Active Warden Attack: On the (In)Effectiveness of Android App Repackage-Proofing

Haoyu Ma, Shijia Li, Debin Gao, Daoyuan Wu, Qiaowen Jia, and Chunfu Jia

Abstract—App repackaging has raised serious concerns to the Android ecosystem with the repackage-proofing technology attracting attention in the Android research community. In this article, we first show that existing repackage-proofing schemes rely on a flawed security assumption, and then propose a new class of *active warden attack* that intercepts and falsifies the metrics used by repackage-proofing for detecting the integrity violations during repackaging. We develop a proof-of-concept toolkit to demonstrate that all the existing repackage-proofing schemes can be bypassed by our attack toolkit. On the positive side, our analysis further identifies a new integrity metric in the Android ART runtime that can robustly and efficiently indicate bytecode tampering caused by either repackaging or active warden attacks. By associating this new metric with two supplemental verification mechanisms, we construct a multi-party verification framework that significantly raises the bar of repackage-proofing and identify conditions under which the proposed framework could detect app repackaging without getting compromised by active warden attacks.

Index Terms—Android security, app repackage-proofing, active warden attack

1 Introduction

Android has become the most popular operating system for mobile devices [27]. Not surprisingly, the fact that Android is favored all over the world also makes apps running on the platform targets of malicious activities, among which app repackaging is an important one. A typical app repackaging adversary tampers with the internal logic of a victim app in a way satisfying her malicious purposes, and then packages the modified app and publishes it (as either a new app or a mimic of the victim) such that unwitting users may be lured into using it. Besides violating intellectual property rights, app repackaging could also lead to a number of collateral consequences, including depriving economic benefits via compromised in-app purchases or advertisements and allowing piggybacked malicious code to be executed. An early study showed that 5 to 13 percent of apps were plagiarisms in Android markets, and among 1,260 malicious apps, 86 percent were propagated via app repackaging [12]. More recent studies [1], [15] showed that more and more sophisticated tricks from traditional desktop malware samples have now emerged in app repackaging cases, including adding hook code, hiding malicious payload within resource files, mounting obfuscation, and VM-aware mechanisms. App repackaging today has also started to challenge machine-learning-based detection techniques [7]. Meanwhile, conventional countermeasure against the attack, namely off-line repackage detection [5], [10], [33], [35], suffers from shortcomings such as delayed detection, ineffectiveness against obfuscation, and lack of capability in detecting multi-generation repackaging [1], [14]. This suggests the necessity of giving Android apps the ability of defending themselves against the threat of repackaging.

Inheriting the idea of software tamper proofing, a promising countermeasure against app repackaging is to build Android apps with built-in capability of fighting off integrity violations, called *repackage proofing* [16]. To the best of our knowledge, existing repackage-proofing schemes (as of early 2020) include Droidmarking [22], Stochastic Stealthy Network (SSN) [16], AppIS [26], BOMBDROID [34], and different variations of Self-Defending Code (SDC) [6], [28]. These schemes verify integrity metrics obtained through well-defined Android APIs, including the public key used for signing the protected app and digests/checksums that can be read from or computed with certain key files. To protect these verification routines, code protection techniques, such as the self-decrypting code based on one-way functions [23] and tamper-proofing mechanisms like the guards network [4], were also adopted such that at least part of their defensive capability would survive should the attacker tries to compromise them by means of static and/or dynamic program analysis. Table 1 presents a comparison on the effectiveness of these repackage-proofing schemes against various attacks.

Although not explicitly mentioned, existing repackage-proofing solutions established their effectiveness upon an assumption that interactions between protected apps and

- *Haoyu Ma is with the School of Computing and Information Systems, Singapore Management University, Singapore 188065, and also with the School of Cyber Engineering, Xidian University, Xi'an, Shaanxi 710126, China. E-mail: hyma@xidian.edu.cn.*
- *Shijia Li and Chunfu Jia are with the College of Cyber Science, and the Tianjin Key Laboratory of Network and Data Security Technology, Nankai University, Tianjin 300071, China. E-mail: sjli@mail.nankai.edu.cn, cfjia@nankai.edu.cn.*
- *Debin Gao is with the School of Computing and Information Systems, Singapore Management University, Singapore 188065. E-mail: dbgao@smu.edu.sg.*
- *Daoyuan Wu is with the Department of Information Engineering, Chinese University of Hong Kong, Hong Kong, China. E-mail: dywu@ie.cuhk.edu.hk.*
- *Qiaowen Jia is with the Institute of Software, Chinese Academy of Sciences, Beijing 100190, China. E-mail: jiaqw@ios.ac.cn.*

Manuscript received 12 Dec. 2020; revised 15 July 2021; accepted 26 July 2021. Date of publication 30 July 2021; date of current version 2 Sept. 2022. (Corresponding author: Chunfu Jia.) Digital Object Identifier no. 10.1109/TDSC.2021.3100877

TABLE 1
Comparing Effectiveness of Existing Repackage-Proofing Schemes Against Different Repackaging Attack Strategies

Repackage-proofing scheme	app repackaging (vanilla)	app repackaging (assisted by existing static analysis)	app repackaging (assisted by existing dynamic analysis)	Our attack
Droidmarking [22]	*effective*	*effective*	partially disabled	**disabled**
SSN [16]	*effective*	partially disabled	**disabled**	**disabled**
AppIS [26]	*effective*	partially disabled	**disabled**	**disabled**
BOMBDROID [34]	*effective*	*effective*	partially disabled	**disabled**
SDC [6], [28]	*effective*	*effective*	partially disabled	**disabled**

the Android system can be trusted. However, some latest developments occurred in the Android community had raised practical challenges toward this assumption by taking advantage of certain design features of Android. For example, Android runtime works in the form of user-level shared libraries, and interactions with the framework layer utilize cached local proxies within each app's own user space. The recently emerging *Android plugin* technology [3], [21], [29] has demonstrated a typical example of such challenges. Despite its original and benign motivations for hot patching, reducing the released APK size, and etc., this technology abuses the Android application framework to intercept the communications between the "slaved" apps and the system. In this paper, we systematically study the impact of such API interception on the effectiveness of app repackage-proofing. *Unlike existing studies and industrial reports on the security risks of Android plugin [17], [24], [36], [37], our contribution is to highlight a long neglected attack scenario where an Android app is maliciously modified to defeat its built-in self-protecting mechanisms*. More specifically, we propose a new class of *active warden attack* that enables a repackaged app to falsify known integrity metrics adopted in the existing repackage-proofing schemes without root privileges. Our experiments with attack demos suggest that all existing repackage-proofing schemes are vulnerable to this new attack.

On the defense side, our detailed understanding and analysis of the active warden attack also enables us to identify a new integrity metric in the Android ART runtime which reflects an app's bytecode integrity while being static and consistent across app restarting, re-installation, and system reboots. We identify the conditions under which the proposed metric could detect app repackaging without getting compromised by active warden attacks, and argue that the proposed metric significantly raises the bar of repackage proofing by making the proposed new attack detectable. Furthermore, we introduce the native-level verification for key API call routines to elevate our new metric into a multiparty verification loop. The key idea is to utilize a carefully selected composition of different integrity verification mechanisms to cover all app components as well as routines of key API invocations, making it difficult for active warden attacks to forge all metrics at the same time.

The rest of this paper is organized as follows. In Section 2, we briefly introduce related work on app repackage proofing and the active warden attack. Following that, in Section 3, we present the general idea of our attack strategy against repackage-proofing, and then present proof-of-concept demos of this attack against all existing repackage-proofing schemes. Next, in Section 4, we propose our multiparty verification framework, and evaluate its effectiveness with experiments and analyses. Finally, we discuss some implications on our new verification framework in Section 5, and conclude the paper in Section 6.

2 Background

2.1 Repackage Proofing

Repackage proofing can be seen as a special application of software tamper proofing specifically on Android apps. So far, all existing repackage-proofing schemes verify the integrity of a protected app by injecting additional verification code into the app that performs integrity checks using Android APIs or key files of the app:

- SSN [16] and BOMBDROID [34] measure an app's integrity by checking public key inside its certificate via `Certificate.getPublicKey`, while the certificate is retrieved via `PackageManager.getPackageInfo`.
- Droidmarking [22] instead sends out pre-stored certificate information to an external verifier (via an intent) using `startService` such that the latter could check whether the certificate it receives is authenticated.
- BOMBDROID further checks the code digests, which are read from `MANIFEST.MF`.
- BOMBDROID, AppIS [26], and SDC [6] also verify the checksums of code snippets, which require reading the app's compiled code files.

Some most recent works, namely BOMBDROID, SDC and another variation [28], adopts self-decrypting code [23] and decrypts its defense code snippets using checksums of the protected app's code. SDC has two schemes, one of which constructs a customized Dalvik Virtual Machine (DVM) to support extra Dalvik instructions for the SDC decryption, while the other constructs "Twin SDC" to perform encryption (and thereafter decryption) recursively. To identify crashes caused by incorrect decryption, both SDC schemes rely on an external auditor app to check timestamps and decrypted code outputs from the SDC snippets.

