POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Grab'n Run: Practical and Safe Dynamic Code Loading in Android

Relatore:
Prof. Federico Maggi

Tesi di Laurea Magistrale di:
Luca Falsina, matricola 798965

Anno Accademico 2013–2014

a Roberta,
Cinzia e Guido.

Abstract

Recent studies [Poeplau. 2014] showed that developers of Android applications and frameworks, even the most famous ones (e.g., Facebook, Google Mobile Ads SDK), need to load code dynamically. This technique has the advantages of minimizing the code memory footprint and enabling silent updates strategies to decouple updates of the main application from those of its third-party libraries.

Unfortunately, there are security drawbacks as well. Firstly, malware authors may use dynamic code loading to bypass antivirus checks by loading malicious code at runtime from an apparently benign application. Secondly, Android does not perform any code verification for dynamically loaded code. Therefore, a man-in-the-middle attacker can effectively modify the dynamically loaded code executed on the victim's machine without being spotted. These vulnerabilities appeared in 16% of the top 50 free applications of the Google Play Store in August 2013. Arguably, the main source of error is assuming that developers are security experts, which is not often the case.

In this work, we remove such assumption and we propose a backward-compatible redesign of the Android API functions needed for implementing dynamic code loading, making this functionality secure by default. We implemented our proposed design in a Java library, named Grab'n Run (GNR), that can be incorporated in any Android project. Differently from previous works, GNR requires no modification of the underlying runtime, ensuring easier adoption. To further help developers migrating existing applications, we also designed and implemented a repackaging tool, which rewrites dynamic-code-loading calls to port them to use our secure API.

We validated GNR through a case study involving 9 Android developers. Without GNR, 6 of them introduced security vulnerabilities by using HTTP connection for retrieving the code instead of HTTPS, whereas 4 of them introduced another vulnerability by storing the code in a world-writable location, and, in the end, all of them forgot to implement custom integrity checks on the fetched code. The participants also confirms that the learning effort for GNR is little or close to zero and that this library is easier to maintain, simpler to read, and more flexible than native `DexClassLoader` API. Finally, by comparing the performance of GNR against `DexClassLoader`, we found out that the overhead introduced by our library on load operations is negligible.

Sommario

Studi recenti [Poeplau, 2014] hanno mostrato che gli sviluppatori di applicazioni e framework per Android, anche i più celebri e complessi (e.g., Facebook, Google Mobile Ads SDK), hanno spesso l'esigenza di caricare codice dinamicamente. Questa tecnica di programmazione ha indubbi vantaggi, sia perché minimizza l'uso del codice presente in memoria durante l'esecuzione (caricandone solo la porzione strettamente necessaria), sia perché permette di implementare aggiornamenti silenziosi, cioè dà la possibilità alle librerie di terze parti, incluse in un'applicazione, di scaricare ed eseguire l'ultima versione del proprio codice senza forzare ogni volta un aggiornamento dell'applicazione principale. Questa politica risulta essere ad ovvio vantaggio dell'utente finale, che potrà, da un lato, ridurre il numero di volte in cui é costretto ad aggiornare l'applicazione, e dall'altro, beneficiare delle ultime funzionalità e patch di sicurezza, introdotte nelle librerie utilizzate dalle sue applicazioni.

Purtroppo, questo meccanismo ha anche due principali svantaggi in termini di sicurezza. Primo, è utilizzato dagli autori di malware per bypassare controlli antivirus, caricando solo a run time il codice malevolo all'interno di un'applicazione apparentemente innocua, installata sul telefono della vittima. Secondo, le API di Android per il caricamento di codice dinamico non implementano alcun controllo sull'integrità del codice caricato, né tanto meno sull'identitá di colui che lo ha implementato. Per questo motivo, mentre lo sviluppatore che carica una applicazione sul Google Play Store è sicuro che, grazie al meccanismo di code signing, l'applicazione installata sul device dell'utente non sia stato modificata nel tragitto, la stessa conclusione non vale per il codice caricato dinamicamente. Questa falla permette ad un aggressore man-in-the-middle di modificare il codice caricato dinamicamente e, dunque, di manipolare il comportamento dell'applicazione durante l'esecuzione, senza che il sistema operativo se ne possa accorgere. Vulnerabilità di questo tipo sono state riscontrate nel 16% delle 50 applicazioni gratis più scaricate sul Play Store in Agosto 2013. Discutibilmente, tali vulnerabilità, così come quelle derivanti dall'uso errato delle librerie crittografiche, esistono per l'assunzione, molto spesso errata, che gli sviluppatori siano esperti di sicurezza.

In questo lavoro, rimuoviamo questa assunzione e proponiamo una reimplementazione delle API di Android per il caricamento di codice dinamico, rendendo questa funzionalità sicura di default e retro-compatibile con le API esistenti. Abbiamo implementato questa idea in una libreria Java, chiamata Grab'n Run (GNR), che può essere inclusa facilmente in ogni progetto Android. Differentemente dalle precedenti soluzioni, GNR non richiede alcuna modifica del framework sottostante e ciò ne garantisce una semplice adozione. Per facilitare gli sviluppatori nel compito di migrare le loro applicazioni, abbiamo anche

sviluppato uno strumento di repackaging, che riscrive le chiamate per il caricamento di codice dinamico sostituendole con invocazioni alle nostre API sicure, senza richiedere allo sviluppatore né il codice sorgente, né alcuna modifica nel codice dell'applicazione da migrare.

Abbiamo valutato GNR attraverso un caso di studio condotto su 9 sviluppatori Android. Senza GNR, 6 di loro hanno introdotto vulnerabilità di sicurezza utilizzando una connessione HTTP per recuperare il codice da caricare invece di una HTTPS; d'altra parte, 4 di loro hanno introdotto un'altra vulnerabilità salvando il codice in una posizione sovra-scrivibile da chiunque in memoria; infine, nessuno di loro si è ricordato di implementare degli ulteriori controlli di sicurezza volti a verificare l'integrità del codice, prima di caricarlo. Inoltre, i partecipanti hanno confermato che la difficoltà per imparare ad utilizzare GNR è minima, se non addirittura nulla, e che questa libreria è più facile da manutenere, più semplice da leggere, e più flessibile rispetto alla corrispondente API nativa (i.e., `DexClassLoader`). Infine, comparando le performance di GNR rispetto a `DexClassLoader`, abbiamo verificato che il ritardo introdotto dalla nostra libreria sulle operazioni di caricamento è trascurabile.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Dynamic code loading, or DCL, is a programming technique to execute code loaded at runtime. This code may come from different sources such as the local storage or from a remote location (e.g., storage server). DCL offers some advantages: first, it minimizes code memory footprint since only the static code is stored in memory for the program's lifetime; second, it enables the implementation of silent update strategies, for example decouple updates of the main program from those of its third-party libraries, or whenever a continuous software release must be implemented. These requirements should be met without bothering the user. In Android, developers can accomplish DCL through several application programming interface (API) functions, namely `DexClassLoader`, `PathClassLoader`, and `android.content.Context.createPackageContext`.

Unfortunately, DCL has significant security drawbacks: firstly, malware authors may use it to bypass antivirus checks, by loading malicious code at runtime from a statically benign application; secondly, Android API do not implement integrity check on the code loaded dynamically. This implies that, differently from the Google Play Store, where users are sure that installed applications are exactly as implemented by the developers thanks to code signing, they cannot rely on this property for code loaded dynamically. Because of this, a man-in-the-middle (MITM) attacker, who succeeds in modifying the byte code of the container used for DCL, can manipulate the execution flow of the application.

Moreover, also developers of benign applications introduce security bugs in their application. More precisely, they can introduce these bugs by fetching remote code in an unsafe way, or storing it in a world-writable location, or forgetting to include integrity checks to verify that the code to load has not been spoofed. As shown in a study published in [15] over 1,632 popular applications from Google Play Store in 2012, loading external code in an insecure way was an issue in as much as 9.25% of these applications and in 16% of the top 50 free ones in August 2013. These numbers highlight that implementing remote DCL in a safe way is an issue for Android developers. In Chapter 2, we present this problem and, in particular, how errors introduced by developers may lead a MITM attacker to execute arbitrary code by exploiting sloppy implementations of remote DCL.

Previous work proposed different methodologies to mitigate this issue. Poeplau et al. [15] proposed a modification of Android's runtime to add integrity checks on DCL through an external verification service, so as to ensure that

only verified code is loaded. A significant drawback is that this approach requires a partial rewriting of Android runtime, thus requiring an OS update on all the Android devices, which is cumbersome considering the issue of Android market fragmentation [14], not to mention the difficulty of ensuring backward compatibility with the recent introduction of the ART (Android RunTime) executable format [4]. Differently, Vidas and Christin [18] presented a protocol for end-to-end verification of Android applications. The approach proposes to use DNS to enforce a public key infrastructure (PKI). More in the details, the authors propose that developers must place the certificate to authenticate an app on a DKIM or TXT record of their own domain name, which is required to match the reversed Java package name of the container to validate. In our opinion, this protocol is too rigid (e.g., a developer may not be able to enforce the proposed match on his domain). Moreover, the signature is checked upon installation and not at runtime, which implies that any code change between installation and execution, will pass unnoticed, rendering it useless to mitigate the exploitation of DCL vulnerabilities.

Having considered the limitations of the current approaches, we propose to mitigate the problem at the origin, by reducing the chances that a developer introduces target vulnerabilities in terms of insecure implementations of DCL.

In this thesis, we propose Grab'n Run, or GNR, a Java library that helps developers implementing DCL in a secure way. This library is backward compatible with current API and porting apps to use it requires little effort for developers. Our tool is effective and innovative since it relieves the developers from thinking about security. GNR implements a caching strategy to contain performance overhead and to work partially even when little or no connectivity is available. Alongside, we propose a repackaging tool to help developers in migrating their applications to use our secure API. This tool does not require the source code of the app to patch, but simply a copy of its application package file and some settings on how to carry out the repackaging process.

GNR implements a verification protocol, which validates each code container, used as a source for DCL, through signature verification against a trusted certificate. In particular, the protocol stores securely-fetched containers into an app-private folder to prevent their tampering and it retrieves the certificate from a remote URL either directly provided by the developer or reconstructed from reverting the package name of the class to load. The protocol allows DCL only on successfully verified containers. In November 2014 we released GNR as an open source project, publicly available on GitHub [10], whereas we plan to release the repackaging tool in May 2015.

We evaluated GNR both qualitatively and quantitatively. First, to verify our claim that GNR is easy to use for developers and more secure than the native API, we set up a use case study: We contacted 9 Android app developers with different levels of expertise, asking them to implement a DCL snippet in a toy app, then to implement the same functionality using GNR. Even if we explicitly asked developers to write their code carefully, many of them introduced security vulnerabilities (6 of them failed in retrieving code securely, 4 of them in storing it, and not even one of them thought about implementing custom integrity checks). These statistics, together with the positive feedbacks from the participants, support our claim. Then, we compared the execution time of `DexClassLoader` versus GNR API and we show that, excluded the time to fetch certificates for the verification, strongly influenced by network latency,

performance overhead of our solution is almost negligible compared to the native API.

To summarize, our contributions over current state of the art are:

- We propose a verification protocol that ensures by design secure remote DCL by reducing the chances for a developer to introduce vulnerabilities because of insecure implementation of DCL.
- We implement this protocol into Grab'n Run, a Java library to make DCL secure in Android benign applications, easy to use and to integrate in any project, and whose performance overhead on load operations is negligible over native `DexClassLoader` API.
- We propose and implement a repackaging tool, which takes an application and some user preferences and port it automatically to use GNR API, without requiring developers to write code or the source code of the application to patch.

Chapter 2

Motivation

In this chapter, after a recap on Android and some insights on its security model (Section 2.1), we will focus the attention on DCL by presenting, at first, `DexClassLoader`, one of the native API for this task. Next, we will discuss how benign Android app developers can accomplish and benefit from DCL, as well as, how malware authors can use it in several ways to bypass Android security model and detection tools (i.e., Google Bouncer) (Section 2.2).

After this overview on DCL, we will restrict our view only to benign app developers, and, in particular, to the issue represented by sloppy-security implementations of remote DCL in Android apps, which can ultimately lead a MITM attacker to execute arbitrary code on the target device (Section 2.3).

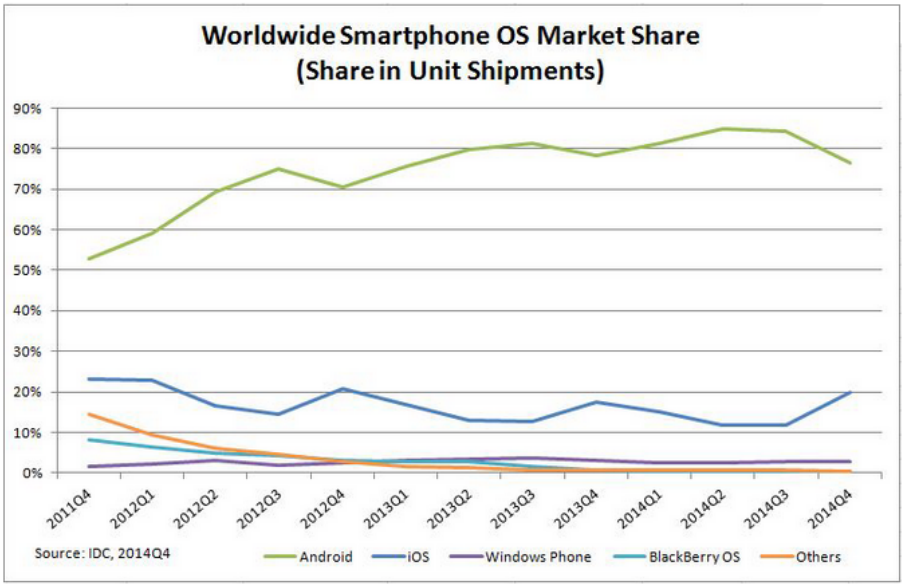
Later (Section 2.4), we will present some of the related solutions from the research community that previously tried to fix this issue and their corresponding shortcomings.

Finally, at the end of this chapter (Section 2.5), we set the goals for this thesis work and the constraints we had to bear in mind in the design of our system.

2.1 Background: Android and its security model

Smartphones gained more and more importance over the last decade. In this general trend, Android, an OS for mobile devices based on the Linux kernel, has raised its relevance over the years and it currently dominates the market share, as shown in Figure 2.1. There are several reasons behind this enormous success but the main one is that Android's source code is released under open source license by Google, in the so called Android Open Source Project (AOSP), and therefore carriers and hardware vendors can customize it freely before selling their devices to the users. The result of this process is a multitude of different Android devices and this effect is also known as fragmentation.

