the device which will be installed in the modified APK file. This is the easy method which can be used to obtain the OAT files that modify the control flow of the application.
4) We patch the checksum field of the modified OAT file with the checksum of the original APK file. Note that the checksum value is related to the original APK file, not to the OAT file. If the checksum of the OAT file is not equal to that of the original APK file, the execution request is ignored. In addition, ART will re-compile the original APK file and will overwrite the existing OAT file.
5) We insert the OAT file into the target device. We confirmed that the application successfully operated under the flow we induced, not the original.

TABLE I. LIST OF THE APPLICATIONS USED IN OUR EXPERIMENTS.

Category	Application name
Finance	Paypal
	Bank of America
Productivity	Outlook
	1Password
	Dropbox
	Skydrive
Business	Azure Authenticator
	Blizzard Authenticator
Medical	TexasHealthMyChart
Tools	Google Authenticator
Communication	Facebook messenger
Social	Pinterest
Music & Audio	Amazon Alexa
Travel & Local	Booking

We collected Android applications which can be abused to steal personal information and operated maliciously. Table I shows the collected 14 applications from 9 different categories. These applications were modified using the process mentioned above.

IV. ACIDROID

In this Section, we describe the design and implementation of ACIDroid, our proposed verification system. In Section IV-A, we introduce the process of the verification system. In Section IV-B, we explain the converting rules for optimizing instruction and illustrate the experiments and proof for constructing the converting rules.

A. Design of ACIDroid

We suggest a new method to verify the integrity of OAT cache files on Android ART framework as shown in Fig. 4. We assume that the integrity of the APK file is guaranteed because the APK file can be protected by several methods [3], [15]–[17]. The verification processes are as follows:

1) ACIDroid extracts DEX file A from `base.apk` and DEX file A' from OAT files, which contain the optimized DEX. The optimized DEX A' is included in `base.odex` file on Android versions 5 [18] to 7, and it is included in `base.vdex` on Android versions higher than 8.
2) ACIDroid compares the sizes of DEX files (A and A') extracted from `base.apk` and OAT files. If the sizes are not the same, the execution request is denied.
3) If the sizes are the same, ACIDroid compares the names of the classes and methods. If any of the names is different, the execution request is denied.
4) If all the names are the same, ACIDroid identifies mismatched codes. ACIDroid converts the mismatched DEX codes in A to the optimized DEX codes using converting rules, and compares them with codes in the optimized DEX file A' from `base.odex`. At this step, we provide two options (full inspection and partial inspection) for developer. In full inspection, the whole DEX files are compared. This kind of inspection has a 100% detection rate and a 0% error rate theoretically and also requires a lot of