Some other designs, such as those in AppIS, leverage another conventional tamper-proofing framework known as the guards network [4]. This is to build an interdependent network consisting of multiple integrity detection and abnormal response components (called the guards), making each individual guard potentially protected by some other guards. To defeat such a guards network, attacks against it need to disable all the guards together, which is difficult due to the complexity of interdependencies built into the network.

2.2 Active Warden Attack

Active Warden Attack (henceforth AWA for short) is better known as an attack model against steganography [2]. The concept was first described in the "Prisoners' Problem": Alice and Bob are in jail and wish to hatch an escape plan. Communications between them can only go through the warden, Willie. If Willie discovers any suspicious information, the escape plan fails and Alice and Bob will be thrown into solitary confinement [25]. Hence, Alice and Bob must hide their ciphertext within some innocuous-looking covertext. On the other hand, Willie has two strategies against the potential conspiracy: transmitting while monitoring the communication until abnormalities are detected (but nothing more), or acting proactively to remove possible covert messages while preserving their explicit meanings. In the second scenario, Willie is referred to as an *active warden*. Essentially, AWA is a special man-in-the-middle attack, and was included into the threat model against software watermarking and fingerprinting [8] where the warden is a program designed to fool the recognizer of such techniques.

3 AWARE: Active Warden Attack Against REpackage-Proofing

As mentioned in Section 1, the existing repackage-proofing schemes were established on an implicit assumption: *all the inter-component communications injected to provide the defense, e.g., API calls and intent-based IPCs, are assumed to be trusted*. However, we found this assumption to be faulty in real-world practices due to the existence of various program hooking techniques targeting both Android and Linux (on which Android is built). We will demonstrate that building on such a flawed assumption puts existing repackage-proofing schemes under the threat from AWA, which ends up undermining their effectiveness significantly.

We hereby refer to the AWAs that are launched specifically for compromising repackage-proofing schemes as the *AWARE* (**A**ctive **W**arden **A**ttacks against **RE**package proofing). Note that although the underlying principles utilized by AWARE are not necessarily new in the sense that they have been applied in other application scenarios (e.g., hot patching), we are the first to utilize them for a new attack of defeating app repackage proofing and to propose the engineering details realizing such an attack against all existing repackage-proofing schemes.

3.1 Overview and Threat Model

Repackage proofing retrieves and verifies certain integrity metric(s) of Android apps at runtime. Existing schemes implemented their metric acquisition routines via two approaches: the API-based and the file-based integrity checking (recall Section 2.1). Such routines, together with self-decrypting code to obfuscate the defensive payload, formed the root of trust for these schemes. Therefore, we show the effectiveness of AWARE by explaining in detail how the attack compromises both types of integrity checking and tampers with program semantics protected by self-decrypting code — should these key mechanisms be defeated, the existing repackage-proofing schemes would fail due to the lost of trustworthy sources of integrity.

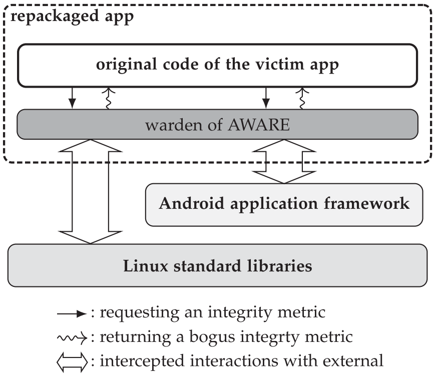

repackaged app
original code of the victim app
warden of AWARE
Android application framework
Linux standard libraries
: requesting an integrity metric
: returning a bogus integrty metric
: intercepted interactions with external

Fig. 1. The basic idea of AWARE.

Fig. 1 illustrates the basic idea of AWARE. Given a victim app under the protection of existing repackage-proofing schemes, the proposed attack injects an executable payload, i.e., the warden module, into the app's code sections. This warden is designed to take over the victim app's key interactions with Android system (or with an external verifier app), feed bogus readings to integrity checks of the embedded repackage-proofing scheme which rely on the compromised interactions, while preserving the semantics and correctness of the victim app. Of course, as an enhanced repackaging attack strategy, AWARE will also re-sign the victim app using the attacker's public key such that the repackaged instance could pass Android's signature verification during installation. We emphasize that AWARE is, by all means, still an application-level attack, i.e., it works entirely within the victim app's sandbox and memory space to forge communications with the Android runtime. It is not capable of tampering with the latter directly, and it also does not need to enslave the victim app as a plugin of another master app to work properly.

Our attack works within the same threat model adopted by the existing repackage-proofing schemes. We assume that

- both framework layer and Linux kernel/standard libraries of the Android system are unmodified; and
- neither the victim app nor the AWARE payload require root privilege or any specific permissions.

In addition, we assume that AWARE is allowed to perform offline/online analyses and to modify the victim app's code and data. That said, such analyses and modifications are not assumed to be powerful enough to bypass program protection techniques such as code patching and obfuscation.

3.2 Deceiving API-Based Integrity Checking

Integrity Verification Mechanism. Most of the existing repackage-proofing schemes perform integrity checks using Android APIs [16], [22], [34]. Built on top of Linux, an important feature of Android is that many key components of its application framework (e.g., `ActivityManagerService` or `AMS`, and `PackageManagerService` or `PMS`) run as user-level modules. When the app calls an API of such a system component, the *Binder* object of that component needs to be acquired and converted into an interface such that the target API can be properly referenced. For performance consideration, Android avoids frequent Binder acquisition by

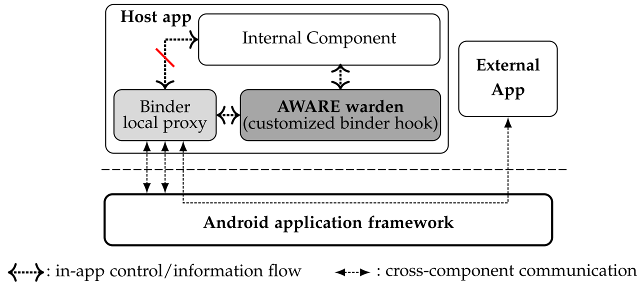

Host app
Internal Component
External App
Binder local proxy
AWARE warden (customized binder hook)
Android application framework
: in-app control/information flow
: cross-component communication

Fig. 2. AWARE's mechanism of compromising interactions with the Android application framework.

caching a local proxy within the app's user space for each framework layer component to be used directly as its interface.

Attack Strategy of AWARE. The aforementioned local proxies are maintained in the form of global variables that can be easily modified by any component within the same address space. An app could therefore overwrite these proxies to redirect them to customized hooks (as illustrated in Fig. 2), causing all IPCs between its own user space and the application framework be intercepted and manipulated. Note that this hooking technique has already been leveraged in the implementation of Android plugin toolkits (e.g., [3],[21] and [29]) to create phony system service interface exposed to slaved apps — which exist in the form of components of the plugin framework, similar to the relationship between a repackage-proofing scheme and an app being protected. Therefore, AWARE adopts this application framework hooking to deploy its warden for counterfeiting repackage-proofing metrics that rely on the related APIs, including

- the public key certificate, adopted in both SSN [16] and BOMBDROID [34]; and
- the metric-carrying intents sent out to third-party verifiers as utilized in Droidmarking [22].

3.3 Deceiving File-Based Integrity Checking

Integrity Verification Mechanism. File-based integrity check is another approach adopted in existing repackage-proofing schemes [6], [26], [34]. Specifically, the protected app retrieves certain files of itself and then verifies the file integrity to infer its overall integrity status. As discussed in Section 2.1, BOMBDROID verifies code digests of the app (MANIFEST.MF) and checksums of code snippets (ahead-of-time compiled Java methods) in addition to the app's public key. Similarly, AppIS [26] and SDC [6] adopt code checksum verification for their integrity checks.

Attack Strategy of AWARE. The "Achilles heel" of file-based integrity verification is Android's installation path generation mechanism. Android 8.0 and subsequent versions enforce a security policy to format such paths according to /data/app/⟨packagename⟩-⟨SecRan⟩==/, in which SecRan is a random suffix. As the result, these directories can no longer be assumed without looking up the app's ApplicationInfo.sourceDir field at runtime, which again must be retrieved via PMS APIs. Therefore, by manipulating PMS and all available file opening methods (e.g., Java classes File/FileInputStream, and C/C++ functions fopen/open), AWARE could deploy a *path wrapper* to intercept all requests of the victim app on acquiring its installation path or opening any files under that directory (see Fig. 3). Specifically,

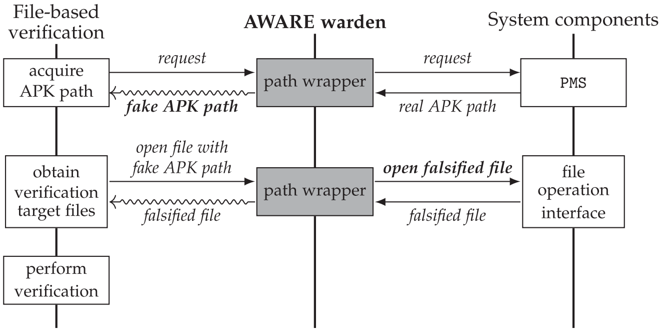


Fig. 3. Our mechanism of intercepting and manipulating file-based app integrity verifications.