Regarding the field of security, Android poses a challenging ecosystem for researchers because of several reasons: (1) nowadays, mobile devices contain an outstanding amount of personal data and, thus, attackers are more than motivated to steal them to gain money or to compromise devices to create distributed botnets; (2) the open source nature of Android makes it a suitable target for attackers, who may try to exploit a zero-day vulnerability, found



Period	Android	iOS	Windows Phone	BlackBerry OS	Others
Q4 2014	76.6%	19.7%	2.8%	0.4%	0.5%
Q4 2013	78.2%	17.5%	3.0%	0.6%	0.8%
Q4 2012	70.4%	20.9%	2.6%	3.2%	2.9%
Q4 2011	52.8%	23.0%	1.5%	8.1%	14.6%

Source: IDC, 2014 Q4

Figure 2.1: Worldwide smartphone OS market share (Q4 2014). Source: International Data Corporation (IDC) [7]

by inspecting its code, to compromise millions of devices in one shot; (3) as explained at page 17 of Chapter 1 of [13], although fragmentation can be seen as a limitation on the scalability of the exploits of an attacker, it also makes infeasible complete code auditing for security researchers because many actors can customize the original code, thus introducing security vulnerabilities.

In the next subsections (Subsection 2.1.1 - Subsection 2.1.3) we deepen some background topics by recalling, in this order, the five-layer architecture of the Android OS, the complete toolchain for the compilation of Android code, and the major security countermeasures introduced in this OS to prevent attackers from succeeding in their exploits on the system.

2.1.1 Architecture

The overall architecture of Android consists of five layers. While describing them, we start from the bottom layer (the Linux kernel), till the top one (the application layer). This separation provides a fine-grained abstraction, thus allowing a developer to extend the functionalities of the device without caring

about the lower level details. In particular, the five layers are:

1. **Linux kernel**. On the lower level of the architecture Android presents a customized version of the Linux kernel with many changes and additions, also in terms of security (see Subsection 2.1.3). The kernel works as an abstraction layer for the underlined hardware of the device and it provides drivers to access all the physical components (e.g., Camera, Bluetooth, WiFi), along with the mechanisms to manage memory, processes, and the networking area.

2. **User-space native code**. This layer's components include system and networking services, as well as libraries, such as OpenSSL, SQlite and libc. In this part of the system resides also the libraries for managing 2D and 3D graphics (OpenGL/ES), as well as media codecs. These libraries are usually written in C/C++, and then expose a Java interface to the higher-level components.

3. **Android runtime**. The aim of this layer is satisfying the need for Android OS to run in an embedded environment, like the one of mobile devices, where battery, memory, and CPU are limited and power consumption must be constrained. In particular, this layer contains two main components: the first one is the set of core libraries that includes Java programming language API (e.g., collection classes, I/O elements, utilities), whereas the second one is the runtime environment, where both applications and the Android framework are executed. Till the release of the latest version of Android (i.e, Lollipop 5.0.1), the default runtime environment of the OS was DVM, a register-based VM designed to interpret DEX byte code to generate an optimized version of it, the optimized DEX (ODEX) byte code. The structure of the DVM was designed to be particularly light and shared across processes, such that a device could run multiple VMs efficiently. This enables every Android application to run in its own process, with its own instance of the DVM. However, starting from Lollipop, DVM was replaced by ART, the new Android runtime environment, introducing relevant changes in the compilation process, alongside improvements in the garbage collection, and more support for developers during both development and debugging phase [2]. For what concerns this thesis, the important notion is that ART, although generating a different final type of executable code, the executable and linkable format (ELF), takes exactly as input the same DEX byte code accepted by the DVM to grant backward compatibility. In particular, Figure 2.2 compares the I/O requirements of both DVM and ART.

4. **Android application framework**. This layer provides the developer with API to manage all features of an Android device and it represents a bridge between applications and the runtime environment (i.e., DVM, or ART). It includes components for managing the user interface, the application life-cycle, and the retrieval of data from the device's sensors.

5. **Android applications**. This upper layer contains all the applications installed on the device. They can be roughly broken into two subparts: pre-installed applications, including Google, original equipment manufacturer, and mobile carrier-provided ones, automatically shipped with the

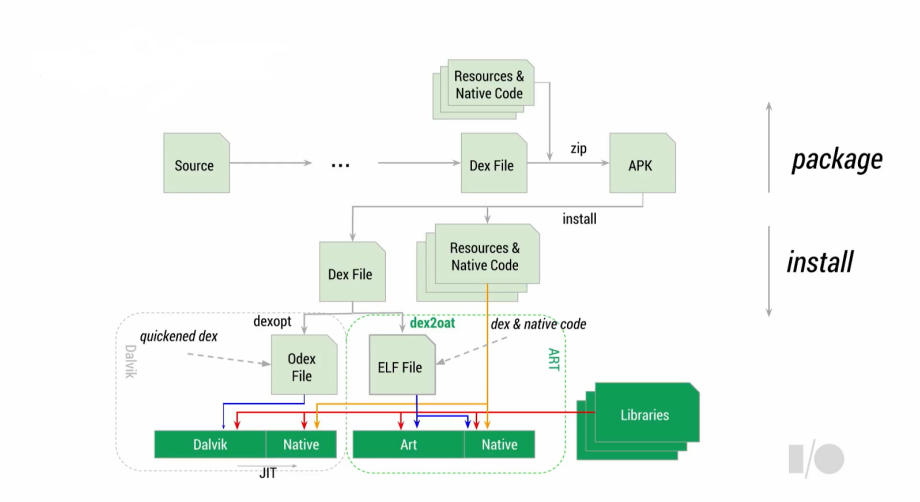


Figure 2.2: A comparison between the input and the output of both DVM and ART.

device, and user-installed apps, which are those ones installed by the user from either Google Play, an alternative market, or manually.

2.1.2 Toolchain

Figure 2.3 summarizes the compilation process for Java and Android applications:

In particular, once a developer finished to write the source code of his application and he wants to run it on a mobile device, the development process, behind the curtains, looks like this:

1. The developer writes his code in the Java language.
2. The source code is compiled into Java byte code (.class) by means of a Java compiler (e.g., javac).
3. The resulting class files are translated into Dalvik byte code by dx, a specific build tool in the Android Software Development Kit (SDK), which outputs a single `classes.dex` file.
4. The resulting DEX file is provided in input to the Android runtime environment (DVM, or ART), which generates its own optimized version of the file (respectively, an ODEX or an ELF), and finally loads and interprets it to execute the application.

2.1.3 Security model and mechanisms

In this last subsection on the Android OS, we will present the mechanisms and tools used in Android to prevent an attacker from compromising the operative system. For the first two paragraphs of this section, we summarize the main notions presented by [13] in Chapter 2 (Page 25-34).

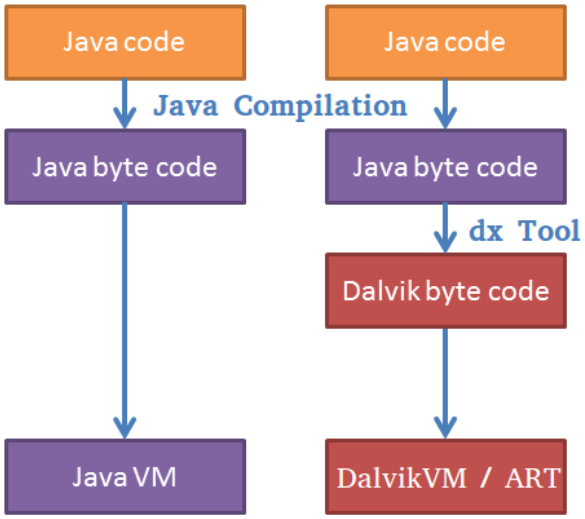


Figure 2.3: A comparison of the compilation process between a standard Java program (left side) and an Android application (right side).

Sandbox Android is built on top of the Linux kernel, inherits directly from it the concept of process isolation. Processes run as separate users and, therefore, cannot interfere with each other, such as by accessing the other process' memory space. Alongside, Linux applies the so-called principle of least privilege, which means that, to access certain resources, any process must belong to a user, or to a group of users, who holds the specific permission for accessing the resource; we can summarize this principle as: "A user can modify only resources for which he holds a permission, everything else is denied from being accessed". Android conjugates this model by using a sandbox based on standard Linux process isolation, unique user ID (UID) for almost all the applications, and restricted file system permissions. In particular, Android shares the concept of Linux's UID and Group ID (GID), but it also defines a map of names to unique identifiers (i.e., numbers), known as Android ID (AID). The initial mapping of the AID contains reserved, static entries for privileged and system-critical users, but later a range of AID is used for identifying app UID. In additions to AID, Android also reuses the concept of supplementary groups to regulate whether an application can access shared or protected resource. For example, whenever an application becomes a membership of the AID_INET supplementary group, it is allowed to open sockets. When an application is executed, its UID, GID, and supplementary groups are assigned to its corresponding process. Running under a unique UID and GID enables the operative system to enforce lower-level restrictions in the kernel and for the runtime environment to control inter-app interactions.

Permission model Android provides three different levels of permissions: (1) API, (2) file system, and (3) inter-process communication (IPC) ones. API permissions (e.g., `READ_PHONE_STATE`, `WRITE_EXTERNAL_STORAGE`) include those used for controlling access to high-level functionality within the Android API or framework and, in some cases, third-party frameworks. On the other hand, file system permissions are relevant because applications' unique UID and GID are, by default, given access only to their respective data storage paths on the file system. However, certain supplemental GID entitle for the access to shared resources, such as sd cards or other external storage. As an example, consider an application that requires the API permission `WRITE_EXTERNAL_STORAGE`: This app will see its UID added to the corresponding supplemental group for the sd card and, thus, the OS will grant it write access at this path. Finally, IPC permissions are those directly related to communication between app components. Granted permissions can be enforced either by the runtime environment, whenever the application invokes certain methods, or by the kernel, or a library at a lower level within the OS. In particular, some of the higher-level permissions are enforced by corresponding lower-level OS capabilities. To establish the app user's rights and the supplemental groups that the app should join, Android parses the `AndroidManifest.xml`, an entry in the APK, which resumes essential information on the application (e.g., its package name, the required API permissions, the main components in it). At install time, the Android OS extracts all the permissions from the manifest and uses these entries to grant the appropriate rights to the corresponding application's process.

Code signing Code signing has the goal of guaranteeing (1) integrity, which means that the executed code is actually the same code written by the developer, and (2) signer authentication, which implies that we can always retrieve the key used to sign the code. Code signing leverages the concept of asymmetric cryptography, where a public and a private key pair is associated to a developer, typically with a certificate. Depending on the implementation, a full-fledged public-key infrastructure (PKI) is also deployed so that the authenticity of the code can be verified against a chain of trust rather than just cryptographically. All the Android developers have to generate their own key pair, and the Google Play Store will identify, and recognize them, based on the corresponding certificate. Moreover, Android policy requires developers to sign their applications, or the Google Play Store will reject them when uploaded on the market. During installation, the device performs signature verification on the target APK and allows or blocks the process accordingly. Although this process seems solid, it has some weaknesses:

- Firstly, the Android OS does not enforce code integrity over time. This means that, once the signature was verified at installation time, if the attacker is somehow able to modify the application code, the system cannot notice this change and it will still run the code without raising any warning.
- Secondly, Android accepts self-signed certificates, which means that the subject of the certificate is also the signer of it, and, therefore no third-party authority grants for the trustworthiness of the certificate, except for the certificate's owner itself. We argue that, despite the security weaknesses caused by this choice, there are two aspects that must be considered.

First, verifying the entire certificate chain can be costly for an embedded system, or sometimes even unfeasible (e.g., no connectivity, and thus no means to fetch the certificate-revocation list). The recent discoveries about the fallacies derived from incautious implementations of PKIs (e.g., compromised CAs, bogus root CA certificates found in computers or used to sign certificates) highlight how difficult it is to engineer and deploy a bulletproof PKI. Second, on the positive side, Android forces developers, who wants to update their already published application, not to modify the package name (i.e., a unique identifier across the whole Play Store for each application), and the private key used to sign the APK, which means that an attacker, who wants to impersonate the benign developer, cannot just create a fake certificate, but she is forced to steal, or sniff somehow, the developer's private key. These requirements are presented and commented in [16].

Finally, we provide some technical details on how signatures are stored inside of an APK, since this will come useful in Chapter 3. An APK is a format of package file used to aggregate Java class files, linked metadata, and resources. In particular, alongside the Java byte code, APK archives present also a special file entry called `MANIFEST.MF` (it is important not to confuse this entry with the `AndroidManifest.xml` since they are two completely different files). When an APK container is signed by invoking the jarsigner tool on it, the manifest file is patched to contain the digests of the signed entries in the archive. Moreover, the tool stores other details on the signatures' digest of all the entries into a signature file, and it creates a signature block file, which contains the actual cryptographic signature in a not human-readable, binary file. Notice that APK constitutes an extension of the standard Java archive (JAR) files, thus inheriting the signing procedure directly from those. More details on how the signature process works in a JAR can be found in [9].

2.2 Dynamic code loading: principles, uses, and vulnerabilities

After having discussed about the Android OS and its security model, we move the focus to DCL, a programming technique to execute additional code, not been previously defined with the rest of the program that is, instead, named static code because of its translation at compile time. Code that is loaded at run time may come from different sources such as local storage on the device or from a remote location (e.g., storage server).

In Android, developers can benefit from DCL thanks to several native API (e.g., `DexClassLoader`, `PathClassLoader`, and `android.content.Context.createPackageContext`), present since the first versions of Android, and remained untouched, except for minor fixes, till nowadays. Under a security perspective, code loaded dynamically runs with the same permissions of the application that loaded it. This means that, if the main running application owns an API permission, like `android.permission.INTERNET`, also any piece of code dynamically loaded by it will benefit of the same permission, and, thus, in this case, it can open sockets. At a lower level, this implies that dynamically loaded code runs in the same process of the main application and, therefore, it has the same UID,

GID, and it is a member of the same supplemental groups.

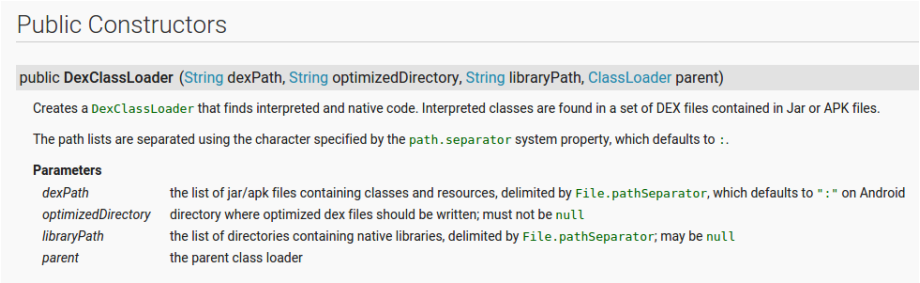
Finally, under an operational point of view, whenever an Android device performs DCL, the OS provides the corresponding DEX classes to the run time environment (i.e., DVM, or ART), which will load and execute them. Since the runtime environment is responsible for this operation and it can interpret only Dalvik byte code, the three input files supported as sources for DCL are: (1) direct DEX files, (2) JAR archives that contains an extra `classes.dex` entry (i.e., invoking the dx tool on the original JAR, as presented in Subsection 2.1.2), and (3) APK archives, whose classes are automatically translated into a DEX file by the Android tools at build time.