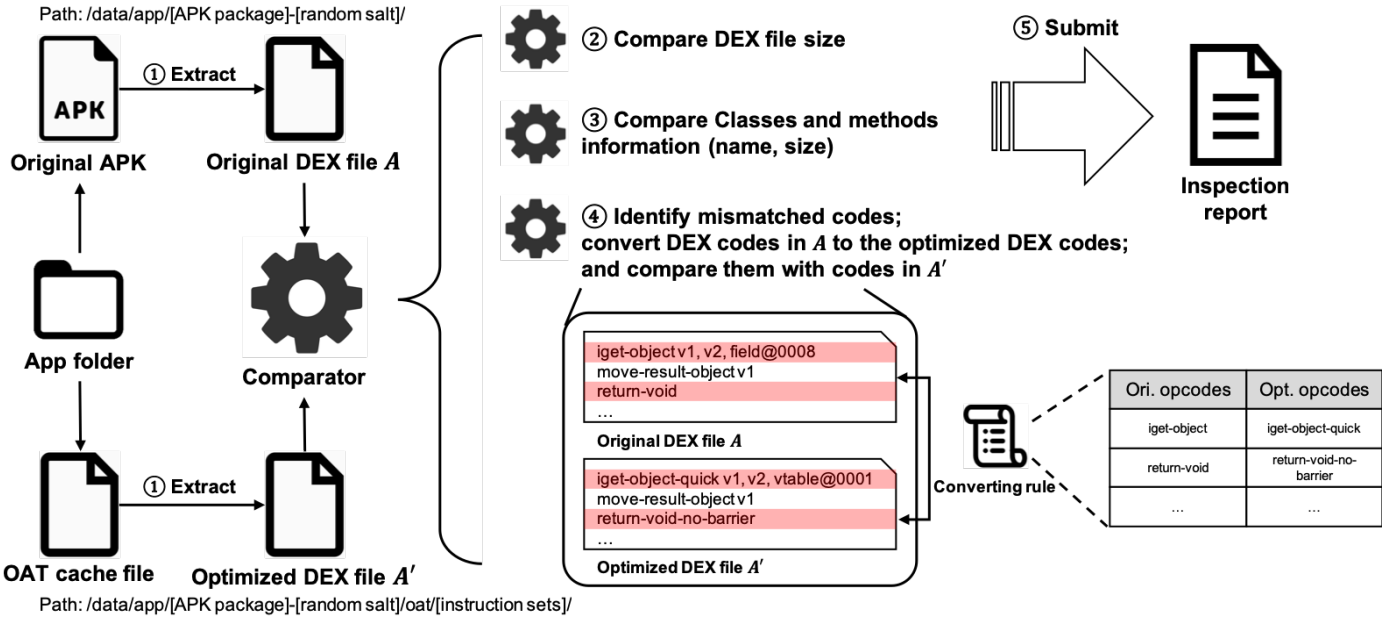

Fig. 4. High-level design of ACIDroid, showing the key stages involved in app cache tampering attack detection.

time and power of the device. In partial inspection, only some portion of DEX files such as specific classes or methods are compared. As a result, when this kind of inspection is used, modified DEX might possibly be undetected. However, partial inspection can still detect modified classes or methods and also protect the parts the user wants effectively.

5) ACIDroid reports the detailed inspection. The inspection report includes the result whether the app cache files have been tampered or not and also the position specifying where in the app cache files has been modified.

We implemented ACIDroid in native code using LIEF library[2]. ACIDroid extracts DEX files from the OAT cache files and compares them with the DEX files extracted from the original APK file. Bytecode comparisons are performed for each method via `lief.DEX.Method.bytecode` provided by the LIEF library. If some bytecodes of the OAT cache files are mismatched with the bytecodes of the original APK file, ACIDroid uses the converting rules to check if the mismatch occurred by the effect of optimization. If the mismatch was not caused by the optimization, ACIDroid decides the mismatch has been caused by the app cache tampering attack.

B. Instruction converting rules

To improve the performance such as speed, space of an application and so on, several optimization methods have been considered. Among a variety of improvement methods, for the speed optimization, ART adopts the method that changes the instruction of DEX files to the optimized instruction. The method does not change all instructions but rather optimizes only some of the instructions. The optimizable instruction list is different depending on the Android version, and this is depicted in Table II.

[2]https://github.com/lief-project/LIEF

DEX code that can be optimized varies depending on the Android version and the `compile-filter` of **dex2oat** tool. For example, on Android 7, there are 12 `compiler-filter` options. If `interpret-only` option is enabled, ART optimizes DEX files only if the interpreter performance would be improved, and the `speed` option does AOT compilation for all DEX files to increase application performance especially speed on Android 7. In addition, **dex2oat** tool provides 10 compiler filters which include four officially provided options [19] and additional options provided on Android 8.

The converting rules can be obtained from the official website [20] or the source code of AOSP [21]. The converting rules according to the Android version are shown in Table II. We performed experiments to verify the consistency with the converting rules identified from the source codes of AOSP. We compared the DEX files of the original APK file with the DEX files of the OAT cache files according to Android versions (5 to 8) using 14 applications which are listed in Table I. As a result, we could confirm that the optimized opcodes of Android 5 were different from the optimized opcodes of the other Android versions. This could also be confirmed from the source code of AOSP.

V. Evaluation

A. Experiment setup

To test the verification system we proposed, we collected 14 applications from 9 categories listed in Table I. The applications are sensitive to the modification of the control flow, and they have the integrity verification system themselves. However, the system only verifies the integrity of the APK file. As shown in Section III, to confirm the app cache tampering attack, we tampered the cache files of 14 applications and confirmed that the modified cache files were working.

To check the device dependency and evaluate the performance of ACIDroid, we used 4 physical devices (Pixel 2,

TABLE II. CONVERTING RULES FOR GENERATING OPCODES IN OPTIMIZED DEX FILES. DIFFERENT RULES CAN BE APPLIED FOR EACH ANDROID VERSION.