- upon capturing a path acquisition request, the path wrapper generates a fake SecRan and returns a bogus APK path with the random suffix being replaced by the forgery;
- upon capturing a file opening request, the path wrapper first checks whether the target file path contains its bogus APK path, and if so, opens and returns the falsified file corresponding to the actual target (prepared in advance).

Note that the bogus APK paths are indistinguishable from repackage-proofing schemes because they are of the same format as real APK paths, while the only information source, i.e., PMS, is in control of the AWARE warden.

In most cases, AWARE's falsified files are merely the original version of those to be checked. An exception is when counterfeiting integrity verification which read ahead-of-time compiled code of Java methods as the metric. Existing work did not specify how exactly such code reading is implemented, but to the best of our knowledge, two possible ways could be used to accomplish such code acquisition. The first approach is to obtain the app's base.odex file,[1] which is also placed under the app's installation path (hence this case is not exceptional). The second approach, on the other hand, is to retrieve the linear address of base.odex by accessing the app's memory maps (provided in the system virtual file /proc/self/maps), and then reading the content of the memory sections allocated to the file. To deceive this approach, a fake memory map alone is not enough. AWARE needs to

1) load the victim app's original code files (including base.odex and other private libraries) into its address space as heap objects; and then
2) return a forged /proc/self/maps (formatted in the same way as an authentic memory map) via the hooked file opening functions, in which addresses related to the app's real code files are replaced by those pointing to the aforementioned heap objects.

To make sure that the fake memory map does not appear abnormal due to suspicious offsets among sections, AWARE

[1] Note that a third-party Android app indeed possesses the privilege to access its own base.odex and base.vdex files.

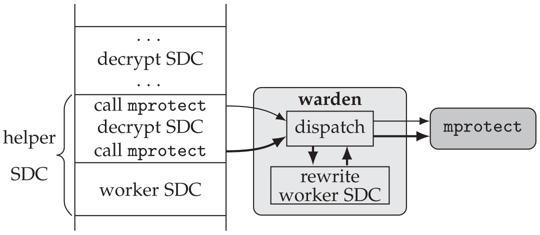


Fig. 4. AWARE's mechanism of compromising twin SDC.

may also need to reload some of the system libraries as heap objects and use addresses of these objects instead of the actual files in the fake memory map.

It's worth mentioning that since AWARE's file falsifying works by taking over the Android file system interface exposed to the repackaged app, any *in-app* integrity verification strategies that rely on integrity of the file system would also get defeated because an in-app file system checking routine only sees what the AWARE's warden shows (especially when it comes to information requested from the Android framework). To given an example, an app could try to detect AWARE by searching for potential bogus files the attack may deploy via directory inspection; however, once the app is taken over by AWARE, the attacker's warden intercepts information regarding the app's directory structure and crop it according to the attacker's interest, thus hiding the existence of bogus files from the app's inspection. On the other hand, any *out-of-app* or third-party directory inspection (e.g., that performed by a systems app) would not be deceived by AWARE. We note that such out-of-app inspection is outside the scope of this paper, and the flexibility AWARE has in disguising any necessary fake files as resource or temporary files would likely hinder such detection ineffective.

3.4 Compromising Self-Defending Code

Self-Defending Mechanism. Different from strategies discussed in Section 3.2 and 3.3, SDC [6] employs self-defending code with two schemes using a customized Android DVM and a twin SDC structure to prevent its payload from being compromised by potential adversaries (see Section 2.1) — we focus on twin SDC given that DVM is already obsolete. This design keeps the repackage-proofing logic, i.e., the worker SDCs, under the protection of encryption and obfuscation provided by the helper SDCs, making it difficult to reveal or remove them during a repackage attack.

Attack Strategy of AWARE. To attack the twin SDC structure, AWARE exploits the fact that in order to modify a user space code segment at runtime, SDC needs to switch its memory pages to writable using `mprotect` and recover after the modification. Therefore, AWARE hooks the invocation procedure of `mprotect` and intercepts the second call to this function made by the helper SDC (see Fig. 4). This second call to `mprotect` is a milestone indicating that an SDC snippet has been decrypted but is not yet executed. The AWARE warden could then rewrite the decrypted code at this particular moment for inserting any malicious activity. As for falsifying integrity measurements used as the key for decrypting the SDCs, AWARE resorts to the same means presented in Sections 3.2 and 3.3.

There are also more practical goals that can be achieved in the SDC tampering. For example, the warden could erase the logging behavior in the decrypted code such that even if an SDC segment causes a crash, the external SDC auditor would still be unable to detect the anomaly. Such SDC tampering could even evade the cross checking among different SDC snippets: the AWARE warden could store an SDC snippet before compromising it, and having the snippet recovered by the injected payload (after the fact of malicious tasks) to clean up the "crime scene". Other SDC snippets cannot detect this due to TOCTTOU (time of check to time of use).

3.5 Proof-of-Concept Demonstrations

To the best of our knowledge, there is no working prototype released by any of the previously proposed repackage-proofing schemes [6], [16], [22], [26], [34]. Commercial apps also rarely announce whether they have adopted protective techniques to defend against app repackaging or what techniques they have adopted. In this section, we resort to proof-of-concept demos to show the capability of AWARE in undermining key mechanisms of repackage proofing claimed in Sections 3.2, 3.3 and 3.4.

We have constructed a proof-of-concept AWARE toolkit[2] that injects a booter function of the AWARE warden class into the victim app's main activity to ensure that it gets executed before any repackage-proofing component takes control. This can be achieved by inserting the booter into the app's `onCreate` or `attachBaseContext` (if it is overridden by the main activity) method. At runtime, the booter initializes the warden class and invokes other methods of it to hook the Android application framework as well as other selected Java and native APIs, and thereby deploys the interception and manipulation of key repackage-proofing mechanisms.

We used `Frozen Bubble`, a FOSS (Free and Open Source Software) app which can be found in both the `F-Droid` catalogue[3] and Google Play, as our subject app. We embedded into the subject app

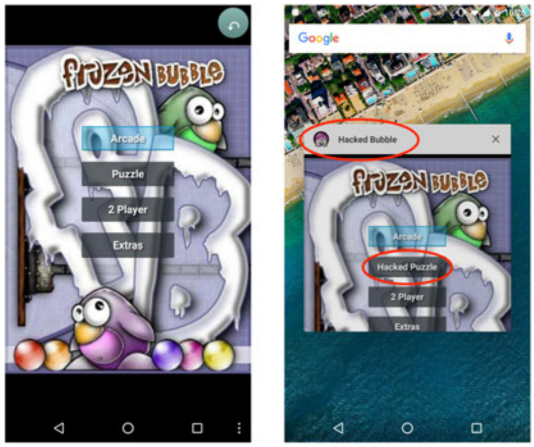
- an API-based integrity check which obtains the app's certificate via `PackageManager.getPackageInfo`, with the API called via reflection (as in SSN),
- a file-based integrity check involving the standard routine of fetching the app's base APK file (as in BOMBDROID),
- a twin SDC component built to protect a `printf` call which outputs "Hello World!".

The modified subject app was then repackaged with changes to its UI (as shown in Fig. 5) with an AWARE warden embedded to compromise the aforementioned integrity checks and SDC component.

On compromising the API-based integrity checking, Fig. 6a shows the attack payload of our warden embedded in the repackaged `Frozen Bubble` instance. Upon hooking `PMS`, our warden intercepts its invocation handler and checks whether the app is calling `getPackageInfo` and

[2] The source code of our proof-of-concept AWARE toolkit can be found at https://github.com/jnsiw/AWARE.

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



(a) original instance (b) repackaged instance

Fig. 5. Instances of `Frozen Bubble` before/after the trivial repackaging introduced for demonstrations.

obtaining its certificate (at line 29). It then replaces the certificate section in the `PackageInfo` object with a bogus one (line 30 to 32). To better demonstrate the underlying details of the attack, we also added a stack dumping command in our warden, so that it outputs stack trace of the app upon being executed. Fig. 6b shows a comparison between the real and the bogus certificate, in which ./`CERT-HOOK.RSA` is our forgery carrying a false public key prefixed with a series of "0xdeadbeef"s. We found that when executing the repackaged `Frozen Bubble` with our warden switched off, the subject app logged the correct public key starting with 0xa7191429, and no stack trace was shown (see Fig. 6c). When the app was run with the warden turned on, on the other hand, we can see from Fig. 6d that the invocation of `PackageManager.getPackageInfo` was redirected to the `PmsHookBinderInvocationHandler` of our attack payload (see the underlined record in the stack trace). Consequently, the subject app outputs the bogus certificate and false public key as highlighted at the bottom of Fig. 6d.