2.2.1 DexClassLoader API

`dalvik.system.DexClassLoader` has been present since the first versions of Android API (version 3), and, as the name suggests, belongs to the family of the *Class Loaders*. A developer can use them to load additional code at run time by providing the URI of an external file, which stores the implementation of the Java classes to load. In particular, since `DexClassLoader` relies on the Android runtime, it takes as input only sources containing DEX files and translates all of them into either an ODEX or an ELF file. Finally, for performance improvements, it caches these optimized files into a folder that the developer indicates at the time of `DexClassLoader` instantiation.

Although using this class properly is not a naïve task, official documentation on the topic is miserable: the only affordable resource for this purpose is the API reference page in the Android Developer website [3]. Still, this page lacks many useful pieces of information. Firstly, description on how to use `DexClassLoader` class is poor. Moreover, differently from other well-written sections of Android documentation and tutorials, there is not even a simple code example to show how to properly setup `DexClassLoader` instances. These two deficiencies together makes learning effort for this API consistent for a developer, especially if he is willing to load additional code from a JAR container, since nowhere it is explained how to add a `classes.dex` entry to a JAR container (i.e., invoking the dx tool, present in the Android SDK, on the target container). Last but not least, the reference page does not present any of the security bugs coming from a sloppy use of `DexClassLoader` class, except for the danger of storing optimized classes on the external storage, which, as the guide correctly suggests, a developer can mitigate by requiring an application-private folder for caching optimized classes. After this overview on the behavior, we introduce the two relevant methods of `DexClassLoader` API.

The first one is the class constructor, whose signature is shown in Figure 2.4. It is responsible of interpreting the first String parameter `dexPath`, which is a list of strings pointing to source containers for DCL, separated by a special path char (default value is ":"). Each one of these containers is analyzed and, if any DEX file is found in the resource, the constructor translates it into an optimized file format (i.e., ODEX, or ELF) and then stores the resulting file at the location provided by the developer through the second String parameter `optimizedDirectory`. A developer may also decide to attach in the process extra native libraries, written in C/C++, by filling in the third String parameter `libraryPath`, and he can even opt for changing the parent loader of the `DexClassLoader` instance by setting a different class loader object in the fourth



Public Constructors

public **DexClassLoader** (String dexPath, String optimizedDirectory, String libraryPath, ClassLoader parent)

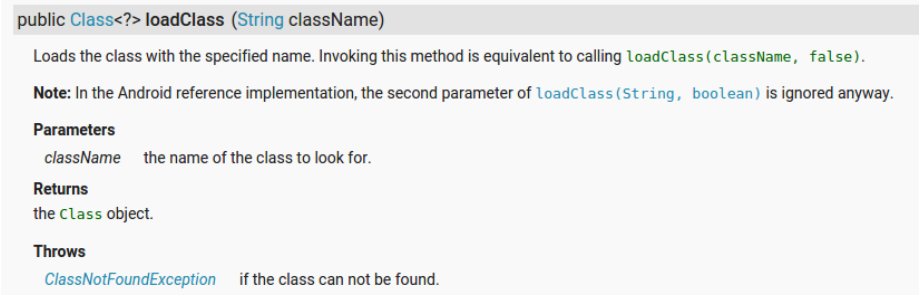
Creates a `DexClassLoader` that finds interpreted and native code. Interpreted classes are found in a set of DEX files contained in Jar or APK files.

The path lists are separated using the character specified by the `path.separator` system property, which defaults to `:`.

Parameters

dexPath	the list of jar/apk files containing classes and resources, delimited by `File.pathSeparator`, which defaults to `":"` on Android
optimizedDirectory	directory where optimized dex files should be written; must not be `null`
libraryPath	the list of directories containing native libraries, delimited by `File.pathSeparator`; may be `null`
parent	the parent class loader

Figure 2.4: `DexClassLoader` constructor. A screen shot of the signature of the constructor for the `DexClassLoader` class, as presented in the API reference page.



public Class<?> **loadClass** (String className)

Loads the class with the specified name. Invoking this method is equivalent to calling `loadClass(className, false)`.

Note: In the Android reference implementation, the second parameter of `loadClass(String, boolean)` is ignored anyway.

Parameters

className	the name of the class to look for.

Returns

the `Class` object.

Throws

ClassNotFoundException	if the class can not be found.

Figure 2.5: `DexClassLoader loadClass()`. A screen shot of the signature of the `loadClass()` method, as presented in the `ClassLoader` API reference page.

parameter `parent`. Still, modifying these last two parameters from the default values (respectively `null` and `getClassLoader()`) is usually not necessary.

The second method of interest is `loadClass()`, whose signature is presented in Figure 2.5. This method, inherited directly from `ClassLoader`, takes as an input a String parameter that is the name of the class that `DexClassLoader` look for, among all the cached ODEX or ELF files created by the constructor. If `DexClassLoader` finds one class implementation matching the target class name, it returns an instance of such a class; otherwise it raises a `ClassNotFoundException`. Notice that, to successfully find and load a class, `DexClassLoader` requires its full name, which embodies both the package name and the class simple name, separated by a dot.

In the end of this subsection, we present a naïve code example in Listing 2.1 showing how to properly initialize a `DexClassLoader` instance and how to set it up for loading an external class, named `com.example.MyClass`, from a JAR archive, whose path is stored in the helper variable `jarContainerPath`. This snippet also shows how to setup an application-private folder for caching the optimized version of the code container and which exceptions must be handled in the process.

Listing 2.1: DexClassLoader example snippet.

```
MyClass myClassInstance = null;
String jarContainerPath = getFilesDir().getAbsolutePath() + "/exampleJar.jar";
String dexOutputDirPath = getDir("dex", MODE_PRIVATE).getAbsolutePath();

DexClassLoader mDexClassLoader = new DexClassLoader(  jarContainerPath,
                                                      dexOutputDirPath,
                                                      null,
                                                      getClassLoader());

try {
       Class<?> loadedClass = mDexClassLoader.loadClass("com.example.MyClass");
       myClassInstance = (MyClass) loadedClass.newInstance();

       // Do something with the loaded object myClassInstance
       // e.g. myClassInstance.doSomething();

} catch (ClassNotFoundException e) {
       e.printStackTrace();
} catch (InstantiationException e) {
       e.printStackTrace();
} catch (IllegalAccessException e) {
       e.printStackTrace();
}
```

2.2.2 Benign developers and dynamic code loading

After presenting an example of API for DCL, the next question that we address is how benign developers of Android applications can benefit from the use of it. Indeed, there are several reasons that justify DCL in application development:

- **Behavior customization**. This is the most obvious reason why a developer should use DCL. The concept is to select the piexe of code to load dynamically depending on the configuration of the running mobile device. As an example, think about an application, customized by a phone vendor, using the camera, which needs, depending on the model of the user's device, to load a different class to handle it properly. Also suppose that, although being produced by the same vendor, some devices may have both a front and a rear camera, whereas some others only the rear one and, thus, we should find a solution that manages properly this differences in the hardware components. Well, a neat way to abstract properly over this plethora of different configurations would be to set up, in the main application, a specific component that, at run time, checks the model of the mobile phone and load dynamically the correct handler class, so to support each camera properly.

- **Flexible code reuse**. Many applications may decide to share a certain library container, common to all of them (e.g., video codec, or graphic libraries) by dynamically loading classes from it, instead of having a local copy of the same library file for each app. The plus is that storage space is saved for better uses; the minus is that, since these applications are sharing the same source container, we should put extra care on how the system manages this container to avoid possible security threats.

- **Application extensibility**. The typical scenarios for showing this advantage are games that let the user add extra features by paying a certain

fee or premium versions of popular applications. The idea here is to introduce extra functionalities into the same application and, therefore, DCL can be used to load code from a previously fetched, remote source archive that contains, for example, code for a new level of a game, or an unlocked functionality for a premium application.

- **Self-upgrade functionality**. This feature regards especially non-standalone libraries (e.g., advertisement frameworks) included into other applications. The default update distribution mechanism from application stores conflicts with continuous release-oriented development practices, where small and frequent updates are released often. In particular, non-standalone libraries developers are forced with the standard process to rely on the final application, using their libraries, to have their code updated to the latest version, since the app developer is responsible to import always the latest versions of all the libraries in his code. On the other hand, we think that it would make more sense to have the system working in the opposite way round, that is, every time that a new version update is released for a library, all the applications should just be pointed to run it automatically. We name this latter strategy, which decouples the update mechanism of libraries from those of the applications that contain them, silent-update, and we find out that DCL is extremely useful to implement it since each application relying on an external library can simply fetch at run time the latest version of the source container, from the web domain of the library's developer, and, thus, execute the newest version of the library's code.

- **Memory footprint reduction**. Differently from static code, where all the classes are immediately compiled and stored in the phone memory, a positive effect of DCL is storing only code that needs to run in memory and, thus, contributing in minimizing the code footprint.

2.2.3 Malicious developers and security threats

Although we showed that DCL brings several advantages, unfortunately there are some security drawbacks as well. In particular, malware authors may use DCL techniques to attack mobile devices with two different strategies, fully described in [15] and resumed in the next two paragraphs.

Evasion of off-line detection systems An attacker may use DCL to execute a malicious payload from a statically benign application in different ways. For example, she may use it to bypass the Google Bouncer, the off-line detection system that analyzes all the incoming applications willing to be published on the Play Store. In this case, to avoid detection, the attacker needs to design the submitted application so that it does not contain malicious code statically, but rather it downloads and load the malicious payload at run time, after having been installed on users' devices. Notice that, in this scenario, it is impossible for an off-line system, like the Bouncer, to detect the malicious functionality in the app because the analyzed code does not constitute a threat at the moment of the analysis, and, moreover, the Bouncer cannot infer anything on the genuineness of the code that will be loaded at runtime. With the same technique, an attacker can also avoid detection from antivirus programs, since in Android they are

nothing more than regular applications, limited by the sandbox model, and thus, able only to compare signatures of the static APK against notorious malwares. For this reason, also antivirus have no chance to detect the loaded malicious code at run time.

Code injections against benign applications A different attack scenario involves the use of DCL into benign applications. More in the details, native Android API for DCL does not implement on the code loaded dynamically any check on integrity, nor on developer authentication. In particular, while a user downloading an application from the Play Store, can be sure that the installed APK was not tampered on the road thanks to code signing, the same does not hold for DCL. In fact, since the runtime environment (DVM or ART) does not perform any signature verification on the loaded code, it will execute any byte code provided in the code containers. Thus a remote attacker, who succeeds in modifying the bytecode of the original source APK or JAR container, will see her repackaged code executed by the runtime environment. This attack is particularly dangerous, if you recall that dynamically loaded code runs with the same permissions of the callee application and, so, in case of a successful attack, the malicious injected code will run with the same permissions and full access to the same data of the original benign application.

2.3 Threat model and problem statement

After having presented in the previous subsection the two attack scenarios for a malicious guy relying on DCL, from now on, we will restrict the analysis only on the latter presented case, in which developers of benign applications do not get it right and inadvertently introduce security bugs in their code using DCL.

In particular, in our threat model the attacker is a MITM able to exploit the fact that a benign application does not validate dynamically loaded code properly, so as to execute arbitrary code. This can happen in various ways, remotely or locally. To clarify which kind of attacks are allowed in our threat model, we now introduce a simple interaction diagram, represented in Figure 2.6 that summarizes the main steps that a benign developer has to face to perform DCL from a remote code container. Here it is the procedure:

1. The developer must fetch the code container from the remote location.
2. The developer has to save the fetched code archive on the mobile device's storage.
3. Because of the behavior of some API for DCL (e.g., `DexClassLoader` API), the developer has to initialize a folder, where the already loaded optimized version of the DEX files will be stored for caching. This helps to improve the API performance in successive load operations on the same source containers.
4. Finally, the developer needs to initialize the loading object from the native API. During the initialization phase, he will define the fetched archive as a source container for DCL and, later, he will use the object to perform DCL of the target classes.

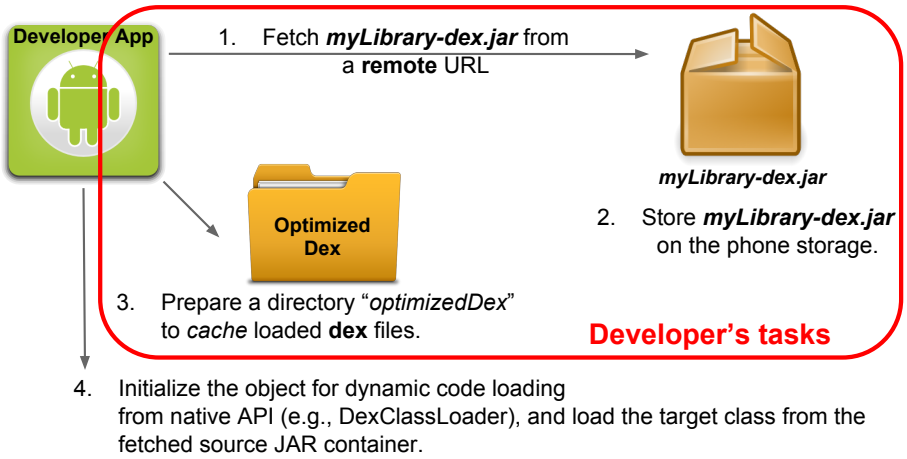


Figure 2.6: An interaction diagram that outlines the main steps that a developer has to follow to perform DCL from a remote code container. In particular, notice that steps 1, 2, and 3 must be implemented by the developer, even if native API could have possibly taken care of them in his place.