Opcode (hex)	Opcode name	Android 5		Android 6, 7, 8	
		Optimized opcode	Optimized opcode name	Optimized opcode	Optimized opcode name
0E	return-void	73	return-void-no-barrier	73	return-void-no-barrier
1F	check-cast	00	nop	00	nop
52	iget	E3	iget-quick	E3	iget-quick
53	iget-wide	E4	iget-wide-quick	E4	iget-wide-quick
54	iget-object	E5	iget-object-quick	E5	iget-object-quick
59	iput	E6	iput-quick	E6	iput-quick
5C	iput-boolean			EB	iput-boolean-quick
5D	iput-byte			EC	iput-byte-quick
5E	iput-char			ED	iput-char-quick
5F	iput-short			EE	iput-short-quick
5A	iput-wide	E7	iput-wide-quick	E7	iput-wide-quick
5B	iput-object	E8	iput-object-quick	E8	iput-object-quick
6E	invoke-virtual	E9	invoke-virtual-quick	E9	invoke-virtual-quick
74	invoke-virtual/range	EA	invoke-virtual-quick/range	EA	invoke-virtual-quick/range
55	iget-boolean	-	-	EF	iget-boolean-quick
56	iget-byte	-	-	F0	iget-byte-quick
57	iget-char	-	-	F1	iget-char-quick
58	iget-short	-	-	F2	iget-short-quick

Galaxy S9, Nexus 5x and Galaxy S5) and 8 images which were publicly use-able images (called AOSP). We tested 12 environments on 3 different Android versions. That is, we tested 2 devices (Pixel 2 and Galaxy S9) and 1 AOSP image (x86_32) on Android 8. We tested 1 device (Nexus 5x) and 2 AOSP images (x86_32, x86_64) on Android 7, 1 device (Galaxy S5) and 2 AOSP images (x86_32, x86_64) on Android 6, and 3 AOSP images (x86_32 and x86_64 on Android 5.1 and x86_32 on Android 5.0). All AOSP images were installed and executed on the computer which had Intel core i7, 16 GB memory and GeForce GTX 1080 graphics card.

B. Experiment results

We evaluate our integrity verification system on app cache files (ACIDroid) in terms of overhead time ratio which can affect the battery consumption and the application performance, and in terms of effectiveness how exactly the system detects the modified classes or methods. We evaluated 3 categories: time overhead, storage overhead and detection ratio.

Time overhead To measure the time overhead ratio, we measure the time T_{verify} the system spends on verification and the time T_{ready} it takes for the application to reach the ready state from the start state. ACIDroid is developed in native code, so we can measure the exact time by calculating the time difference between the start point of code and the end point of code, and printing out through the logcat.

For comparing the overhead time when ACIDroid is working against the application execution time when ACIDroid is not working, we developed an application which executes the target application. We assumed that the start time of the target application is the time printed before ACIDroid tries to execute the target application. On the other hand, the end time of the target application was estimated by using the logs of the logcat. We collected and analyzed logs from various different applications. As a result, we could figure out some common features existed in all the recorded logs, i.e, the common features existed in the recorded logs of every application. We conjecture that the repeated common features are the system logs to help maintain the signal that the application receives from the user. Overall, the overhead time can be calculated as shown in Equation (1).

TABLE III. EXECUTION TIME OVERHEAD OF ACIDROID IN APPLICATIONS FOR EACH ANDROID VERSION.

App name	# of DEX	Android 8	Android 7	Android 6	Android 5
Paypal	2	1.45 %	3.54 %	5.74 %	3.03 %
Bank of America	2	1.15 %	3.18 %	5.12 %	3.14 %
Outlook	8	1.09 %	2.58 %	4.25 %	4.16 %
1Password	1	0.88 %	2.25 %	3.18 %	3.33 %
Dropbox	1	0.76 %	2.18 %	3.35 %	2.76 %
Skydrive	3	1.05 %	3.66 %	7.78 %	-
Azure Authenticator	9	1.07 %	2.60 %	3.72 %	0.82 %
Blizzard Authenticator	2	0.85 %	1.87 %	2.82 %	2.76 %
Google Authenticator	1	0.44 %	1.20 %	1.24 %	0.71 %
TexasHealthMyChart	2	0.76 %	1.78 %	2.38 %	2.11 %
Facebook messenger	5	0.19 %	2.77 %	12.06 %	-
Pinterest	3	0.90 %	2.53 %	6.65 %	-
Amazon Alexa	3	0.83 %	2.03 %	3.37 %	2.76 %
Booking	4	0.84 %	2.07 %	3.99 %	2.79 %
Overall average (%)	3.29	0.88 %	2.45 %	4.69 %	2.58 %
Overall average (ms)	3.29	8.76 ms	24.45 ms	46.89 ms	25.81 ms