For the file-based integrity checking, the attack payload of our warden was slightly different from Fig. 6a, in that it intercepted `PackageManager.getApplicationInfo` such that the `sourceDir` field in the returned object could be overwritten.[4] As shown in Fig. 7a, the authentic path of the repackaged `Frozen Bubble` contained a system-generated suffix "`–66njDkL5tM04wna9fsuWzg==`" (see the logs tagged with "`PackageCodePathWrapper`", which were recorded by our warden). Nevertheless, after returning from our warden to the subject app's own code, this installation path had been given a fake suffix "`–hook–success`", as in the logs recorded by the subject app itself (those tagged with "`Detection`"). This demonstrated that AWARE could indeed manipulate the file instance returned.

Finally, Fig. 7b demonstrates the simulation result of tampering the twin SDC instance using AWARE. Without loss of generality, we switched the permission of code pages containing the worker SDC snippet of the twin SDC structure to `RWXP` at the time of decrypting. Accordingly, with `mprotect` hooked, our warden monitored and intercepted all requests of switching code pages from `RWXP` to `R-XP` (as shown in the second line of the upper half of Fig. 7b), which allowed it to correctly determine the memory range of worker SDC (1 memory page starting from 0x7fbeb22c6000). Following that, our warden immediately overwrote the parameter of the protected `printf` call, making the tampered worker SDC print "Hacked!" instead of the expected text.

Although AWARE rely on a number of carefully designed control flow interception strategies, implementing the attack is actually easy at engineering level thanks to the numerous open-source hooking and virtualization frameworks for Android [24], [36]. These existing toolkits could be applied on both the Java and native partition of an app, making it possible for an adversary to develop an AWARE payload as simple extensions of them. For example, our AWARE demo itself is partially based on such a virtualization framework, namely `whale`.[5]

4 Potential Mitigation

In this section we further discuss the possibility of mitigating the AWARE attack at the *application layer*. Our intention here is to investigate *practical* defense that are potentially deployable in the current Android ecosystem, and will leave OS-level and hardware-assisted solutions as future work.

4.1 Security Goal and Assumptions

Note that even with the AWARE attack taken into account, the purpose of the adversary is still about launching an enhanced repackaging attack, which is carried out before a victim app is installed. Accordingly, the intention of an application-level mitigation of the AWARE attack is to make a subject app capable of verifying its own integrity at runtime (at which point the off-line modification to the app has completed), and detects violations caused by either embedding the AWARE payload or other modifications for the purposes of repackaging attack.

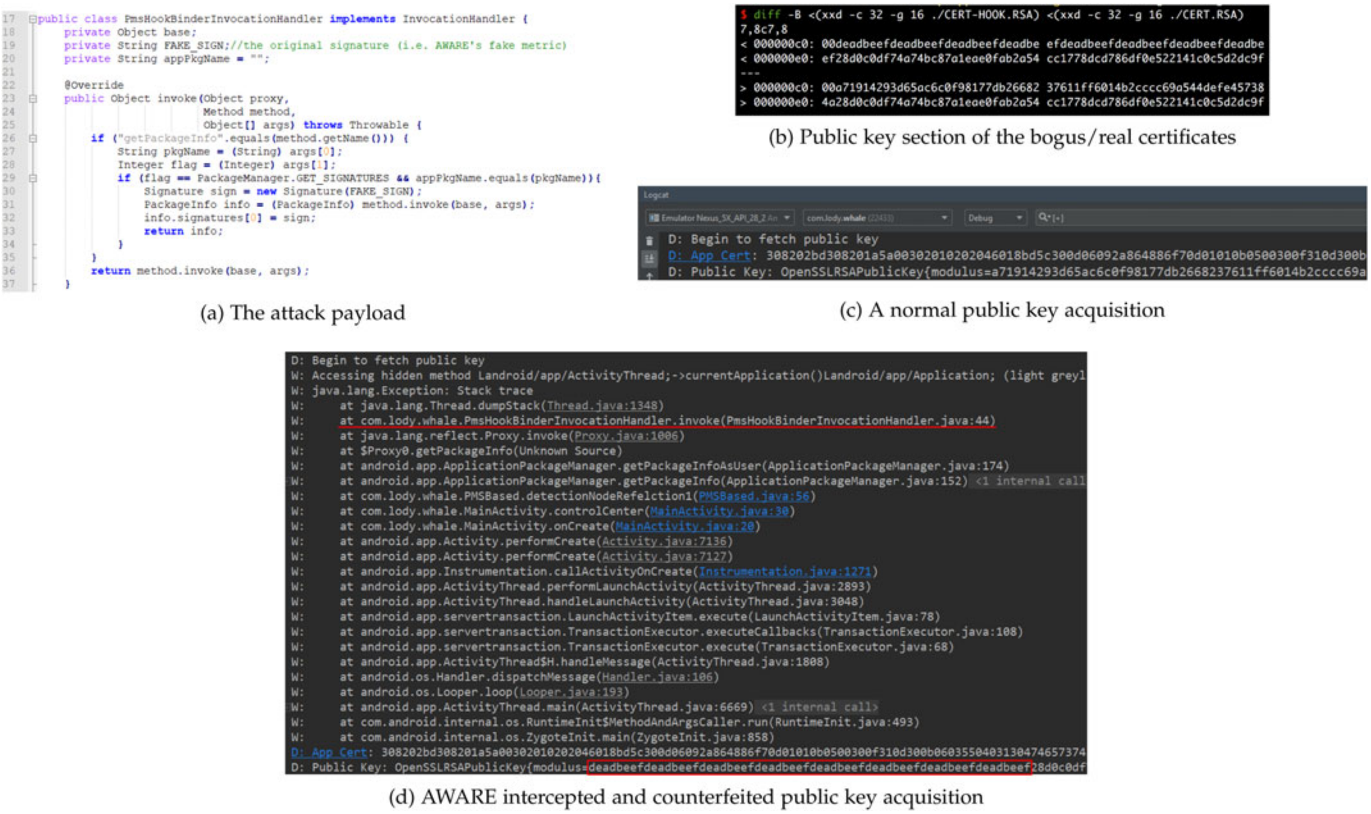
We assume that the adversary could gain access to the APK of the subject app, but not its source code. In other words, the adversary can work with encoded files (e.g., DEX files and .so libraries) within the APK, while the building process of the APK before repackaging, as well as the runtime environment in which the repackaged app runs, is out of his/her reach. More powerful adversaries capable of tampering with the target Android internal are out of our scope. In addition, we assume that the adversary wants to at least preserve the essential functionalities of the subject app, given that his main purpose is repackaging.

Finally, we emphasize that this paper never intends to bring a complete app repackage-proofing scheme. This is because a typical repackage-proofing design consists of two portions:

- components for acquiring and verifying certain integrity metrics of the subject app, as well as
- routines and mechanisms that protects the integrity of those metric acquisition and verification components.

[4] To avoid redundancy, here we skip the injected code but simply show the attack output. The same goes to the next demonstration on attacking SDC.

[5] https://github.com/asLody/whale.



```
public class PmsHookBinderInvocationHandler implements InvocationHandler {
    private Object base;
    private String FAKE_SIGN;//the original signature (i.e. AWARE's fake metric)
    private String appPkgName = "";

    @Override
    public Object invoke(Object proxy,
                         Method method,
                         Object[] args) throws Throwable {
        if ("getPackageInfo".equals(method.getName())) {
            String pkgName = (String) args[0];
            Integer flag = (Integer) args[1];
            if (flag == PackageManager.GET_SIGNATURES && appPkgName.equals(pkgName)){
                Signature sign = new Signature(FAKE_SIGN);
                PackageInfo info = (PackageInfo) method.invoke(base, args);
                info.signatures[0] = sign;
                return info;
            }
        }
        return method.invoke(base, args);
    }
```

(a) The attack payload

```
$ diff -B <(xxd -c 32 -g 16 ./CERT-HOOK.RSA) <(xxd -c 32 -g 16 ./CERT.RSA)
7,8c7,8
< 000000c0: 00deadbeefdeadbeefdeadbeefdeadbe efdeadbeefdeadbeefdeadbeefdeadbe
< 000000e0: ef28d0c0df74a74bc87a1eae0fab2a54 cc1778dcd786df0e522141c0c5d2dc9f
---
> 000000c0: 00a71914293d65ac6c0f98177db26682 37611ff6014b2cccc69a544defe45738
> 000000e0: 4a28d0c0df74a74bc87a1eae0fab2a54 cc1778dcd786df0e522141c0c5d2dc9f
```

(b) Public key section of the bogus/real certificates

```
D: Begin to fetch public key
D: App Cert: 308202bd308201a5a00302010202046018bd5c300d06092a864886f70d01010b0500300f310d300b
D: Public Key: OpenSSLRSAPublicKey{modulus=a71914293d65ac6c0f98177db2668237611ff6014b2cccc69a
```