After having discussed the interaction diagram, we next highlight the three types of errors possibly introduced by benign developers that a MITM attacker can leverage to create an exploit. As a summary, Figure 2.7 presents small additions over the previous diagram by highlighting in red the errors introduced by developers and in green the corresponding best practices that fix them. In particular, the three conceptual mistakes are the following:

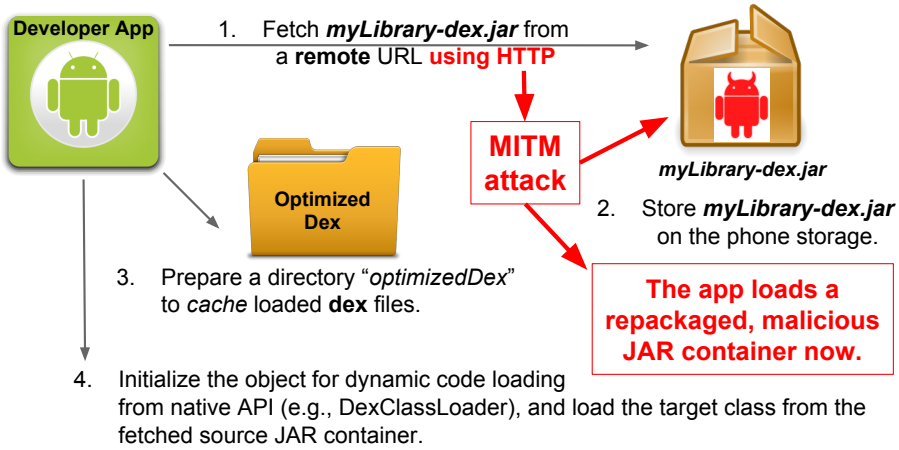
1. **Fail to fetch the remote code in a safe way**. No matter how simple this may sound but not using the secure HTTPS protocol, or using it incorrectly, may lead to an easily exploitable MITM vulnerability that an attacker can leverage to inject arbitrary code in the running device. A practical example is shown in Figure 2.7a, where the developer uses an insecure HTTP connection to fetch the remote code container and, because of this, the MITM attacker is able to substitute the original library file with a repackaged version containing an extra malicious payload.

2. **Fail to store code in a private location**. Developers could just not be aware of the file-system permissions in Android; for this reason, they may simply store the (securely) retrieved code in a world-readable, or worse, world-writable path. If it is not the developer's fault, the underlying, vendor-customized OS may just adopt unsafe default permissions, different from the best practices. In both cases, a local attacker (e.g., malicious application) may just exploit a race condition and overwrite the original code with a repackaged version that, when executed, will inject its malicious code at runtime. As an example, at first, consider Figure 2.7b that presents the case where a developer stores the securely fetched remote container into a world-writable location (e.g., external storage) and the attacker is able to overwrite it, and execute her code without raising any warning. Similarly, Figure 2.7c depicts another code injection attack,

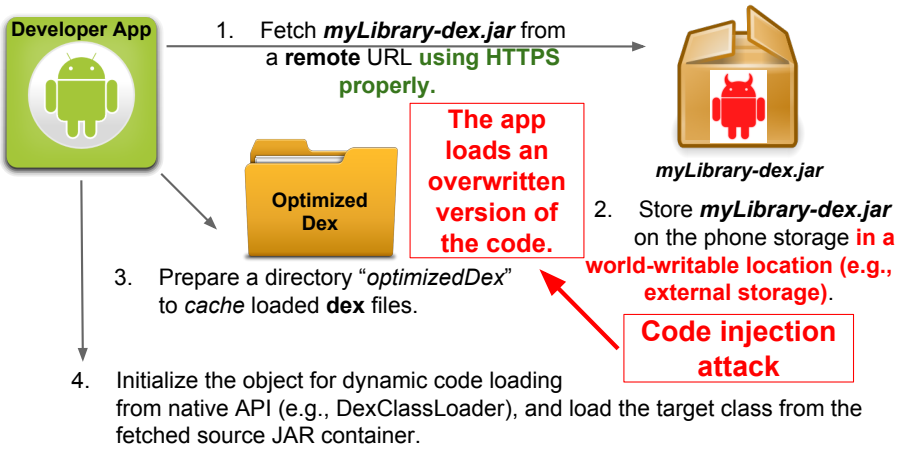
based on the same principle, where the developer fetched the source code in a safe way and he stores it in an appropriate application-private folder but, unfortunately, he carelessly initializes the optimized DEX folder in a world-writable location. The attacker can now simply copy, or possibly overwrite, a repackaged version of the optimized file and, at the next load on the same code container, the Android OS will cache the malicious optimized file and execute it.

3. **Miss or fail to implement security checks on the fetched code**. A developer who fails in implementing this operation cannot evaluate whether the fetched and stored code has been spoofed. This last error is the key point of the whole discussion: In fact, all the attacks presented in Figure 2.7 can be prevented by enforcing a proper policy based on code signing of the code container, prior to DCL. By adding this security requirement, the Android OS will allow the injection of additional code only if the source container successfully verifies security checks based on code signing. Still, by now, it remains unrevealed how this security verification should be carried out.

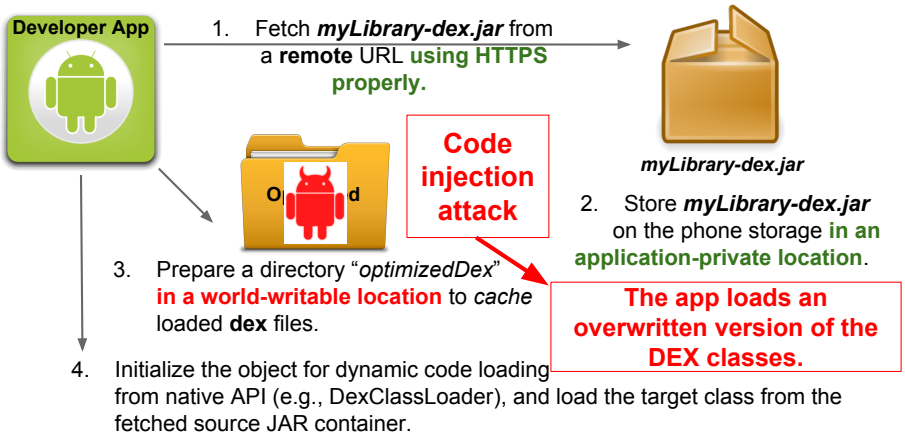
To summarize, the problem that we want to solve is designing a protocol, and later on an easy-to-use tool, based on the same protocol, that helps the benign developers in implementing remote DCL securely in their Android applications. The final system should fulfill this requirement under the threat model that we outlined at the beginning of this section, if not even under a lighter extension of it, which assumes that the attacker was also able to compromise the web domain holding the code container and, therefore, she can substitute the original code container with a repackaged version of it at her liking. To achieve the previous goal, we require our tool to perform its task without incurring into anyone of the security errors that we presented a while ago.



(a) Failure to fetch the source code securely: The developer uses an HTTP connection, instead of an HTTPS secure one.



(b) Failure to store the fetched code: The developer stored the fetched source container into a world writable folder (e.g., external storage).



(c) Failure to store the fetched code: The developer stored the optimized cached version of the code container into a world writable folder (e.g., external storage).

Figure 2.7: Comparison through interaction diagrams of the main errors that a developer may introduce in his code while performing DCL.

2.4 State of the art

After stating the problem that this work attempts to solve, we present in this section some of the related research solutions targeting the similar issues. In particular, we try to present the different alternatives by categorizing them according to the similarity of the issue that they claim to solve.

More in the details, Subsection 2.4.1 presents two solutions to the issue of verifying code prior to its execution on the mobile devices; whereas, Subsection 2.4.2 focuses on how to analyze and validate properly third-party libraries before an application employs them in its code.

2.4.1 Code verification

The problem of code verification prior to execution has been a challenging issue in several areas of computer science. For the Android OS, the first issue regards DCL since the Android runtime environment performs no code verification (i.e., cryptographic signature verification) on the implied source containers but, instead, it simply injects the found byte code. To solve this issue, Poeplau et all proposed in [15] a partial rewriting of the DVM, which, at their time of writing, was the unique runtime environment in Android OS. This provides the DVM with an extra verification mechanism, mandatory for all the applications, that grants the integrity of the code prior to its execution and mitigates all attacks resulting from the ability to load external code at runtime. To perform integrity checks, the proposed rewritten DVM relies on external verification providers, which are trusted elements that states whether a certain piece of code is allowed to run or not. Although the modifications of the DVM are quite cheap in terms of performance, applying them to the underlying Android framework is not; in particular, they would require forcing a system update for all the Android devices, which will certainly be a troublesome operation given the issue of fragmentation due to different OS versions, devices, and vendor-customizations.

A slightly different problem, faced in [18], is increasing the authentication properties of the Android markets whenever a new application is installed. In fact, differently from other systems, the Android OS suggests developers to generate and use self-signed certificates for signature verification. This design choice fell short under a security point of view as many malware authors tried in the past to distribute, also through the official market, repackaged version of popular applications containing an additional malicious payload. This attack is indeed extremely easy in the current Android scenario since there is no verification on the effective entity that release a certain certificate, and, therefore, an attacker could simply impersonate a popular company that provides applications, with the Android OS incapable of discriminating between the attacker's certificate and the real company's one. The solution that Vidas and Christin developed in their work was to design a verification protocol that alleviates this issue by creating a PKI-like infrastructure over DNS: As to verify the authenticity of their code, the developers must place the signing certificate on a DKIM or TXT record of their own domain name, which is required to match the reversed string extracted from the Java package name. With this trick, the authors ensures end-to-end integrity for applications; still, their approach makes no attempt to analyze the inner workings of an application or to protect the user from originally-malicious applications. Anyway, in our scenario, this solution

would not be sufficient because it offers stronger authentication properties only on application marketplaces and not also on the devices at runtime. For this reason, like other off-line detection techniques (e.g., the Google Bouncer), also this solution cannot counteract attacks relying on DCL.

2.4.2 Third-party libraries' security checks

Another research problem, which is quite close to the one stated before, regards library-centric threats and, in particular, assessing whether a third-party library, embedded into an application, is indeed benign and trustworthy or present some suspicious, or even malicious, behaviors. To overcome this problem, Hu et all presented in [12] Duet, a library integrity verification tool for Android applications that acts at application store level. Regarding its internals, this tool, at first, fetches copies of many libraries across the web, as distributed by the original library providers and reverse-engineer each one of them to reconstruct a set of Java byte code classes (.class). Once this process is completed, Duet computes, for each of these reverse-engineered libraries, both the digest of a merged file containing all the set of the .class classes and the digests for each single .class entry in the library. Finally, it stores all of these digests into a reference database. Whenever a new application containing third-party libraries is submitted to the market, Duet computes once again both types of digests on the incoming reverse-engineered library and, if matches are found in the database, then we are sure that indeed the incoming library was not tampered. Although useful, Duet has the limit of being, by design, a yes/no detection tool, which implies that, once there is no match in the digest, the tool will simply label the tested library as suspicious although the reasons behind this can be not malicious (e.g., the reference database may not contain an instance of the incoming library and, therefore, Duet will not find any matching digest during the analysis).

A different research work [11] by Grace et all. targets embedded ad libraries in the Android ecosystem, and in particular, it presented a system called AdRisk to identify potential risks in third-party libraries systematically. The idea, here, was to first decouple the embedded ad libraries from their host apps, and then analyze them through AdRisk so to detect statically any risk, ranging from uploading sensitive information to remote (ad) servers, till executing untrusted code from Internet sources. After having analyzed 100 representative in-app ad-libraries, this study claimed that many of these applications collects private user data and, although some of these data were clearly used for benign purposes, it was difficult to justify the collection of many others of them. The authors also pointed out that among these 100 representative libraries, five of them fetched and loaded code dynamically by relying on the use of `DexClassLoader` API. Although the API for DCL were detected, AdRisk, a static analysis tool, could not label this applications as certainly malicious. Proof of this is that, among these five applications, the authors were able to found, only after manual inspection, two samples that indeed fetched a malicious JAR container that, when injected at run time, would have turned the host into a bot listening to the commands coming from a remote controller.

2.5 Goals and challenges

Finally, after having introduced the threat model and the problem that this work aims to solve in Section 2.3, we summarize the requirements for the design of our system by presenting a list of three goals, along with the challenges to bear in mind for each one of those. Here is the list:

1. Our first goal is to design and implement a code-verification protocol suitable for DCL scenarios. While designing this protocol, we should consider that:
   (a) The verification protocol should be practical enough to be implementable as a drop-in, developer-friendly Java library that replaces the native API without requiring any code modifications beyond simple refactoring.
   (b) Our library should introduce negligible runtime overhead, and be able to work securely even when no Internet connectivity is available (i.e., should handle caching without any code or permission leaking).
   (c) Our library should handle three basic functionalities, namely retrieving, storing, and loading code, wrapped in one simple high-level function.
   (d) While designing the library, we need to balance usability, which would mean a high-level, and probably more secure API, with flexibility, which would mean exposing more freedom to the developer, resulting in a less secure API.
   (e) Finally, a main assumption is that the code loaded dynamically is benign, which is reasonable because it is developed and deployed by benign developers. As such, once landed on the client, the code will not try to escape our API at loading time (e.g., using reflection or other tricks typically adopted by malware developers).
2. The second goal is to help developers to use our library with little or no effort.
3. The third goal is to help application vendors and distributors, which do not need to be developers or security experts, to migrate existing applications effortlessly from the native API for DCL to the proposed one, without need for the source code or for writing even a single line of code.

Chapter 3

Grab'n Run: approach

In this chapter we present the approach designed to overcome the issues outlined in Section 2.3.

At the beginning of the chapter (Section 3.1), we provide an overview of the verification protocol for secure dynamic code loading by splitting it into logical phases that we map into a sequence diagram of a simple use case.

Each step is detailed in Section 3.2. In Section 3.3 we show how GNR solves by design all the errors presented in Section 2.3 and conclude with alternative approaches and their shortcomings (Section 3.4).

In Section 3.5, we outline the methodology used to design our repackaging tool to translate calls to the original API to calls to the GNR API, without the need for the application's source code.

We reserve details on the implementation of GNR library and the repackaging tool for the next chapter.

3.1 Approach overview

We focus our approach on remote DCL (i.e., code fetched from a remote server). Local DCL is actually a simplification where the man in the middle is on the device.

In essence, our approach is based on the idea of running a cryptographic code-verification protocol before executing every dynamically loaded code container (e.g., JAR, APK, DEX class files). The verification protocol follows five steps:

- **Step 1: Code retrieval.** Fetch remote code containers via HTTP/HTTPS. Both protocols are acceptable since the protocol verifies the code out of band. For local containers, we do not need this step since code is already on the device.