$$T_{overhead} = \frac{T_{verify}}{T_{verify} + T_{ready}} \tag{1}$$

To compare the time overhead of ACIDroid against the time overhead of Wan et al. [2]'s verification system, we implemented the integrity verification system using hash value in native code just like we implemented ACIDroid. The verification system using hash extracts the DEX files from the app cache files and hashes the DEX files using SHA256. The system compares the hashed value with the stored hash value when the app cache files are installed. The time overhead of the verification system using hash can also be calculated according to the Equation (1), and the time overhead of 12.67% was achieved when the verification system using hash was deployed on Android 8. On the other hand, ACIDroid achieved the time overhead of 0.88% as shown in Table III. This result shows that ACIDroid is faster than the verification system using hash by about 14 times.

To test how much the time overhead varies depending on which Android version ACIDroid is deployed, we measured

the time overhead individually according to the applications and the Android versions as shown in Table III. On Android 8, the time overhead incurred was less than that of other versions. This is because the devices which were used during the experiment on Android 8 environment were the most recent releases, and they have the best specification compared to other devices. In addition, Android 8 has more improved optimization. As shown in Table III, when older version of Android was used, it incurred a lot more time overhead compared to the case when Android 8 was used. This is because the devices used in Android 6 and 7 experiments have the lower specification compared to the devices used in Android 8 experiment. Unlike the cases when other Android versions were used, the performance has increased when Android 5 was used since the time overhead decreased to 2.58%. In this case, the experiment environments only involved AOSP images, not the physical devices. As a result, even though an initial version of ART framework is used, the time overhead incurred on Android 5 can be less than the time overhead incurred on Android 6.

Storage overhead Wan et al. [2] proposed a mechanism to protect app cache by pre-computing all possible optimized DEX hash values. However, their approach enforces the system to store all possible hash values of optimized DEX in secure storage. According to their proposed mechanism, the storage size required to store the hash values can be calculated as shown in Equation (2), Here, $B_{overhead}$ is the increase in size of bytes, N_{inst} is the number of `instruction-set` such as *Mips, Mips64, x86, x86_64, Arm, Arm64, Thumb2* which varies depending on the Android version, $N_{compile}$ is the number of `compile-filter` such as `assume-verified`, `everything`, `speed-profile` which varies depending on the Android version supported by `dex2oat`, and N_{DEX} is the number of DEX files in the application.

$$B_{overhead} = N_{inst} \times N_{compile} \times (1 + 4 \times N_{DEX}) \quad (2)$$

For example, suppose that the Booking application has eight DEX files, 10 `compile-filter options` exist in Android 8 and the mobile device has seven instruction sets. Then, according to the Equation (2), the storage overhead of Booking app comes out to be 2,310 bytes on Android 8. The storage overhead will be increased if the booking application has to be used on other versions of Android. ACIDroid, on the other hand, solves this storage overhead problem.

Detection ratio ACIDroid can decide the target methods, classes or DEX files. In the experiment we conducted, the inspected part was the first DEX file. That is, ACIDroid checked the modification of the first DEX file only and then decided whether the app cache files have been tampered or not. We tested ACIDroid with 14 tampered applications and 14 original applications. As a result, ACIDroid achieved 100% detection rate.

VI. Discussion

A. Possible deployments

ACIDroid can be incorporated into the existing APK integrity verification procedure. Fig. 5 shows a possible deployment of the app cache file integrity verification process using an external server [3]. To check the integrity of app cache files in an app, the app reports its own app information and the hash value of the optimized DEX to an external verification server. The server then obtains the original APK file with the received app information and checks the validity of the received hash value of the optimized DEX codes using ACIDroid. The server responds with a success message or *secret key* if the hash value is valid. Here, we assume that the *secret key* is needed to continue the execution of the app (e.g., the app's database files are encrypted with the *secret key*). Also, the *secret key* can be dynamically updated over time to prevent replay attacks. If the app successfully receives the *secret key*, the app normally runs with the key. Otherwise, the app forcibly terminates the app itself. Naturally, the role of such an external server can be alternatively performed by a trustworthy process running on either Android's Linux kernel or TrustZone.