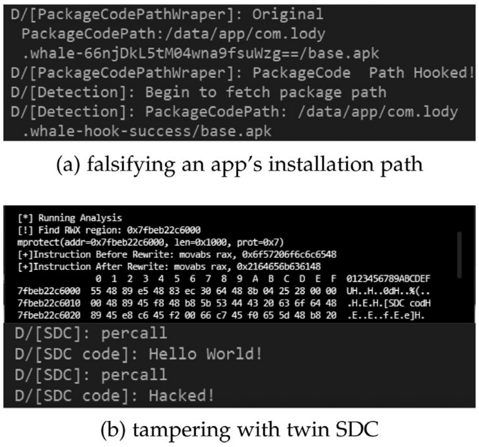
(c) A normal public key acquisition

```
D: Begin to fetch public key
W: Accessing hidden method Landroid/app/ActivityThread;->currentApplication()Landroid/app/Application; (light greyl
W: java.lang.Exception: Stack trace
W:      at java.lang.Thread.dumpStack(Thread.java:1348)
W:      at com.lody.whale.PmsHookBinderInvocationHandler.invoke(PmsHookBinderInvocationHandler.java:44)
W:      at java.lang.reflect.Proxy.invoke(Proxy.java:1006)
W:      at $Proxy0.getPackageInfo(Unknown Source)
W:      at android.app.ApplicationPackageManager.getPackageInfoAsUser(ApplicationPackageManager.java:174)
W:      at android.app.ApplicationPackageManager.getPackageInfo(ApplicationPackageManager.java:152) <1 internal call
W:      at com.lody.whale.PMSBased.detectionNodeRefelction1(PMSBased.java:56)
W:      at com.lody.whale.MainActivity.controlCenter(MainActivity.java:30)
W:      at com.lody.whale.MainActivity.onCreate(MainActivity.java:20)
W:      at android.app.Activity.performCreate(Activity.java:7136)
W:      at android.app.Activity.performCreate(Activity.java:7127)
W:      at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1271)
W:      at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2893)
W:      at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:3048)
W:      at android.app.servertransaction.LaunchActivityItem.execute(LaunchActivityItem.java:78)
W:      at android.app.servertransaction.TransactionExecutor.executeCallbacks(TransactionExecutor.java:108)
W:      at android.app.servertransaction.TransactionExecutor.execute(TransactionExecutor.java:68)
W:      at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1808)
W:      at android.os.Handler.dispatchMessage(Handler.java:106)
W:      at android.os.Looper.loop(Looper.java:193)
W:      at android.app.ActivityThread.main(ActivityThread.java:6669) <1 internal call>
W:      at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:493)
W:      at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:858)
D: App Cert: 308202bd308201a5a00302010202046018bd5c300d06092a864886f70d01010b0500300f310d300b0603550403130474657374
D: Public Key: OpenSSLRSAPublicKey{modulus=deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef28d0c0df
```

(d) AWARE intercepted and counterfeited public key acquisition

Fig. 6. A demonstration of our AWARE demo evading integrity checks based on public key acquisition.

And, *the AWARE attack works by compromising the trustworthy of the first portion of repackage-proofing defenses, but not the integrity of any of its program semantics*. As such, the goal of a mitigation of this attack should be more about forcing the adversaries to fight against the (remaining effective) second portion of existing repackage-proofing defenses. To this end, we put our focus mainly on looking for new metric acquisition approaches reliable against the AWARE attack, rather than proposing new tricks for safeguarding the repackage-proofing code. We acknowledge the limitation of an application-level solution without the help of any trusted components from the OS or hardware. The security goal of our defense is to raise the bar of repackage proofing hopefully to the extent that conducting such attacks becomes no easier than re-implementing the app, rather than (unrealistically) making the attack impossible.



```
D/[PackageCodePathWraper]: Original
 PackageCodePath:/data/app/com.lody
 .whale-66njDkL5tM04wna9fsuWzg==/base.apk
D/[PackageCodePathWraper]: PackageCode  Path Hooked!
D/[Detection]: Begin to fetch package path
D/[Detection]: PackageCodePath: /data/app/com.lody
 .whale-hook-success/base.apk
```

(a) falsifying an app's installation path

```
[*] Running Analysis
[!] Find RWX region: 0x7fbeb22c6000
mprotect(addr=0x7fbeb22c6000, len=0x1000, prot=0x7)
[+]Instruction Before Rewrite: movabs rax, 0x6f57206f6c6c6548
[+]Instruction After Rewrite: movabs rax, 0x2164656b636148
                0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
7fbeb22c6000  55 48 89 e5 48 83 ec 30 64 48 8b 04 25 28 00 00  UH..H..0dH..%(..
7fbeb22c6010  00 48 89 45 f8 48 b8 5b 53 44 43 20 63 6f 64 48  .H.E.H.[SDC codH
7fbeb22c6020  89 45 e8 c6 45 f2 00 66 c7 45 f0 65 5d 48 b8 20  .E..E..f.E.e]H.
D/[SDC]: percall
D/[SDC code]: Hello World!
D/[SDC]: percall
D/[SDC code]: Hacked!
```

(b) tampering with twin SDC

Fig. 7. AWARE compromising file-based integrity checking and twin SDC.

4.2 Multi-Party Verification Framework against AWARE

We first summarize the reasons why existing repackage-proofing schemes fail in stopping AWARE as follows:

- Insecure mechanisms — specifically, self-decrypting code is not secure when the key system functions involved (in particular, `mprotect`) cannot be trusted.
- Flawed integrity metrics — while they were adopted due to their uniqueness and coverage (e.g., public key is app-specific), their corresponding retrieval mechanisms are not trustworthy with AWARE in the game.

Intuitively, a more effective defense against AWARE thus needs a new (set of) metric(s) for Android apps which is not only sensitive to integrity violations caused by the embedding of AWARE payload, but also capable of resisting various app behavior manipulations. To this end, we propose a verification framework across an app's bytecode and native partitions, with *multiple verification mechanisms* that have each other covered within the ring of protection.

4.2.1 ART-Based Bytecode Integrity Metric

Recall that the AWARE attack requires deploying a booter at the victim app's code entry (see Section 3.5), indicating inevitable modification of the victim app's bytecode component(s), i.e., method(s) written in Java/Kotlin. Therefore, a valid metric against AWARE should indicate the integrity

TABLE 2
Selected Fields of `ArtMethod` and Their Consistency Status in Various Scenarios

Name of the field	Consistency under normal scenarios			Consistency under attacks	
	App reinstall	App restart	System reboot	Repackaging	AWARE
declaring_class	×	×	×	×	◯
access_flag_	◯	◯	◯	◯	×
dex_code_item_offset_	◯	◯	◯	×	×
dex_method_index_	◯	◯	◯	×	◯
method_index_	◯	◯	◯	◯	◯
entry_point_from_quick_compiled_code_	◯	◯	×	×	×

status of an app's bytecode with good sensitivity, while being

- app-specific (unique to each app); and
- static (remaining unchanged during normal dynamic execution and across app restarting, app reinstalling, and system rebooting).

With these in mind, we look into the Android ART runtime and particularly, a key data structure it maintains called `ArtMethod` which stores key metadata that helps resolving entry of Java/Kotlin methods inside an app's `base.odex` (or `base.vdex` for Android 8 and above). These structures are created and maintained dynamically by the runtime, and tampering with them could easily crash the corresponding app. We tested six fields shared by all different versions of `ArtMethod` to see how they respond in various scenarios. Results are presented in Table 2, where "×" means that value of the corresponding field may change under specific circumstances and "◯" indicates otherwise.

We found that 4 fields in `ArtMethod` are static in normal executions, in which only the `dex_code_item_offset_` field responds to bytecode tampering (i.e., consistent under normal scenarios while changed once being attacked). After a further investigation of the Android source code, we found that the value of this field is originated from the `code_off` field within the `encoded_method` structure of direct or virtual methods. According to its definition, `code_off` either gives the offset of the corresponding method's bytecode item within DEX file, or 0 if the method is abstract or native.[6] Being a densely encoded format, the relative offset of one section or item within a DEX file can be easily affected by modifications (e.g., as a result of the AWARE attack) changing the size of other sections/items. On the other hand, the positioning of code items inside DEX files is not sensitive to any implementation details of the Android system. With these, we identify and use `dex_code_item_offset_` of selected (multiple) Java/Kotlin (but not abstract) methods of the protected app as a metric to detect tampering of bytecode caused by AWARE (or any other code manipulation). We discuss the reliability of this integrity metric in Section 4.3.

4.2.2 Acquisition of the New Metric

Obtaining `ArtMethod` is common in Android apps with native code embedded via NDK (Native Development Kit), because when invoking a Java/Kotlin method from a native function, the callee's `ArtMethod` is required as a parameter so that JNI can locate its entry. Specifically, when carrying out such an invocation, the caller (i.e., the native function) first invokes JNI method `FindClass` or `GetObjectClass` to obtain the class of the callee, then uses `GetMethodID` or `GetStaticMethodID` (also provided by JNI) to get its `ArtMethod` structure. Therefore, our integrity metric can be fetched at least in two ways:

- Obtaining `ArtMethod` during actual method invocations from the app's native partition; or
- Introducing new payload to obtain `ArtMethod` of some selected Java/Kotlin method of the app.