- **Step 2: Code storing.** Store the retrieved remote containers in a folder accessible (read and write) only by the application that performs DCL; local containers are imported as well in the same application-private folder. This step is not only a security requirement but it also reduces the performance overhead of the whole system since, in case of successive load

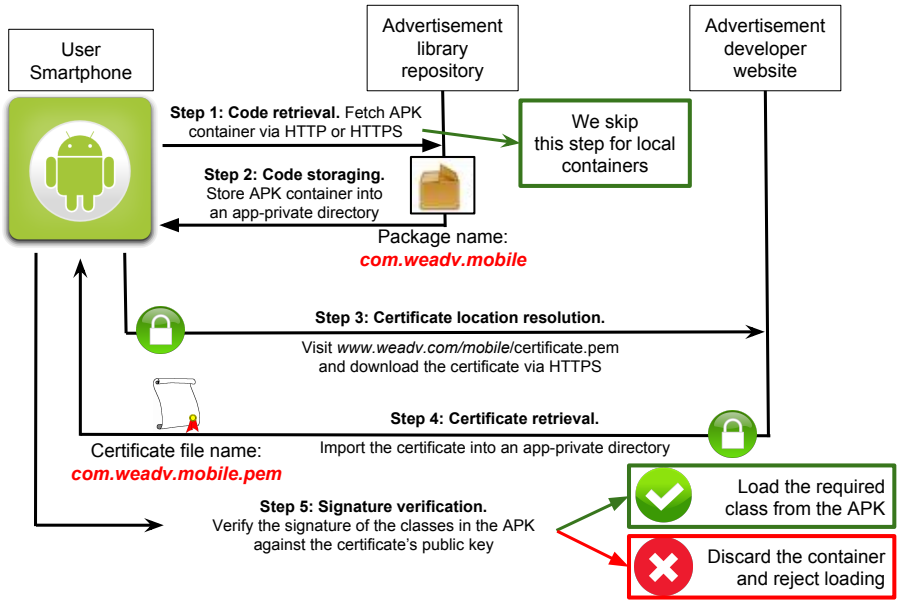


Figure 3.1: Sequence diagram of a simple use case of remote DCL. The diagram shows interactions between the user application and the advertisement library servers. All the steps of our verification protocol are highlighted. All lines marked with a green lock-pad must be secure interactions (e.g, HTTPS encrypted connections).

operations, it prevents to fetch repeatedly the same remote code containers.

- **Step 3: Certificate location resolution.** We need to know where to retrieve the certificate from. We build the location either by means of a naming convention or of configuration information provided by the developer.
- **Step 4: Certificate retrieval.** Fetch remote certificate at the constructed URL via HTTPS. Here, differently from **Step 1**, it is fundamental that the connection is guarantees integrity and sender's authenticity, so that the attacker cannot tamper with the certificate.
- **Step 5: Signature verification.** All the entries inside the retrieved code container are verified against the certificate. If at least one of them does not pass the signature verification check, the container is discarded.

Figure 3.1 provides a self-explanatory use case of the verification protocol, in which a developer wants to dynamically load an advertisement library.

3.2 Verification protocol details

In this section, we dissect and examine in details each step of our approach.

3.2.1 Step 1: Code retrieval

During this first step, the developer needs to provide the location of one or more containers. For all of those, which are not directly accessible on the phone storage, the device must fetch them from the web via HTTP or HTTPS. The protocol used to download the container is not relevant since, later in the process, we plan to verify both integrity and signer authentication on any piece of code prior to its loading. We think that this choice is a good trade-off because, although keeping the system secure against MITM attacks, it does not force library developers to serve all their resources to HTTPS, with obvious advantages in terms of performance since SSL is not necessarily required for code retrieval.

3.2.2 Step 2: Code storing

The next step is storing all the candidate code containers into an application-private directory. In our protocol, forcing containers' storage in an app-private folder is essential to prevent a local attacker (e.g., malicious application) from overwriting the retrieved containers after a successful security verification process. In fact, if the container would have been stored in a world-writable location (e.g., external storage), a local attacker could overwrite the code container. This requirement allows us to claim that, once a container is evaluated positively, it can be always considered a trustworthy element since only the running application can modify it but, in our threat model, we assume the application to be benign, so trustworthy, thus not interested in tampering the container.

To lower times that the system has to fetch a remote container or to import a local one, we set up a caching strategy. More in the details, whenever the system imports a new code container into the application-private folder, it renames the container as digest of the file plus the file extension (".jar" or ".apk"). This is an easy solution to disambiguate each container and avoid naming conflicts. For the local case, before importing a container, the system calculates its digest, checks whether there is a matching file in the app folder according to the introduced naming convention, and, if so, caches this match instead of importing the external container. On the other hand, for the remote case, the system needs a strategy to understand whether a remote container has been already stored without fetching it. One suitable solution is to use a table structure with three fields per entry: (1) the URL at which the remote container is located, (2) the name of the corresponding local container under our convention, and (3) a time-stamp that indicates when the container was fetched. Thanks to this table, every time that a remote container should be fetched, the system checks instead whether a container associated with the input remote URL is present in the table. If so, it retrieves the name of the linked local container and it checks whether this cached copy actually exists in the folder. If this is the case, the system verifies the container's freshness by means of the time-stamp in the table and, if the outcome is positive, it picks that container. On the contrary, whenever any of the previous checks fail, the system attempts to fetch the remote container and, in case of success, it adds an entry to the table for the newly fetched container.

Input package name	Corresponding remote certificate's URL
`it`	Invalid package name thus no valid URL.
`it.polimi`	https://polimi.it/certificate.pem
`it.polimi.necst.mylibrary`	https://polimi.it/necst/mylibrary/certificate.pem

Table 3.1: Construction of certificate URL from package name. The table lists the rules used to reverse a container package name into a valid URL location. The final suffix is "certificate.pem" by default. Package names must have at least two not-empty, dot-separated subfields to be considered meaningful, otherwise no link will be produced. The first two subfields are reverted and used as domain name; whereas the successive terms are just appended to reconstruct the folder structure of the URL. We enforce all the constructed URL to use the HTTPS protocol.

3.2.3 Step 3: Certificate location resolution

The next step is retrieving the certificate of the developer that signed and published that code. Our approach provides two alternative ways to obtain the remote location of the certificate: (1) by constructing a URL by reversing the package name of the target class to load; (2) a configuration map that developers fill in the application's source code. This object associates each package name to the remote URL of the certificate that the protocol must use to validate all the classes, and therefore all the containers, that share that package name.

For the first method, Table 3.1 summarizes the rules and gives examples on how we construct remote certificate's URL starting from a package name. While this method is extremely simple, it does not grant enough flexibility since it requires the developer to satisfy tight constraints on web domain names. That is why we decided to provide a second solution, which requires little extra code but greatly improves flexibility for the developer. This latter solution is acceptable under a security point of view because, in our threat model, the MITM attacker is not able to modify the code of the running application in memory therefore she cannot even change the association between package names and certificates in the developer-provided map. Table 3.3 presents an example of such a map, whereas Table 3.4 shows how the different containers, presented in Table 3.2, are linked with the appropriate certificate's location according to the configuration map.

3.2.4 Step 4: Certificate retrieval

For this step many of the considerations made in **Step 1** and **Step 2** still hold. However, there is a strong difference: here it is necessary that the remote certificate is fetched through an encrypted and authenticated connection (e.g., via HTTPS). This is a single point of failure in our model since, as soon as the remote certificate is not correctly fetched (e.g., an attacker exploits a vulnerability in the SSL protocol to substitute the legitimate certificate with a different one), the attacker can easily compromise the security of our protocol. In different words, we may say that remote certificates are the trusted elements of our model.

Once the remote certificate is fetched, our approach requires to store it into

Source code container for DCL	Corresponding package names
Container1.apk	`com.example.cont1`
Container2.jar	`it.polimi` `it.polimi.net`
Container3.apk	`com.example.cont3`

Table 3.2: Example of containers and their package names. Differently from APK, which has a unique package name, JAR containers may have several of them.

Package name	Corresponding remote certificate's URL
`com.example.cont1`	https://polimi.it/certificate1.pem
`it.polimi`	https://polimi.it/certificate2.pem
`it.polimi.net`	https://polimi.it/certificate2.pem

Table 3.3: Example of a possible configuration map that a developer can use to indicate the location of remote certificates. Each entry connects one package name with the corresponding URL of the certificate that the system must use to validate a container including that package name.

Source code container for DCL	Corresponding remote certificate's URL
Container1.apk	https://polimi.it/certificate1.pem
Container2.jar	https://polimi.it/certificate2.pem
Container3.apk	No certificate for the verification since the developer did not link any of them to `com.example.cont3`

Table 3.4: Binding between containers and certificates. This table shows how the code containers, presented in Table 3.2, are linked with the corresponding certificate for the verification according to the settings provided in the configuration map of Table 3.3. In particular, the last container has no binded certificate because there is no entry in the map matching its package name to a certificate's location.

an application-private folder to prevent an attacker from tampering or overwriting it. During the process, we rename certificates with the corresponding package name associated with them in **Step 3**. For this reason, whenever a new remote certificate is required, at first our system looks for a local certificate in the folder named with the corresponding package name and, if it finds one, it caches that copy; otherwise it fetches and import the remote certificate. One may argue that a package name is not the best way to identify a certificate but, in this particular scenario, it can be an acceptable choice for three facts: (1) Google suggests developers to generate key pairs, and thus certificates, that last for a long period of time (e.g. 25 years) [16] and, therefore, an application changing its certificate is an unlikely event; (2) since developers are identified with their certificate, once more, they are forced to keep the same certificate for as much time as possible; (3) also package name must always remain untouched across different versions of the same application. To sum up, both certificates and package names are elements that tend to remain constant across different versions of benign applications and that is why our system can reasonably link them.

3.2.5 Step 5: Signature verification

The input of this step is a container, which carries those classes that the developer wants to load dynamically, and a certificate, which is we use to verify the genuineness of the container. The output of this step is a yes/no answer indicating whether the code can be loaded (i.e., genuine) or not. The answer is based on two aspects, integrity and authentication. Firstly, we consider the integrity of all the entries of the container, which means that we will not accept to load code from a container, in which there is one (or more) entry that does not verify its signature. Secondly, we authenticate each entry of the container to verify that it is actually signed by the developer associated with the trusted certificate.

Algorithm 1 presents in pseudo-code how the signature verification process works. The idea is that at first, we look for the manifest of the container. The manifest of a Java-based archive contains all the signatures of the relevant entries in the archive. A missing manifest or a non-matching is a necessary (yet not sufficient) condition to discard the archive up front. For the reminder archives, we verify the corresponding signature, stored in the container's manifest, against the public key, stored in the trusted certificate. A non-matching signature happens in two cases: (1) the file entry in the archive has been altered after the initial signing process; (2) the file entry has been signed or resigned with a private key different from the one coupled with the public key in the trusted certificate. In both cases, the algorithm rejects the container. Note that the verification can fail for non-intentional causes. The benign library developer, for example, may forget to sign the container or to resign it after a modification of one of the entries; or, then again, the app developer may have coupled the wrong trusted certificate with the code container in the app's sources. Anyway, our protocol assumes that it is library developer's responsibility to sign his code prior to publication and app developer's task to pair correctly code containers and certificates. We think these requirements are not troublesome to accomplish since: (1) they can be easily automatized and integrated into development's tools; (2) app developers are used to sign their not dynamic code.

Moreover, our system introduces a caching strategy to benefit from the results of previous signature verifications, so to verify each container only once, independently from the number of load operations performed on it. To achieve this goal, the system relies on package name to discriminate which classes can be immediately loaded or rejected without an extra signature verification. In particular, whenever the system verifies a container, it propagates the result of this verification to all the package names contained in it. Thus, when a new load operation involves an already evaluated package name, the system can simply refer to the previous outcome of the verification, instead of performing a new one.

3.2.6 Code and certificate binding

After having analyzed the steps of our verification protocol, we discuss how our system binds containers, certificates, and classes.

Containers and certificates In **Step 5** the binding between containers and certificates is constructed through the package name, which is an identifier for

```
input  : An APK/JAR container cont, a trusted certificate cert
output: True/False on whether cont verifies signature against cert

if cont has no Manifest then
    // The container is not signed at all.
    return False;
end

pk ←extractPublicKeyFromCert(cert);

foreach file entry fe in cont do
    if fe is in the Manifest then se ←getSignatureForAnEntry(fe);
    else
        // This entry is not signed.
        return False;
    end
    valid ←verifySignatureAgainstTrustedPK(fe,se,pk);
    if ! valid then
        // Entry fe was altered or not signed by the expected trusted
           private key.
        return False;
    end
end

// Reaching this statement implies that all the entries passed the
   signature verification.
return True;
```

Algorithm 1: Signature verification of a container against a trusted certificate.

APK containers. Indeed, once an app is released on the Play Store, the policy forces the developer to keep the same package name across different versions. JAR containers, differently from APK, can have many package names (potentially their number can be equal to the amount of classes in the JAR) thus a developer, who wants to load many classes from the same JAR, could possibly need to fill the configuration map with many package names all pointing to the same certificate. To solve this issue, we decide to extend the current definition of package name by introducing the concept of "root package name". This is the shortest but still significant package name, which is a common prefix for the highest number of package names in the container. In particular, we consider a package name significant if and only if it is composed by two or more not-empty words separated by single dots. Table 3.5 presents an example of the JAR corner case that helps to understand how useful is the addition of root package name for both our protocol and for the application developers, who will have to fill in the configuration map with far fewer entries. According to this new principle and to make developers' job easier, we will ask them, while using our final library, to link root package names with remote certificate's URL.

Containers and classes Also for containers and classes we define a binding through the package name. In this case, we did not actually make a choice but we inherited this constraint from Android `ClassLoader` API for DCL because many of them (e.g., `DexClassLoader`, `PathClassLoader`) only provide the final user with the possibility to state which class to load but not in which container to look for that class. Since one of our goals is designing a library, whose API are as close as possible to the Android native ones, and since we want to avoid a

Classes in the JAR	Entry without root package name	Entry with root package name
it.polimi.ClassA	it.polimi	it.polimi
it.polimi.test.ClassB	it.polimi.test	it.polimi
it.polimi.test.one.ClassC	it.polimi.test.one	it.polimi
it.polimi.settings.ClassD	it.polimi.settings	it.polimi
it.polimi.main.ClassE	it.polimi.main	it.polimi

Table 3.5: JAR corner case and root package name. We consider a JAR archive containing five classes with five different package names. As shown in the table, without the root package name, a developer has to populate the configuration map with five different entries, each one pointing to the same remote certificate; whereas, with this concept, the number of entries in the map is reduced to one (i.e., the root package name `it.polimi` summarizes all the classes in the JAR).

full scan (and signature verification) on all the containers per load operation, we decide to maintain an associative map to link package names to their containers. With this simple addition, whenever a user attempts to load a class, our system extracts its package name from the full class name, retrieves in one shoot the possibly linked container, and finally it performs a signature verification of this container against the certificate associated to the same package name (or too its root package name when no certificate is directly associated with it). Figure 3.2 shows a summarizing example on how, starting from a class to load, our library should extract the package name and use it to retrieve both the associated container and the trusted certificate.

3.3 Approach validation

In Section 2.3, we presented three main errors that a developer can introduce while using DCL with native API. In this subsection, we discuss how our remote verification protocol solves and prevents each one of these three issues. More in the details:

- The first error was fetching the code to load in an unsafe way (e.g., make use of HTTP connection). Our protocol fetches code containers coming from both HTTP and HTTPS connections. However, independently from the chosen transport protocol, the system performs integrity verification on any piece of code, before loading it dynamically. Therefore, even if an attacker is able to replace the original container with a malicious one by exploiting an unsafe HTTP connection, our protocol detects and rejects the repackaged container, thus preventing the malicious code from being loaded.