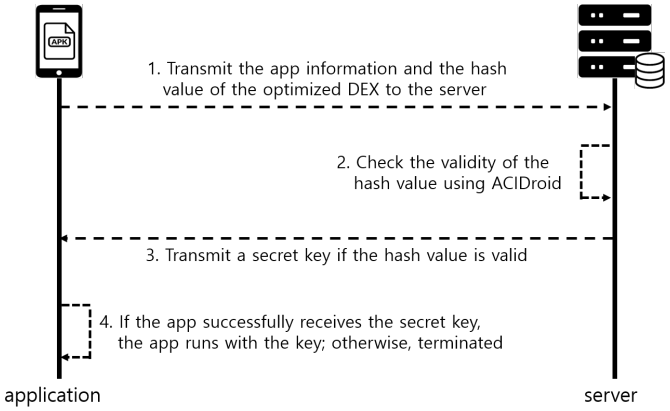

Fig. 5. App cache integrity verification process with an external server.

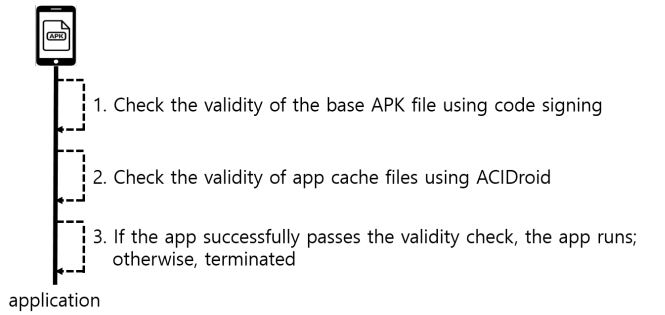

Fig. 6. App cache integrity verification process without an external server.

Even without an external verification server, ACIDroid can also verify the integrity of the app cache files through a self-defense mechanism as shown in Fig. 6. In order to check the integrity of DEX codes in app cache files, the integrity of base APK file should be first protected. Here, we assume that the integrity of base APK file can be effectively checked by signing the APK file with a *trusted* developer's signing key. If the base APK file is valid, ACIDroid extracts DEX codes from the base APK file and compares them with the optimized DEX codes in OAT cache files to identify mismatched parts between the original DEX codes and the optimized DEX codes. For those mismatched codes, ACIDroid checks whether the optimized DEX codes are normally generated using the converting rules shown in Table II. If the app successfully passes this self-

integrity check, the app normally runs continuously. Otherwise, the app forcibly terminates the app itself.

B. Limitations

We note that ACIDroid checks the code regions of opcodes and register fields rather than the region of parameter fields. We found that the parameter field values related to `vtable` are unique to each device. Therefore, it is not possible to use the optimization code generation rules in order to check the validity of parameter field values unlike opcodes and register fields. Consequently, our ACIDroid implementation can certainly result in a security hole that will be exploited by sophisticated attackers who can develop code modification attacks with the changes in `vtable` fields similar to virtual function table hijacking attacks in C++ language. However, we argue that it is very challenging to change the control flow of Android apps by only modifying the `vtable` values because there may be a small chance of generating such an exploit code with the existing methods only in the view of attacker. We will conduct more experiments to analyze the possibility of such attacks for further work.

The security of ACIDroid is somewhat limited when ACIDroid runs on the app itself. Without a platform-level secure execution environment, all self-integrity check mechanisms can be practically bypassed by reverse engineering and app repackaging [22] because the codes for the integrity check in apps can be modified by attackers. Thus, a platform-level support for ACIDroid is ultimately required.

VII. Conclusion

Android introduced the app cache mechanism to boost the performance of apps running on ART environment. However, this mechanism brings a new security concern about the protection of app cache files. To address this problem, we propose ACIDroid, which is a practical app cache integrity protection system for checking the validity of mismatched parts between the original DEX codes and the DEX codes in app cache files. Compared with the existing method [2] that uses a large size of app cache file signatures, ACIDroid can efficiently detect the modification of the optimized DEX codes (that we are particularly interested in) in app cache files.