The first acquisition approach is tightly bound to the protected app's own semantics, making it more difficult to be removed or compromised despite its limitation on the aspect of time-of-check. The second approach provides flexibility, but the `ArtMethod` acquisition operations might be suspicious for the lack of dependency (e.g., no subsequent invocations). As a compensation, bogus invocations guarded by opaque predicates can be inserted into the protected app to create disguising dependencies for the newly introduced `ArtMethod` acquisitions. For the implementation of such opaque structures, we refer interested readers to existing works [31], [32] since it is not a contribution of this paper.

4.2.3 Multi-Party Verification

An `ArtMethod` field alone is not enough to fight against AWARE. First, it is possible for the AWARE warden to tamper with PLT/GOT stubs inside the victim app's private libraries to intercept JNI invocations for obtaining `ArtMethod` and invoking Java/Kotlin methods (where `ArtMethod` is used as an input), so that

- in the procedure of obtaining `ArtMethod`, a fake value is assigned to the `dex_code_item_offset_` field in the returned structure to deceive repackage proofing;
- during method calling in which the fake `ArtMethod` is involved, the warden recovers the authentic structure before continuing the invocation.

Second, `dex_code_item_offset_` cannot be used to detect integrity violations of repackaging attacks that only tamper with the subject app's native code, because `code_off` of native methods are set to 0.

To address the above issues, our solution is to include the new metric into a multi-party verification framework in

[6] See https://source.android.com/devices/tech/dalvik/dex-format for the definition of `code_off`.

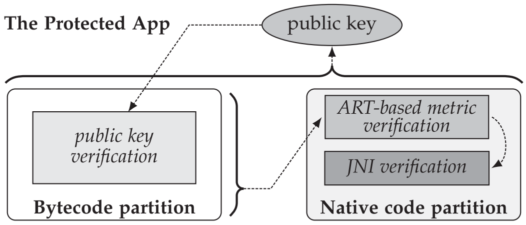


Fig. 8. Our multi-party verification framework using ART-based integrity metric and PLT/GOT verification.

which two additional integrity verification mechanisms are introduced to form a ring of protection against AWARE. The first supplemental verification targets the app's JNI invocation routines by checking the integrity of the corresponding PLT/GOT stubs. In Android apps, calling a native method from another native one also leaves a return address pointing to the caller method. Using this return address as an anchor point, location of the to-be-checked PLT/GOT stubs can be assumed by fixed offsets. Our JNI verification payload thus leverages the return address left by the caller of its residing method to inspect the presumed location of certain PLT/GOT stubs within the executable file where it is deployed. Since the PLT stubs are code pieces that look up the addresses in the GOT section, our verification inspects their concrete texts to determine whether any of them had been compromised by techniques like inline hooking. On the other hand, the GOT stubs are actual offsets for the external symbols as filled in by the linker at load time, making their values unknown at the off-line phase. Therefore, to verify the GOT stub of a JNI method, our verification is designed to inspect the distance between the GOT offset of the subject JNI method and those of other selected JNI methods. In this way, should the subject GOT stub be compromised, the inspected distances would demonstrate anomalies because the rogue GOT offset no longer points to an entry inside `libart.so`. By injecting JNI verification into the protected app's native code, our verification framework could then include the acquisition procedure of `ArtMethod` into its coverage, hence establishing the credibility of our new integrity metric.

Note that neither `dex_code_item_offset_` nor the JNI verification mechanism could check the overall integrity of the app's native code. This brings out the second supplemental integrity checking of our verification framework, which inspects the app's public key from its Java/Kotlin methods (same as some existing schemes [16], [22]). This conventional metric is picked because, as illustrated in Fig. 8, by indicating whether the protected app has been (unauthorizedly) re-signed, the public key metric implies its integrity as a whole (including that of its native code). Meanwhile, with the JNI verification mechanism endorsing its credibility, our `dex_code_item_offset_` metric can be deemed as a measure for the integrity of the protected app's bytecode partition in general. This allows our verification framework to trust the authenticity of the retrieved public key, given that attacks such as AWARE (with the purpose of manipulating this signature) will result in bytecode tampering involving multiple methods of the app, and hence would be detected. Intuitively, the three verification mechanisms of our framework leaves no unchecked "loose ends" behind.

4.3 Security Effectiveness

In this section, we evaluate the effectiveness of leveraging `dex_code_item_offset_` as an integrity metric of Android apps, and give an empirical discussion on its resilience against targeted attacks.

4.3.1 Against Attacks Using Toolkits in the Wild

First, we consider the AWARE attack which resolves DEX files of a victim APK into Smali code with `Apktool`,[7] injects the warden at the Smali level while carrying out other code tampering, then rebuilds the package using the building tool chain provided by `Apktool`. As discussed in Section 3, this is effective against conventional repackage-proofing schemes.

To test the effectiveness of the `dex_code_item_offset_` metric under such typical AWARE and app repackaging attacks, we use `Instagram` as an example (given its popularity) and examined the `code_off` of Java methods in its DEX files (which, as explained in Section 4.2, will be assigned to the `dex_code_item_offset_` field of the `ArtMethod` structures of these methods on the end-user devices) before and after repackaging. Two different samples were produced in this test. The first sample was the product of a trivial repackaging, i.e., no changes were made except signing the package with a new certificate. The second sample was embedded with the AWARE warden deployed by our attack prototype. Table 3 gives the `code_off` of five methods selected from the app, showing that both the trivial repackaging and the insertion of AWARE warden resulted in the change of `code_off` for all the selected methods. We obtained similar results on both emulators and real Android devices.

Note that tampering with methods that are not selected to be monitored using `dex_code_item_offset_` could potentially still get detected due to change of offsets induced on methods being protected. That said, we encourage protection of all important methods (those of essential functionality) with our new metric to gain the best security property.

4.3.2 Against attacks using advanced toolkits

By further looking into the building process of DEX files, we found that the difference between building tool chains in `Apktool` and `Gradle` is a major factor which contributes to the varying method offsets. Specifically, `Apktool` uses `dexlib2` to build the DEX files which does not organize the data section's items (including code, strings, types, etc.) according to the sequence defined by the Android official documentation. This means that another test is needed to understand whether the `dex_code_item_offset_` metric is still effective against advanced attacks where the adversary utilizes the official tool chain to build the repackaged DEX files.

We did not find any publicly available repackage tools with the desired building system. Therefore in this test, we simulated a series of app repackaging attacks on `Frozen Bubble`, the same subject as used in Section 3.5, and again compared the `code_off` values of selected methods. Specifically, we directly manipulated the Java source of the app before building its DEX files using `Gradle`, which we

[7] https://github.com/iBotPeaches/Apktool

TABLE 3
code_off of Selected Methods in Instagram Before and After App Repackaging (Carried Out Using Apktool)

Method	Value of code_off		
	Original	Trivial repackaging	AWARE
com.instagram.app.InstagramAppShell−> onCreate()	0x35d4f4	0x90b4ec	0x90c0f0
com.instagram.mainactivity.MainActivity−> onCreate(Landroid/os/Bundle; p0)	0x3b8edc	0x94335c	0x943f60
com.instagram.model.mediasize.TypedUrlImpl−> getHeight()	0x518fa0	0x948c00	0x949804
com.instagram.adshistory.fragment.RecentAdActivityFragment−> isOrganicEligible()	0x6f8814	0x908f4c	0x909b50
com.instagram.reels.fragment.ReelDashboardFragment−> onActivityResult()	0x753a9c	0x96212c	0x962d30

TABLE 4
code_off of Selected Methods in Frozen Bubble Before and After Simulated "App Repackaging" (Carried Out Using the Standard Gradle Tool Chain)

Method	Value of code_off			
	Original	Repackaging (trivial)	Repackaging (incremental)	AWARE
com.efortin.frozenbubble.AccelerometerManager–> isListening()	0x29b3c	0x29b3c	0x29b3c	0x29d10
com.efortin.frozenbubble.NetworkManager–> cleanUp()	0x2a730	0x2a730	0x2a730	0x2a904
org.jfedor.frozenbubble.FrozenGame–> releaseBubbles()	0x37dd8	0x37dd8	0x37dd8	0x37fac
org.jfedor.frozenbubble.PenguinSprite–> getTypeId()	0x41008	0x41008	0x4100c	0x4281c
org.jfedor.frozenbubble.Sprite–> saveState()	0x41fcc	0x41fcc	0x41fd0	0x437e0

believe is the closest possible mimic to a real app repackaging attack that leverages the official building tool chain. In addition, to better understand the effectiveness of the dex_code_item_offset_ metric, we conducted the simulated app repackaging transformations in three different settings: a trivial repackaging without any code modification, a trivial repackaging with the size of a specific method (releaseBubbles in the FrozenGame class) being increased by four bytes, and finally, a non-trivial repackaging supported by AWARE.