- The second error was storing the fetched code in a world-writable location on the device (e.g., external storage). We design our system to store any fetched remote resource (i.e., container, certificate) into an application-private folder (modifiable only by the running application, which we assume trustworthy). The protocol forces also the import of any local container into the same private folder since these archives may have been saved into a writable area, accessible by the attacker. To sum up, the

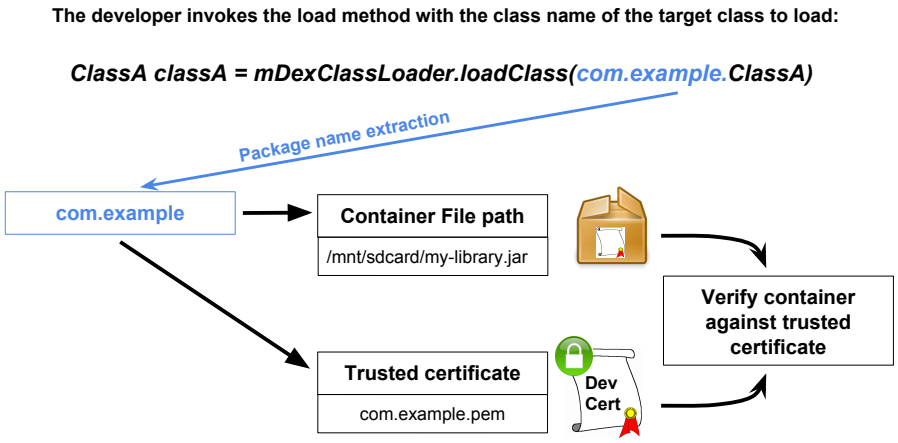


Figure 3.2: Bind containers and certificates via package name. Whenever the app developer tries to load a class, our library should extract the corresponding package name and use it to discriminate which container may have an implementation of the target class and which certificate to use for verifying the signature of the container.

protocol always saves any implied resource into an app-private read and write area of memory.

- The third error was missing or failing to implement integrity checks to verify that the fetched and stored code has not been tampered, before loading it. Our protocol requires a successful signature verification on code containers against a trusted certificate provided by the developer before allowing loading classes dynamically from them. Thanks to code signing, our system grants both that a container has not been tampered or repackaged, after having been signed, and that the signing process was performed by the owner of the public key stored in the trusted certificate. A successful result on both of these checks is a sufficient evidence to allow DCL from a container.

In conclusion our protocol secures (by design) the DCL of Android under the outlined threat model and it can detect at runtime whether a source code container passes the signature verification check and load classes accordingly.

3.4 Alternative approaches

After having presented our approach, we discuss three alternative solutions and highlight their shortcomings.

3.4.1 Use certificate in the container for verification

One may think to use directly the certificate inside of the code container instead of fetching an external trusted one. However, this is extremely insecure under

our threat model. In fact, we cannot trust the certificate inside the container as the only element for the verification because, obviously, if we need the system to be able to retrieve the container via HTTP and an attacker can substitute the container with a repackaged version, she may also easily resigns this container with a private key under her control before performing the substitution; in this case, the repackaged container would result as perfectly legitimate since it would successfully verify the signature against the certificate inside of it. Instead, besides integrity of the container against its own certificates, our verification protocol requires also that the trusted certificate provided by the developer is among those ones used for the signature verification. Thanks to this measure, we can grant that an executed code container is not only undamaged (i.e., not tampered) but also approved by the legitimate developer.

3.4.2 Use a digest in place of a trusted certificate

Another alternative is the use of a securely-retrieved digest, instead of a trusted certificate, to verify container's integrity. This method does not fully meet the requirements of the final system, specifically in allowing silent updating strategies in the easiest way for developers. Once the certificate is cached, the verification protocol can use it to validate all the containers coming from a certain developer, independently from the container's version (this is a reasonable assumption because it is likely that a developer signs all the versions of his applications with the same private key). Instead, using a digest computed on the container requires that, every time that a new version of a container is released, a new digest must be computed, published, and stored securely. This is bothersome for the developer, who must remember to make available a new digest via an HTTPS URL every time that he updates his library. In simple words, using a digest makes the system too strict, with respect to updates, with no additional security benefits.

3.4.3 Alternative binding between containers and certificates

In Section 3.2.3, we proposed the use of a configuration map for a flexible binding between code containers and the required certificate for their verification. While designing the protocol, we evaluated an alternative approach based on storing the list of the certificates for the verification into an XML custom tag, or set of tags, into the application manifest. In the end, we opt for the map because, although being a simpler solution, it also offers enough flexibility to the developer, who can customize it with a fine granularity (i.e., one-to-one mapping between each package name and a certificate).

After this choice, a linked problem was deciding which attribute to use as a key in this map. We evaluated several possibilities:

- **Absolute container file path**. This is a naïve solution since file paths are usually long strings and, particularly in our scenario, where all the code containers are stored in the same app-private folder, it is an enormous waste of memory.

- **Container file name**. Although preventing the memory's waste, this is

a bad choice since it is extremely easy to hit naming conflicts, especially when the system fetches many code containers from the web.

- **Hash of the container file**. This idea solves naming conflicts since it is unlikely that different containers hash to the same exact value. However, it does not scale with the release of new versions of the same library. In fact, every time that a new version of the container is published, the hash changes as well; this would force the application developer to insert a different key (hash) in the map per new release thus it would make the system impractical for supporting silent updates of third-party libraries.
- **Package name of the container**. In the end, we selected this attribute as the key since it is a valid identifier for every APK and it remains constant across the different version of an application. Thus, this solution is the most convenient to handle silent updates and to overcome naming conflicts. The only remaining issue is represented by JAR containers but, as explained in Subsection 3.2.6, we were able to overcome this problem thanks to the introduction of root package names.

3.5 Migrating existing code to Grab'n Run

In this section, we present the conceptual details of the repackaging tool to port an Android application to use GNR API.

The idea is modifying as little as possible of the original application by patching only those sensitive points of the archive, which directly make use of DCL methods. With this premise, the goal is, given a non-obfuscated working application using native API for DCL and some settings provided by the original app developer, to obtain a DCL-secured version of the input application, which is still full-working but makes use of GNR API. Here we summarize the steps of the approach:

1. The tool performs static analysis on the input APK to retrieve whether this container needs to be patched and, in case, which further extra permissions (i.e., `android.permission.ACCESS_NETWORK_STATE`, `android.permission.INTERNET`, and `android.permission.READ_EXTERNAL_STORAGE`), required by GNR API, should be added to its manifest.

2. Then, the tool disassembles the original APK container. It manipulates the manifest file for adding extra permissions and parses the intermediate representation of the Dalvik bytecode.

3. Next, the tool identifies the sensitive points, where the native API for DCL is used.

4. The tool substitutes each one of the sensitive points with an equivalent method call, or snippet of code, which makes use of the functions from the GNR API. The patching of the sensitive points differs according to the developer-provided settings.

5. Finally, the tool reassembles the patched version of the original container to reconstruct a runnable and fully-working application.

Chapter 4

Implementation details

In this chapter we describe the implementation details of the Grab'n Run library and the repackaging tool, which are the two implementations of the approach outlined in Chapter 3.

At first (Section 4.1), we present open-source GNR library, which implements our remote verification protocol, acting as a wrapper of `DexClassLoader` API and extends it both in security and functionality. More in details, in Subsection 4.1.1 we provide an overview of GNR architecture and on the use of its API. Later, we examine the main components of the library (Subsection 4.1.2 - Subsection 4.1.6) with particular attention on the two main classes, `SecureLoaderFactory` and `SecureDexClassLoader`.

Next, in Section 4.2 we explain in great details how we implement the signature verification algorithm used by `SecureDexClassLoader` to evaluate whether a class should be loaded.

In Section 4.3 we introduce the details on the implementation of the repackaging tool that we realized to ease migration from the original, insecure API and reduce boilerplate code.

4.1 Grab'n Run: Library implementation

4.1.1 Overview and example usage

Table 4.1 maps each step of our verification protocol, as presented in Section 3.1, to the corresponding classes of the GNR library that execute them. Differently, Figure 4.1 shows a summarizing UML diagram that reports the relevant classes of the current implementation.

The most important classes are `SecureLoaderFactory`, a factory class that initializes secure loading components, and `SecureDexClassLoader`, which wraps `DexClassLoader` and exposes a backward-compatible yet secure code-loading API. We can logically map the main functionalities of `SecureLoaderFactory` to the ones of the constructor of `DexClassLoader` and the `loadClass()` method of the former to the corresponding one in the latter.

Now, we present a resume on how, and in which order, these two classes help the developer obtaining a working snippet for performing DCL.

1. **Initialize** `SecureLoaderFactory`. At first, the developer initializes an

Step of the verification protocol	GNR classes performing the step
Step 1: Code retrieval	`CacheBinder` (`SecureLoaderFactory`) `FileDownloader` (`SecureLoaderFactory`)
Step 2: Code storing	`SecureLoaderFactory`
Step 3: Certificate location resolution	`SecureLoaderFactory`
Step 4: Certificate retrieval	`CacheBinder` (`SecureDexClassLoader`) `FileDownloader` (`SecureDexClassLoader`)
Step 5: Signature verification	`PackageNameTrie` (`SecureDexClassLoader`) `SecureDexClassLoader`

Table 4.1: Mapping between the verification protocol's steps and GNR classes. This table links each step of the verification protocol described in Section 3.1 with one (or more) classes that performs that step in GNR library. Classes in brackets embed the classes that actually perform the step.

instance of `SecureLoaderFactory`, which requires a reference to a running `Activity` object.

2. **Initialize** `SecureDexClassLoader`. Calling the method `createDexClassLoader()` returns a `SecureDexClassLoader` instance. Alongside the usual parameters required by `DexClassLoader`'s constructor, the developer must pass an associative map that links package names to the URL of the remote certificate to verify code signature. We explained the reason for such a map in Subsection 3.2.3.

3. **Load code dynamically**. Next, the developer uses the `loadClass()` method on `SecureDexClassLoader` to load the class specified with its full name. `SecureDexClassLoader` returns a class object if the implementation of the class is inside a successfully verified JAR or APK code container.

4. (Optional) **Wipe out cached resources**. At the end of the process, since GNR caches code containers and certificates to increase both performance and success rate in the loading procedure, a developer may desire to delete any or all the cached resources by using the `wipeOutPrivateAppCachedData()` method on the `SecureDexClassLoader` object.

Listing 4.1 compares the implementation of the same functionality with GNR vs original API `DexClassLoader` (Listing 2.1). The goal of the code snippet is loading an instance of `com.example.MyClass` from the code container at a remote URL saved in the variable `jarContainerPath`. The snippet shows how to properly handle the returned value of the `loadClass()` method since a `null` reference is returned in case of a security constraint's violation during the loading process. We list possible return values and the security constraints in Subsection 4.2.1. Although the developer has to handle only two classes for performing DCL, the interactions among the internal components are more complex: Figure 4.2 presents a sequence diagram showing the internal calls between the five main classes of GNR, starting from the initialization till a class is dynamically loaded.

The next subsections present all the main components of the library by providing an overview on their purpose in the global system, their main methods, and relevant implementation details.

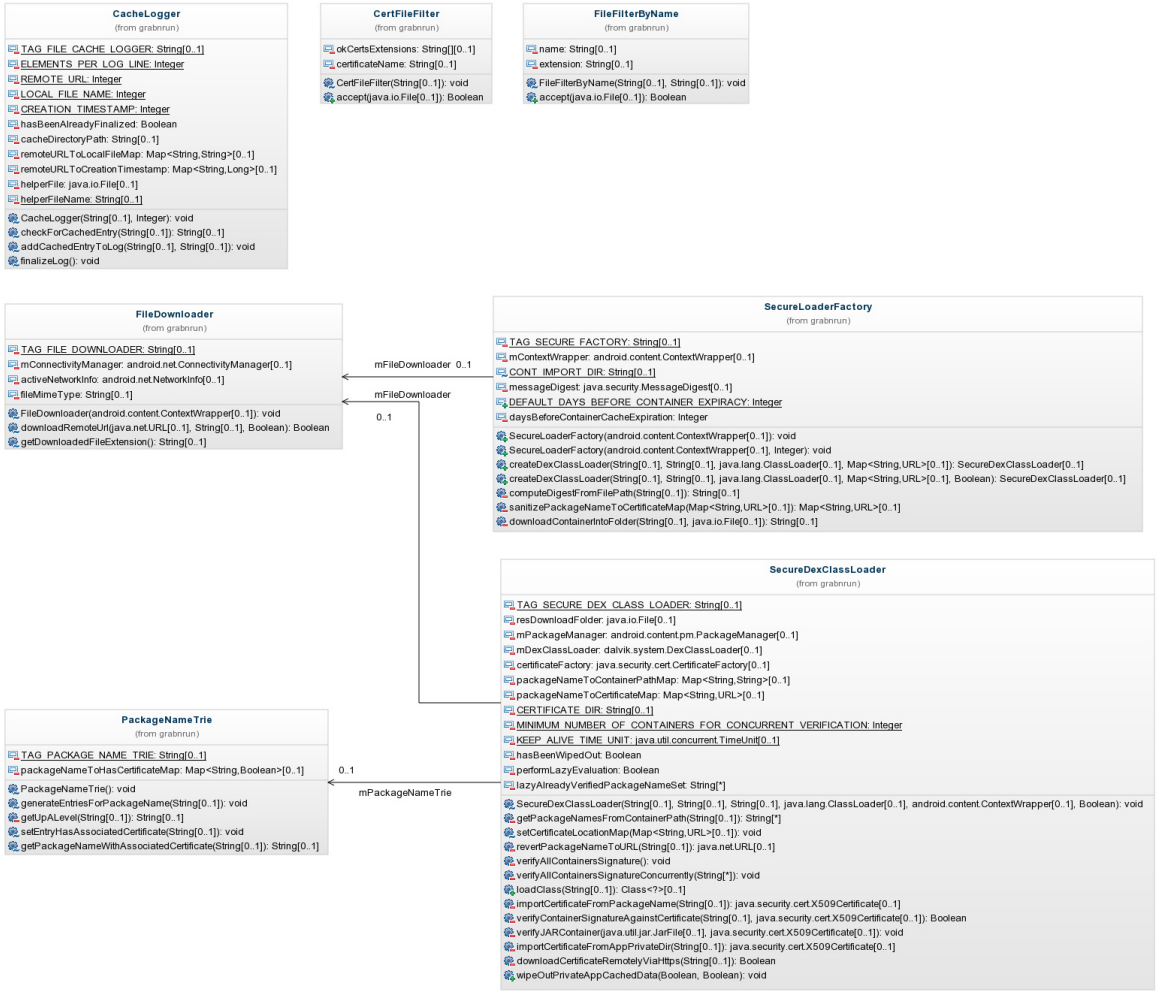


Figure 4.1: GNR UML class diagram. A representation of the main classes that compose GNR library. A final user only instantiates `SecureLoaderFactory` and `SecureDexClassLoader` objects; whereas the remaining classes are internal helper ones. This diagram was obtained by reverse-engineering GNR project with a tool, named GenMyModel.