To show the feasibility of ACIDroid, we implemented a prototype and evaluated ACIDroid's performance with several Android versions (5 to 8). Our experiments, conducted with 14 real-world Android apps, showed that ACIDroid accurately detected the modification of the optimized DEX codes with a small overhead (e.g., 8.76ms on average for Android 8) in execution time.

As part of future work, we plan to make ACIDroid provide more fine-grained and flexible inspection for a particular class or method to be examined. Our current implementation can only check the modification of the optimized DEX codes at the file level. We also intend to conduct real-world experiments through the deployment of ACIDroid in the latest Android devices, and analyze its performance under numerous varying conditions in a real-world setting.

References

[1] R. Yadav and R. S. Bhadoria, "Performance analysis for Android runtime environment," in *Proceedings of the 5th International conference on Communication Systems and Network Technologies*. IEEE, 2015.

[2] J. Wan, M. Zulkernine, P. Eisen, and C. Liem, "Defending Application Cache Integrity of Android Runtime," in *Proceedings of the 13th International Conference on Information Security Practice and Experience*. Springer, 2017.

[3] T. Kim, H. Ha, S. Choi, J. Jung, and B.-G. Chun, "Breaking Ad-hoc Runtime Integrity Protection Mechanisms in Android Financial Apps," in *Proceedings of the 12th Asia Conference on Computer and Communications Security*. ACM, 2017.

[4] J. Hildenbrand, "Android A to Z: What is the JIT?" https://www.androidcentral.com/android-z-what-jit, 2012.

[5] H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. Moon, "Evaluation of Android Dalvik virtual machine," in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM, 2012.

[6] "ART and Dalvik," https://source.android.com/devices/tech/dalvik, 2017.

[7] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, "ARTist: The Android runtime instrumentation and security toolkit," in *Proceedings of the 2nd European Symposium on Security and Privacy*. IEEE, 2017.

[8] A. Sinhal, "Closer Look At Android Runtime: DVM vs ART," https://android.jlelse.eu/closer-look-at-android-runtime-dvm-vs-art-1dc5240c3924, 2017.

[9] V. Costamagna and C. Zheng, "ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime." in *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems*. ACM, 2016.

[10] M. Sun, T. Wei, and J. Lui, "TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime," in *Proceedings of the 23rd SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[11] P. Sabanal, "Hiding behind ART," Online: https://www. blackhat. com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp. pdf, 2015.

[12] A. Ludwig and M. Mille, "Diverse protections for a diverse ecosystem: Android security 2016 year in review," https://goo.gl/6o4tBf, 2017. [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf

[13] F. Howarth, "Is rooting your phone safe? the security risks of rooting devices," https://goo.gl/axbkX9, October 2015. [Online]. Available: https://insights.samsung.com/2015/10/12/is-rooting-your-phone-safe-the-security-risks-of-rooting-devices

[14] T. Connor, "Apktool," https://ibotpeaches.github.io/Apktool/, 2018.

[15] Y. Piao, J.-H. Jung, and J. H. Yi, "Server-based code obfuscation scheme for APK tamper detection," *Security and Communication Networks*, vol. 9, no. 6, pp. 457–467, 2016.

[16] E.-R. Latifa and E. K. My Ahmed, "Android: Deep look into Dalvik VM," in *Proceedings of the 5th World Congress on Information and Communication Technologies*. IEEE, 2015.

[17] S. Wessel, F. Stumpf, I. Herdt, and C. Eckert, "Improving mobile device security with operating system-level virtualization," in *Proceedings of the 28th IFIP International Information Security Conference*. Springer, 2013.

[18] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: toward extracting hidden code from packed android applications," in *Proceedings of the 20th European Symposium on Research in Computer Security*. IEEE, 2015.

[19] "Configuring ART," https://source.android.com/devices/tech/dalvik/configure, 2018.

[20] "Dalvik Executable instruction formats," https://source.android.com/devices/tech/dalvik/instruction-formats, 2018.

[21] "Git repositories on android," https://android.googlesource.com, 2018.

[22] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 33rd Symposium on Security and Privacy*. IEEE, 2012.