Table 4 shows the result of our simulation. We can see that when using the official building tool chain, a trivial repackaging which did not tamper with any code would no longer change the offset of the victim app's methods. However, such repackaging is incapable of defeating any other repackage-proofing metric, e.g., the app's public key. On the other hand, we found that a small increase in the size of one of the app's methods had shifted the position of all methods below it accordingly; see method getTypeId in class PenguinSprite and method saveState in class Sprite in Table 4. Finally, in the case of AWARE, offsets of all selected methods were affected because the attack payload introduced new metadata which increased the size of various sections of the resulting DEX file, including (but not limited to):

- A new class_def_item in the class_defs section;
- New method_id_items in the method_ids section;
- Name of the new class and its methods as additional entries in the string_ids section.

In DEX files, all sections mentioned above are placed in front of the data section. Therefore, due to the dense encoding of the DEX format, position of the entire data section (including the code items inside it) will be shifted accordingly. These observations lead to an argument that even if the adversary switches to the official building system, dex_code_item_offset_ is still effective in detecting meaningful app-repackaging and AWARE attacks.

4.3.3 Against Targeted Attacks

Last but not least, we consider the attack scenario where the adversary is committed to deceive our new integrity metric. Specifically, we want to discuss the possibility of modifying the subject app's DEX file in a delicate way, such that the AWARE payload can be embedded while none of the app's existing methods is shifted to a new offset. A potential example is the "callee-side rewriting" strategy [30] in which the adversary directly adds the AWARE payload into target DEX files of the victim app as an additional code snippet appended at the end of the code section, hoping that this could avoid altering the offset of any of the existing methods. This means that the adversary has to rewrite the DEX file at binary level rather than modifying the Smali code to avoid uncontrollable factors introduced by the DEX building process.

Due to the strict validation rules enforced on the format, there are limitations on how an additional code snippet can be inserted into a DEX file. Specifically, each method in a DEX file must be a continuous code section, and code within a method is not allowed to simply jump out of its boundaries to a rogue code section. Containing such invalid control flows will result in a DEX file being unable to pass Android's verification due to section overlap or non-zero padding.[8] Under this condition, even at binary level, the available options for the adversary to inject the AWARE payload into a DEX file are to either embed it as new classes/methods, or merge it into some existing methods within the file. On top of that, in order to deceive our new integrity metric, the payload injection must further preserve the

[8] Readers could find Android's latest DEX verification semantics at https://android.googlesource.com/platform/art/+/refs/heads/master/libdexfile/dex/dex_file_verifier.cc, where padding and overlap between items in a DEX file is checked by DexFileVerifier::CheckIntraSection().

original layout of the target DEX file, leading to a number of additional requirements.

- In case of embedding the payload as new classes/methods, and assuming that the adversary has already placed the new methods at the end of code section to avoid tampering offsets of other methods, he still needs to ensure that the DEX file's header (which includes the lists of classes/methods/prototypes contained in the file) remains of the same size as the original one.
- In case of merging the payload within existing methods, the adversary must ensure that the size of the subject methods do not change after the reconstruction.
- Finally, in case of merging the payload into multiple existing methods, the adversary must choose the subject methods carefully such that invocations among them would not be considered as illegal.

Although fulfilling the above requirements when directly manipulating a DEX file is not impossible, we argue that our new metric, when used in conjunction with our two supplementary measures in a multi-party verification manner, significantly raises the bar of repackaging attacks, potentially to the extent that such attacking effort exceeds that needed for a re-implementation of the subject app.

4.3.4 Limitations of Our New Integrity Metric

As admitted in Section 4, `dex_code_item_offset_` metric can only be leveraged by Android apps with native code. This design choice limits the completeness of our scheme, although going native is the trend of Android apps and we could expect more and more apps embracing NDK in the near future [13].

Moreover, recall that the threat model of the AWARE attack assumed unmodified Android system; see Section 3.1. However, as we have mentioned, an important feature of Android exploited by AWARE is that many of its system modules exist in the form of user-level shared libraries that can be modified inside any user apps. This also includes `libart.so` which contains key JNI methods required by the `dex_code_item_offset_` metric; see Section 4.2.3. Therefore, if we assume a stronger threat model in which the adversary is allowed to tamper with the behavior of Android system libraries, he could then attempt to intercept JNI invocations related to `ArtMethod` by hooking the related JNI methods from the platform side (i.e., within `libart.so`), or compromising the `JNIEnv` structures which are maintained in the `.data.rel.ro` section of `libart.so`. These approaches go beyond the scope of PLT/GOT checking as part of our verification framework. A possible countermeasure could be to further enhance our protection on the integrity of JNI invocations, for example, by extending the PLT verification routine to trace address lookup process down to the location of JNI methods/`JNIEnv` structures within the system library and verifying their checksum directly. However, we see such attacks which tamper with system components more of a system security issue rather than an application one. Our opinion is that *Android should further enhance its mandatory access control policy and stop third-party apps from modifying system libraries (even in their own address space)*.

TABLE 5
An Empirical Comparison on the Performance of Obtaining Integrity Metrics in Repackage-Proofing

Integrity metric	Average overhead	Code bloat
public key (direct API call)	0.8946 ms	178 bytes
public key (using reflection)	1.7042 ms	542 bytes
digests in `MANIFEST.MF`	6.1545 ms	136 bytes
`dex_code_item_offset_`	0.0048 ms	120 bytes

The existing access control policy of Android appears to mainly focus on the enforcement of app sandboxing. For instance, it adopts SELinux to prevent an app from modifying things that may compromise the behavior of other processes, and it blocks certain `procfs` files that may leak the execution status of other apps. Modifying system components to affect the execution of the current app itself, on the other hand, was not constrained because such behavior was not considered a violation of app sandboxing. However, as shown by existing studies on the risk of app virtualization [17], [24], [36], [37] and now by our AWARE attack, the highly modularized structure in fact allows part of an Android app to attack other components of its own and cause significant consequences. In other words, enforcing per-process sandboxing alone is no longer enough, and we suggest that Android could take the integrity of its user-level system components more seriously.

4.4 Performance Overhead

We made an empirical comparison between the overhead of retrieving the conventional integrity metrics and that of retrieving the `dex_code_item_offset_` metric we proposed. Four different integrity metric acquisition components were implemented to respectively obtain the app's public key via API and reflection, to read the digests within `MANIFEST.MF`, and to access the value of `dex_code_item_offset_`. Table 5 shows the comparison of average performance over 50 runs and code bloat for running all the test components on a Google Pixel 3 XL. The result suggests that the tested component for retrieving our new metric causes significantly lower overhead in both time and memory space. It is worth noting that the purpose of this comparison is merely to show that applying our integrity metric does not introduce any additional performance bottleneck. Compared to checking bytecode integrity by computing checksums of .DEX files or AOT compiled code segments of the app (like in AppIS and SDC), `dex_code_item_offset_` is a better option with regard to performance impact, because reading the entire sections in the linear address space is inefficient due to the *demand paging* mechanism, while obtaining `dex_code_item_offset_` raises no such concern.

5 Discussion

5.1 AWARE versus Off-line Repackage Detection

Recall that to launch an AWARE attack, it is necessary to insert a booter into bytecode of the victim app so that the deployed warden could intercept key defensive app behaviors. This raises a question: does such payload correspond to new signatures for off-line repackage detection?

Note that booter of the warden class of AWARE can be implemented as trivial as a simple invocation, as long as it can directly control the warden's main body. This therefore allows the use of obfuscation techniques to conceal the AWARE warden class, making it hard to be distinguished via static analysis. In addition, being originating from similar techniques, behavior of the AWARE warden class is close to that of an app virtualization framework. As shown in recent studies [24], [36], app virtualization frameworks have attracted millions of users and downloads, and cases of benign apps adopting app virtualization for software engineering and/or security reasons are not uncommon. For example, our tests showed that an e-commerce app `Lazada` with more than 100 million downloads has mounted with a virtualization framework called `Atlas`[9]. We therefore argue that it could be difficult to resort to either static or dynamic behavioral signatures to identify the AWARE attack.

5.2 Repackage-Proofing or Obfuscation?

Considering the limitations of our verification framework against an AWARE adversary with the *stronger* threat assumptions stated in the last subsection, we admit that it is difficult to deploy an effective repackage-proofing defense when components of Android system themselves could not be trusted. Under the status quo, applying *code obfuscation* techniques [9], [11], [20], [38] to protect the apps against unauthorized program analyses could still be a valuable application-level countermeasure against app repackaging (given that program analysis is a necessary phase before carrying out such attacks). However, code obfuscation must consider adversaries dedicated on a particular target (besides large-scale automated program analysis), which could make it less effective when being applied on `Dalvik` bytecode originated from strongly-typed Java sources (which has been pointed out in [11]). Furthermore, code obfuscation alone only helps increase the difficulty of program analysis.