Listing 4.1: GNR example snippet.

```
MyClass myClassInstance = null;
jarContainerPath = "http://something.com/dev/exampleJar.jar";

try {
   Map<String, URL> packageNamesToCertMap = new HashMap<String, URL>();
   packageNamesToCertMap.put( "com.example",
                              new URL("https://something.com/example_cert.pem"));

   SecureLoaderFactory mSecureLoaderFactory = new SecureLoaderFactory(this);
   SecureDexClassLoader mSecureDexClassLoader =
   mSecureLoaderFactory.createDexClassLoader( jarContainerPath,
                                              null,
                                              getClassLoader(),
                                              packageNamesToCertMap);

   Class<?> loadedClass = mSecureDexClassLoader.loadClass("com.example.MyClass");

   // Check whether the signature verification process succeeds..
   if (loadedClass != null) {

      // Here class loading was successful, and performed in a safe way.
      myClassInstance = (MyClass) loadedClass.newInstance();

      // Do something with the loaded object myClassInstance
      // e.g. myClassInstance.doSomething();
   }

} catch (ClassNotFoundException e) {
   // This exception will be raised when the container of the target
   // class is genuine but its implementation is missing in the
   // source archive..
   e.printStackTrace();
} catch (InstantiationException e) {
   e.printStackTrace();
} catch (IllegalAccessException e) {
   e.printStackTrace();
} catch (MalformedURLException e) {
   // The previous URL used for the packageNamesToCertMap entry
   // was a malformed one.
   Log.e("Error", "A malformed URL was provided for a remote certificate location");
}
```

4.1.2 SecureLoaderFactory

`SecureLoaderFactory` is a responsible of initializing the components that handle DCL. As regards the remote verification protocol, which we outlined in Section 3.1, this element covers roughly **Step 1** to **3**. At implementation level, `SecureLoaderFactory` provides two methods: a constructor and a generator of `SecureDexClassLoader` objects.

Constructor The invocation of this constructor lets `SecureLoaderFactory` gain a hook to the complete mobile phone state (e.g., network, storage, and file-system access) for later operations. This is accomplished by requiring a `Context` object (e.g., the `Context` inside of an `Activity` class, or even an `Activity` itself) as a mandatory parameter. Developers may also provide an extra integer parameter to set up the maximum time span, in days, for which `SecureLoaderFactory` will consider a local copy of a remote code container fresh enough to be cached. Therefore, whenever a container exceeds this time interval, instead of caching it, `SecureLoaderFactory` discards it and retrieves a fresh copy of the corresponding remote resource.

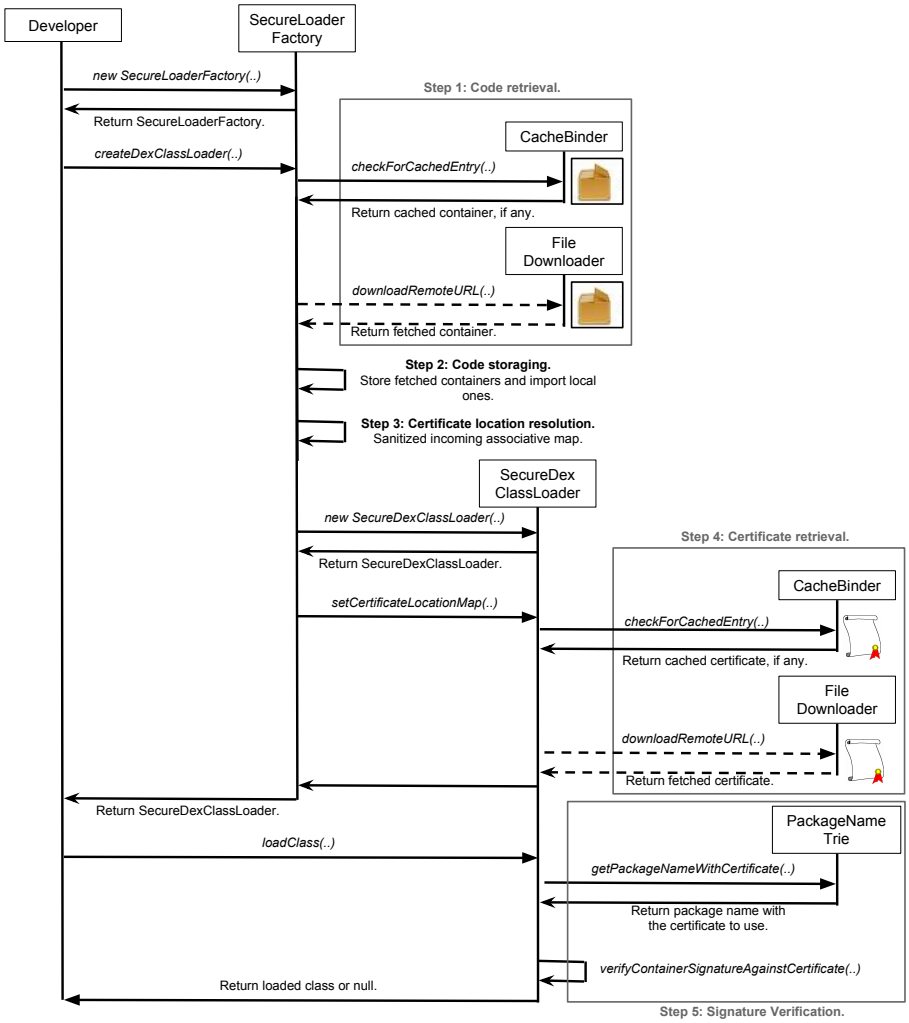


Figure 4.2: Sequence diagram at implementation level. This diagram shows the interactions between the five main components of GNR so to perform a DCL operation. The methods invoked by the components are typed in italics. Dashed lines represent interactions that may not happen (i.e., it is not always necessary to fetch a remote resource like a certificate). In the diagram we also point out how the interactions are linked with the steps of our verification protocol.

Listing 4.2: Signature of `SecureLoaderFactory`'s constructor.

```
public SecureLoaderFactory(   ContextWrapper parentContextWrapper,
                              int daysBeforeContainerCacheExpiration);
```

Create a SecureDexClassLoader instance The second relevant method of this class is `createDexClassLoader()`. Its signature is close to the class constructor of native `DexClassLoader`, except for three differences: (1) this method leaves out the `optimizedDirectory` parameter, which points to the folder used to store ODEX or ELF files during the loading process; (2) it requires a reference to the associative map linking package names to certificates' remote location; (3) it returns a `SecureDexClassLoader` object instead of a `DexClassLoader` one.

Listing 4.3: Signature of `SecureLoaderFactory`'s `createDexClassLoader()`.

```
public SecureDexClassLoader createDexClassLoader(
                                    String dexPath,
                                    String libraryPath,
                                    ClassLoader parent,
                                    Map<String, URL> packageNameToCertificateMap,
                                    boolean performLazyEvaluation);
```

Focusing on the internals of the implementation, the first step is parsing the `dexPath` string to locate the paths of the potential code containers for class loading. Differently from `DexClassLoader`, which is able to handle only containers already saved on the local device storage, our class can manage also paths pointing to remote code containers.

In fact, whenever `SecureLoaderFactory` finds a remote URL, it queries an internal component, named `CacheBinder` and described in the next subsection, to understand whether a fresh copy of the corresponding remote resource has been already stored into a previously reserved application-private folder on the device. If this is the case, the URL of the local copy is selected instead of the remote one; otherwise, `SecureLoaderFactory` delegates another component of GNR library, `FileDownloader` (see Subsection 4.1.5), to fetch the remote container. Once the download is complete, `SecureLoaderFactory` stores the fetched code container in the same app-private folder, after having renamed it accordingly to the naming convention presented in Subsection 3.2.2. Similarly to the remote case, when `SecureLoaderFactory` parses a path pointing to a local archive, it checks whether this resource has been already imported in the application-private folder. If so, it caches the internal copy; otherwise, it imports and renames the local archive with the usual naming convention and then its patches the corresponding `dexPath` segment with the new correct file path of the resource.

Once `SecureLoaderFactory` finishes to analyze all the containers' paths, it sets up another app-private folder that will be used for storing cached ODEX or ELF files generated by successful load operations in `SecureDexClassLoader`.

```
File dexOutputDir = mContextWrapper.getDir("dex_classes", ContextWrapper.MODE_PRIVATE);
```

Next, `SecureLoaderFactory` performs an extra sanity check on the associative map, provided by the developer, on both its keys (i.e., package names) and its values (i.e., remote certificates' URL). In particular, `SecureLoaderFactory`

evaluates both the validity of the package names (i.e., a series of two or more non-empty words, split by dots) and of the corresponding certificate (i.e., not null reference, must use HTTPS protocol otherwise `SecureLoaderFactory` will enforce it).

```
// Sanitize fields in packageNameToCertificateMap:
// - Check the packages names (only not empty strings divided by single separator char)
// - Enforce all the certificates URL in the map can be parsed and use HTTPS
Map<String, URL> santiziedPackageNameToCertificateMap =
     sanitizePackageNameToCertificateMap(packageNameToCertificateMap);
```

After that `SecureLoaderFactory` has taken care of removing invalid entries from the associative map, it generates a `SecureDexClassLoader` instance by invoking its class constructor with the previously validated parameters and, finally, it returns the resulting object to the caller.

```
// Initialize SecureDexClassLoader instance
SecureDexClassLoader mSecureDexClassLoader = new SecureDexClassLoader(
                finalDexPath.toString(),
                dexOutputDir.getAbsolutePath(),
                libraryPath,
                parent,
                mContextWrapper,
                performLazyEvaluation);

// Provide packageNameToCertificateMap to mSecureDexClassLoader
mSecureDexClassLoader.setCertificateLocationMap(santiziedPackageNameToCertificateMap);

return mSecureDexClassLoader;
```

4.1.3 CacheBinder

`CacheBinder` is an helper component, instantiated by `SecureLoaderFactory`, to keep track of the binding between remote code containers, identified by their remote URL, and the corresponding local file copies, cached in the application-private folder during previous fetching operations. The need for such a component comes out because of our naming convention, which requires the digest of the container to identify a cached archive. Since it is impossible to compute the hash of a remote resource without actually fetching it (and we clearly do not want it since we implemented a caching mechanisms precisely to avoid to fetch continuously remote containers), we need a component, `CacheBinder`, to store and manage these pieces of information.

Listing 4.4: Signature of `CacheBinder`'s constructor.

```
CacheLogger(String cacheDirectoryPath, int daysTillConsideredFresh);
```

In particular, `CacheBinder` relies on a simple data structure, a three-column table, which links the remote URL of the target container with the corresponding file name of the local cached copy in the application-private folder and with a time-stamp, recording when the local copy was fetched from the web.

Whenever the caller requires a new `SecureDexClassLoader` object, `SecureLoaderFactory` creates a `CacheBinder` object by providing a parameter, `daysTillConsideredFresh`, which was previously set in `SecureLoaderFactory`'s contructor.

```
CacheLogger mCacheLogger = new CacheLogger(importedContainerDir.getAbsolutePath(),
     daysBeforeContainerCacheExpiration);
```

When initialized, `CacheBinder` recreates the previously introduced three-column table by parsing an helper file, stored in the app-private folder and containing previous valid bindings. Every time that `SecureLoaderFactory` finds a remote URL for a source container, it queries `CacheBinder` by providing this URL to the method `checkForCachedEntry`; `CacheBinder` now answers with either the name of the corresponding local cached copy or with a `null`.

```
final String checkForCachedEntry(String remoteURL) {

   // If the remote URL is contained in the map, return the
   // linked fresh local container
   if (remoteURLToLocalFileMap.containsKey(remoteURL))
      if (new File(cacheDirectoryPath + File.separator +
           remoteURLToLocalFileMap.get(remoteURL)).exists())
        return remoteURLToLocalFileMap.get(remoteURL);

   // Otherwise no cached entry..
   return null;
}
```

Notice that `CacheBinder` answers `null` in the following situations: (1) when it does not find a valid entry in the table, (2) when it founds an entry pointing to a not-existent local resource, or (3) when the corresponding local container results not fresh enough after comparing its time-stamp with `daysTillConsideredFresh`; in the latter two scenarios `CacheBinder` updates the table by removing the line of the missing or rotten container. In case of a `null` return value, `SecureLoaderFactory` fetches and imports the remote container with the usual naming convention and, later, it notifies `CacheBinder` to add a new entry to the table for the new local resource; otherwise it caches the local copy. Finally, when `SecureLoaderFactory` ends to parse the path strings, it notifies `CacheBinder` of such an event; `CacheBinder` now frames the current situation by storing the state of the table into the previously mentioned helper file. This operation helps to keep consistency between the state of the application-private folder and the `CacheBinder` instances.

4.1.4 SecureDexClassLoader

`SecureDexClassLoader` is the main class of the library and it handles the last part of the remote verification protocol (**Step 4** and **Step 5**). `SecureDexClassLoader` wraps an instance of `DexClassLoader` and, accordingly to the outcome of the security checks presented in step 5, it either prevents the call to the native `loadClass()` method on the wrapped `DexClassLoader` or invokes it allowing the dynamic load operation to happen.

Listing 4.5: Signature of `SecureDexClassLoader`'s constructor.

```
SecureDexClassLoader(   String dexPath,
                        String optimizedDirectory,
                        String libraryPath,
                        ClassLoader parent,
                        ContextWrapper parentContextWrapper,
                        boolean performLazyEvaluation);
```

Initialization When `SecureLoaderFactory` invokes `SecureDexClassLoader`'s constructor, the latter object instantiates its internal components including a `FileDownloader`, responsible for fetching remote resources and presented in the next Subsection, and a `PackageNameTrie`, an object that, given a package name, returns the certificate that should be used for the corresponding container verification (we describe `PackageNameTrie` in Subsection 4.1.6).

```
// Initialization of the linked internal DexClassLoader
mDexClassLoader = new DexClassLoader(dexPath, optimizedDirectory, libraryPath, parent);

certificateFolder = parentContextWrapper.getDir(CERTIFICATE_DIR,
     ContextWrapper.MODE_PRIVATE);
containerFolder = parentContextWrapper.getDir(SecureLoaderFactory.CONT_IMPORT_DIR,
     ContextWrapper.MODE_PRIVATE);

mPackageManager = parentContextWrapper.getPackageManager();
mFileDownloader = new FileDownloader(parentContextWrapper);
mPackageNameTrie = new PackageNameTrie();

// Maps initialization
packageNameToCertificateMap = new LinkedHashMap<String, URL>();
packageNameToContainerPathMap = Collections.synchronizedMap(new LinkedHashMap<String,
     String>());
```

Next, `SecureDexClassLoader` parses the paths of the code containers from which classes will be loaded and, for each one of those, it extracts all the package names associated to the container. Operatively, we can accomplish this task easily for an APK container since a simple query to the `PackageManager` object returns the package name of the archive.

```
// APK container case:
// Use PackageManager to retrieve the package name of the APK container
if (mPackageManager.getPackageArchiveInfo(containerPath, 0) != null) {

   packageNameSet.add(mPackageManager.getPackageArchiveInfo(containerPath, 0).packageName);
   return packageNameSet;
}

return null;
```

On the other hand, we do not have a similar API for JAR containers and so the extraction of the package names is a bit more laborious: at first, we need to extract the `classes.dex` entry into the JAR archive; not finding such an entry automatically implies an invalid JAR container for DCL; on the contrary, if this entry is found, the program loads temporarily this DEX file into an application private folder; next, it analyzes all the class entries stored in the cached DEX file and, for each class, it retrieves the package name by pruning the last dot-separated token from the full class name; lastly, the algorithm returns a set containing all these collected package names. Figure 4.3 shows an example of run of this algorithm to clarify all the involved steps. As soon as a new set of package names for a container is generated, `SecureDexClassLoader` adds entries to an internal associative map to connect each package name with the respective code container. This is a simple workaround for later use to detect easily which archive may contain the implementation of a class, chosen by the developer with the invocation of the `loadClass()` method on the `SecureDexClassLoader` instance.