5.3 Repackage-Proofing versus Remote Code Attestation

Some may wonder that to cope with the threat of AWARE (and app repackaging in general), could the so-called *remote attestation* technology be "another way out" and provide effective defense [18], [19]? We stress that fundamentally, remote attestation is of a different scenario (trusted computing) compared to repackage-proofing, in that its functionality relies heavily on hardware and/or system-level supports. Note that compared to previous app repackaging techniques, AWARE is more difficult to defend against *at application-level* because it exploits the fact that the Android architecture is designed to be semi-implanted into each app's sandbox, making it possible for third-party code to "play god" (i.e. gain at least some part of the system's capability without having the corresponding privileges). Defending such attacks from OS-level, on the other hand, is considered out of our scope (although such a strategy might indeed serve as a different security mechanism in preventing the threats focused on in this work). That said, indeed we cannot assert that there are absolutely no other application-level defense approaches which could also mitigate the threat of AWARE. We leave this as an open problem to be explored in the future.

6 Conclusion

In this paper, we systematically studied existing repackage-proofing schemes, and proposed an active warden attack (named AWARE) that is able to bypass integrity checks of all previous schemes. We also showed the effectiveness of this new attack with proof-of-concept demos. To the best of our knowledge, we are the first to identify the threat of AWARE to repackage proofing. On top of these, we proposed a new integrity metric and its associated multi-party verification framework. Specifically, we introduced a new ART-based bytecode integrity metric, which, under the support of supplemental verifications on certain JNI invocation routines, is able to effectively indicate code tampering on an app's Java/Kotlin methods caused by both app repackaging and the AWARE attack. Our analyses and evaluations suggested that this new integrity metric can be an effective mitigation against AWARE, and the multi-party verification framework with its participation is resilient to a number of targeted attack strategies. Our empirical study also suggested that the overhead of retrieving such a metric is acceptable.

Acknowledgments

The authors would like to thank the anonymous reviewers and the associate editor for providing valuable feedbacks that helped improving this paper. This work was supported in part by the National Key R&D Program of China under Grant 2018YFA0704703, in part by the National Natural Science Foundation of China under Grants 61972215 and 61972073, in part by the Natural Science Foundation of Tianjin under Grant 20JCZDJC00640, and in part by the Singapore National Research Foundation, National Satellite of Excellence in Mobile Systems Security and Cloud Security, under Grant NRF2018NCR-NSOE004-0001.

References

[1] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon , "Androzoo: Collecting millions of Android apps for the research community," in *Proc. IEEE/ACM 13th Work. Conf. Mining Softw. Repositories*, 2016, pp. 468–471.

[2] R. J. Anderson and F. A. Petitcolas, "On the limits of steganography," *IEEE J. Sel. Areas Commun.*, vol. 16, no. 4, pp. 474–481, May 1998.

[3] asLody, "VirtualApp," 2018. [Online]. Available: https://github.com/asLody/VirtualApp

[4] H. Chang and M. J. Atallah, "Protecting software code by guards," in *Proc. ACM Workshop Digi. Rights Manage.*, 2001, pp. 160–175.

[5] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 175–186.

[6] K. Chen, Y. Zhang, and P. Liu, "Leveraging information asymmetry to transform Android APPS into self-defending code against repackaging attacks," *IEEE Trans. Mobile Comput.*, vol. 17, no. 8, pp. 1879–1893, Aug. 2018.

[7] X. Chen *et al.*, "Android HIV: A study of repackaging malware for evading machine-learning detection," *IEEE Trans. Inf. Forensics Secur.*, vol. 15, pp. 987–1001, 2020.

[9] https://github.com/alibaba/atlas.

[8] C. S. Collberg, C. Thomborson, and G. M. Townsend, "Dynamic graph-based software fingerprinting," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 6, 2007, Art. no. 35.

[9] S. Dong *et al.*, "Understanding Android obfuscation techniques: A large-scale investigation in the wild," in *Proc. 14th Int. Conf. Secur. Privacy Commun. Syst.*, 2018, pp. 172–192.

[10] L. Glanz *et al.*, "Codematch: Obfuscation won't conceal your repackaged APP," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 638–648.

[11] Z. He *et al.*, "Exploiting binary-level code virtualization to protect Android applications against APP repackaging," *IEEE Access*, vol. 7, pp. 115062–115074, 2019.

[12] X. Jiang and Y. Zhou, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 95–109.

[13] Z. Kan, H. Wang, L. Wu, Y. Guo, and G. Xu, "Deobfuscating Android native binary code," in *Proc. 41st Int. Conf. Softw. Eng. Companion*, 2019, pp. 322–323.

[14] L. Li, T. F. Bissyandé, A. Bartel, J. Klein, and Y. L. Traon, "The multi-generation repackaging hypothesis," in *Proc. 39th Int. Conf. Softw. Eng. Companion*, 2017, pp. 344–346.

[15] L. Li *et al.*, "Understanding Android app piggybacking: A systematic study of malicious code grafting," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 6, pp. 1269–1284, Jun. 2017.

[16] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu, "Repackage-proofing Android APPS," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2016, pp. 550–561.

[17] T. Luo, C. Zheng, Z. Xu, and X. Ouyang, "Anti-plugin: Don't let your APP play as an Android plugin," 2017. [Online]. Available: https://www.blackhat.com/docs/asia-17/materials/asia-17-Luo-Anti-Plugin-Don't-Let-Your-App-Play-As-An-Android-Plugin-wp.pdf

[18] M. Nauman, S. Khan, X. Zhang, and J.-P. Seifert, "Beyond kernel-level integrity measurement: enabling remote attestation for the Android platform," in *Proc. Int. Conf. Trust Trustworthy Comput.*, 2010, pp. 1–15.

[19] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1429–1446.

[20] M. Protsenko, S. Kreuter, and T. Müller, "Dynamic self-protection and tamperproofing for Android APPS using native code," in *Proc. 10th Int. Conf. Availability, Rel. Secur.*, 2015, pp. 129–138.

[21] Qihoo 360, "RePlugin," 2018. [Online]. Available: https://github.com/Qihoo360/RePlugin

[22] C. Ren, K. Chen, and P. Liu, "Droidmarking: Resilient software watermarking for impeding Android application repackaging," in *Proc. 29th ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2014, pp. 635–646.

[23] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proc. 16th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2008.

[24] L. Shi, J. Fu, Z. Guo, and J. Ming, ""Jekyll and Hyde" is risky: Shared-everything threat mitigation in dual-instance apps," in *Proc. 17th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2019, pp. 222–235.

[25] G. J. Simmons, "The prisoners' problem and the subliminal channel," in *Proc. Adv. Cryptol.*, 1984, pp. 51–67.

[26] L. Song *et al.*, "APPIS: protect Android APPS against runtime repackaging attacks," in *Proc. IEEE 23rd Int. Conf. Parallel Distrib. Syst.*, 2017, pp. 25–32.

[27] Statista, "Number of IOS and Google Play mobile APP downloads worldwide from 3rd quarter 2016 to 4th quarter 2018 (in billions)," 2019. [Online]. Available: https://www.statista.com/statistics/695094/quarterly-number-of-mobile-app-downloads-store/

[28] S. Tanner, I. Vogels, and R. Wattenhofer, "Protecting android APPS from repackaging using native code," in *Proc. Int. Symp. Found. Prac. Secur.*, 2019, pp. 189–204.

[29] wequick, "Small," 2018. [Online]. Available: https://github.com/wequick/Small

[30] M. Wißfeld, "ArtHook: Callee-side method hook injection on the new Android runtime art," Ph.D. Dissertation, Saarland Univ., Saarbrücken, Germany, 2015.

[31] D. Xu, J. Ming, and D. Wu, "Generalized dynamic opaque predicates: A new control flow obfuscation method," in *Proc. 19th Int. Conf. Inf. Secur.*, 2016, pp. 323–342.

[32] H. Xu, Y. Zhou, Y. Kang, F. Tu, and M. Lyu, "Manufacturing resilient bi-opaque predicates against symbolic execution," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2018, pp. 666–677.

[33] W. Yang *et al.*, "AppSpear: Bytecode decrypting and DEX reassembling for packed Android malware," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2015, pp. 359–381.

[34] Q. Zeng, L. Luo, Z. Qian, X. Du, and Z. Li, "Resilient decentralized Android application repackaging detection using logic bombs," in *Proc. Int. Symp. Code Gener. Optim.*, 2018, pp. 50–61.

[35] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2014, pp. 25–36.

[36] L. Zhang *et al.*, "APP in the Middle: Demystify application virtualization in Android and its security threats," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 1, pp. 1–24, 2019.

[37] C. Zheng, T. Luo, Z. Xu, W. Hu, and X. Ouyang, "Android plugin becomes a catastrophe to Android ecosystem," in *Proc. 1st Workshop Radical Experiential Secur.*, 2018, pp. 61–64.

[38] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang, "DIVILAR: Diversifying intermediate language for anti-repackaging on Android platform," in *Proc. 4th ACM Conf. Data Appl. Secur. Privacy*, 2014, pp. 199–210.