After the constructor invocation, `SecureLoaderFactory` calls an auxiliary method to provide `SecureDexClassLoader` with the sanitized associative map,

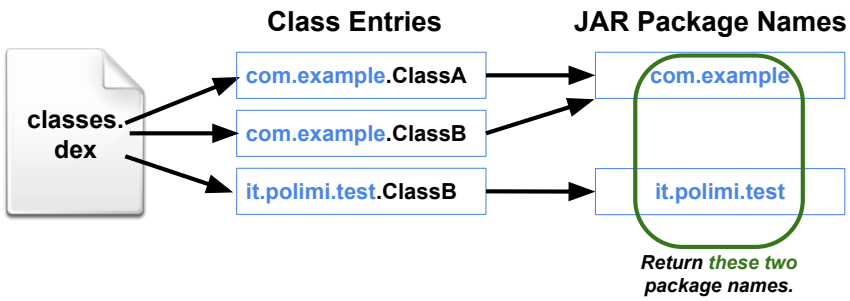


Figure 4.3: JAR package names extraction. This figure shows a run of the algorithm to extract package names from JAR containers. At first, the `classes.dex` entry is extracted and parsed to retrieve all the class names. Finally, for each class name, the algorithm erases the last subfield after the dot and it returns the set of the collected package names.

filled by the developer, that connects package names to the related certificate's URL. `SecureDexClassLoader` replaces now all the empty entries in the map with URL constructed by reverting the package name accordingly with the rules summarized in Table 3.1.

Signature verification After the initialization of `SecureDexClassLoader`, the next step for a developer is querying this object to load dynamically new classes by invoking the `loadClass()` method. Since this is the core part of our implementation we take the time to describe it fully in Section 4.2.

Wiping out app-private cached resources A developer may find sometimes useful to remove resources (i.e., APK, JAR containers, or certificates) that have been cached in the application-private folders to force the retrieval of fresh copies of the same resources in the next load operations. For such a reason, `SecureDexClassLoader` offers a simple public method, `wipeOutPrivateAppCachedData()`, to perform this task.

Listing 4.6: Signature of `SecureDexClassLoader`'s `wipeOutPrivateAppCachedData()`.

```
public void wipeOutPrivateAppCachedData(   boolean containerPrivateFolder,
                                           boolean certificatePrivateFolder);
```

In particular, this method takes as an input two boolean variables: the first one informs `SecureDexClassLoader` whether the content of the cached code containers' folder must be wiped out, whereas the second parameter is used for regulating deletion of the stored certificates. Trivial to say that a call to this method with both the parameters set to `false` has no effect.

4.1.5 FileDownloader

`FileDownloader` is a utility class to fetch remote resources, like code containers or certificates, used for DCL. Both `SecureLoaderFactory` and `SecureDex-`

ClassLoader relies on this component in their code. In particular, SecureLoaderFactory makes use of this class every time that the library user requires a new SecureDexClassLoader instance and provides at least one URL for a remote-located code container, which has not been fetched yet. On the contrary, SecureDexClassLoader delegates FileDownloader to retrieve via HTTPS all the remote certificates, not already fetched, for later signature verifications.

At implementation level, FileDownloader provides a single public method, downloadRemoteURL(), to fetch a remote resource, which requires the remote URL of the resource, the location on the device where to store the retrieved file, and a boolean parameter, which specifies whether FileDownloader should allow redirect links.

Listing 4.7: Signature of FileDownloader's downloadRemoteURL().

```
final boolean downloadRemoteUrl(   final URL remoteURL,
                                   final String localURI,
                                   final boolean isRedirectAllowed);
```

While the first two parameters are self-explanatory, we need to widen more on the use of the third one. The reason for it is that allowing redirect links is necessary in GNR for the implementation of silent updates for third-party libraries. However, following redirect links can be a security threat because an initially protected HTTPS connection may be downgraded to an HTTP one therefore an eavesdropping attacker may try to manipulate data traffic on this unencrypted and unauthenticated connection. For this reason, we implemented GNR with the possibility to decide selectively whether a connection is allowed to follow redirections depending on the type of the fetched file. More in the details, FileDownloader should allow redirection whenever SecureDexClassLoader attempts to fetch remote containers (remember that our verification protocol allows the retrieval of code containers via both HTTP and HTTPS); on the contrary, FileDownloader must deny redirection when SecureDexClassLoader attempts to fetch a remote certificate, as to prevent the attacker tampering with it.

4.1.6 PackageNameTrie

The last presented component is PackageNameTrie. The goal of this class is, given a package name, finding whether or not it exists a root package name, whose definition was presented in Subsection 3.2.6, binded with a valid certificate. To clarify this concept, we present a simple example: Let us assume that a developer initializes a SecureDexClassLoader instance and provides in the associative map a valid certificate linked to the package name it.poli; next, he decides to load the class it.poli.test.one.MainClass; what happens is that, when triggered with the package name it.poli.test.one of the target class, PackageNameTrie returns as a root package name the string it.poli, since it is the longest prefix of the target package name associated with a certificate. In case the developer adds an extra entry to the associative map like (it.poli.test, any remote certificate URL), PackageNameTrie must return the string it.poli.test to the previous method call. On the other hand, if the developer tries to load com.exa.pack.OtherClass from the same SecureDexClassLoader instance, PackageNameTrie must return no root package name

since there is no significant prefix associated to a certificate.

Operationally, as the class name suggests, we designed this component by creating a trie-like data structure. The process requires to populate the trie with all the package names stored in each source container and then to mark among these all of those binded with a valid certificate. Once this setup is done, every query on an input package name can be reduced to a simple visit of the trie, starting from the corresponding leaf and moving backward till the algorithm reaches either a marked node or the root of the trie. Figure 4.4 shows an example of the setup of the trie data structure, the assignment of the package names with valid certificates, and the execution of some queries over it.

At implementation level, mapping this approach into code is not straightforward since Java API does not offer any ready-to-use implementation for a Trie data structure. We thought that the most suitable replacement was an associative map linking each package name to a boolean indicating whether a valid certificate was associated to it. As explained in Subsection 4.1.4, when a new `SecureDexClassLoader` object is instantiated, it parses the strings pointing to the paths of the code containers and, for each one of those, it constructs the list of the related package names. Each package name is stored in the `PackageNameTrie` by invoking the method `generateEntriesForPackageName()`.

```
final void generateEntriesForPackageName(String packageName) {

   String currentPackageName = packageName;
   boolean hasFoundAnAlreadyInsertedPackageName = false;

   while (!hasFoundAnAlreadyInsertedPackageName) {

      if (packageNameToHasCertificateMap.containsKey(currentPackageName)) {

         // In this case this entry has been already inserted in the map
         // so the process of package name generation stops here.
         hasFoundAnAlreadyInsertedPackageName = true;

      } else {

         // Need to insert this entry by populating the map accordingly
         packageNameToHasCertificateMap.put(currentPackageName, false);

         // Now remove the last part of the package name and then
         // repeat the previous step recursively.
         currentPackageName = getUpALevel(currentPackageName);
      }
   }
}
```

This call inserts in the associative map, not only the input package name, but also all the dot-separated prefixes of it. All of these key-entries in the map are coupled with a `false` value because, by now, none of them has a certificate associated. Later, when `SecureDexClassLoader` receives the sanitized map, which links package names to certificates, it invokes the `setEntryHasAssociatedCertificate()` method on the `PackageNameTrie` for all the keys in the latter sanitized map. This method takes as an input a package name and, if this is a key of the internal associative map of the `PackageNameTrie`, it sets the corresponding value to `true`.

```
final void setEntryHasAssociatedCertificate(String packageName) {

   if (packageNameToHasCertificateMap.containsKey(packageName)) {
```

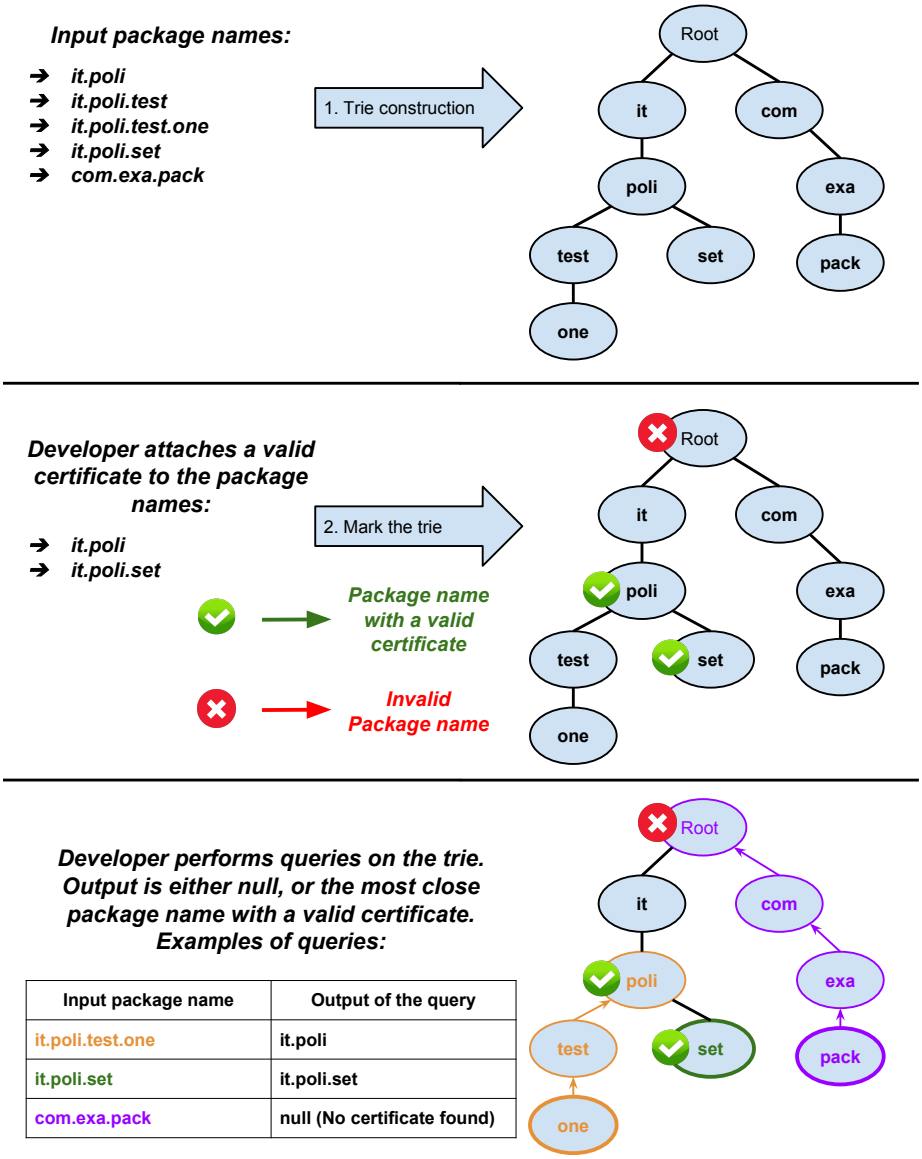


Figure 4.4: `PackageNameTrie` setup, certificate assignment, and use. Given the initial set of the package names of the involved classes, the algorithm constructs the corresponding trie-like structure, where each package name is divided into a set of nodes (the dot character is the separator). Next, the algorithm marks all the entries with a valid certificate on the graph; on the other hand, the root node is always marked as an invalid node. Finally, when the developer provides a new class to attempt to load, the algorithm extracts the corresponding package name and navigates the tree from the associated leaf till the root node. If a valid node is found on the traversal, the algorithm returns as a result the reconstructed package name by appending the node's string content, starting from the root till the valid node; otherwise, if no valid node is found, it returns a `null`.

```
      packageNameToHasCertificateMap.put(packageName, true);
   }
}
```

Finally, when `SecureDexClassLoader` needs to retrieve the root package name, it simply invokes the `getPackageNameWithAssociatedCertificate()` method.

```
final String getPackageNameWithAssociatedCertificate(String packageName) {

   String currentPackageName = packageName;

   if (!packageNameToHasCertificateMap.containsKey(currentPackageName))
      return "";

   while (!packageNameToHasCertificateMap.get(currentPackageName))
      currentPackageName = getUpALevel(currentPackageName);

   return currentPackageName;
}
```

Starting from the input package name, this method checks whether the corresponding value in the internal associative map is set on `true`; if this is the case, `PackageNameTrie` returns the current package name; otherwise it chops the last part of it by invoking the private method `getUpALevel()` and then it repeats the previous check. The end condition for this method is reached when either the algorithm finds a package name with an associated certificate or when the algorithm reaches the root node (the empty string), after a chop, because of a `getUpALevel()` method call.

```
private String getUpALevel(String packageName) {

   int lastPointIndex = packageName.lastIndexOf('.');

   if (lastPointIndex != -1)
      return packageName.substring(0, lastPointIndex);
   else
      return "";
}
```

4.1.7 Implement silent update strategy with GNR

Our library is able to handle correctly silent update strategy. The goal is making a developer able to have his application always running the latest version of an included third-party library without bothering the app users with continuous requests of updates. This should happen while granting the smallest effort for both the third-party library developer and the application developer, which includes GNR in his code. In particular, as depicted in Figure 4.5, the developer of the third-party library needs to setup two links: (1) a redirect link pointing to the latest version of the third-party library archive, signed with the developer's private key; (2) a secure HTTPS link that points to the certificate that embeds the developer's public key. On the other side, Listing 4.8 shows how the developer, who is willing to use the third-party library in his application, has to create an instance of `SecureDexClassLoader` by providing the redirect link as the `dexPath` parameter and an associative map that links the package name of the library (e.g., `com.example`) with the secure link as the `PackageNameToCertificateMap` parameter.

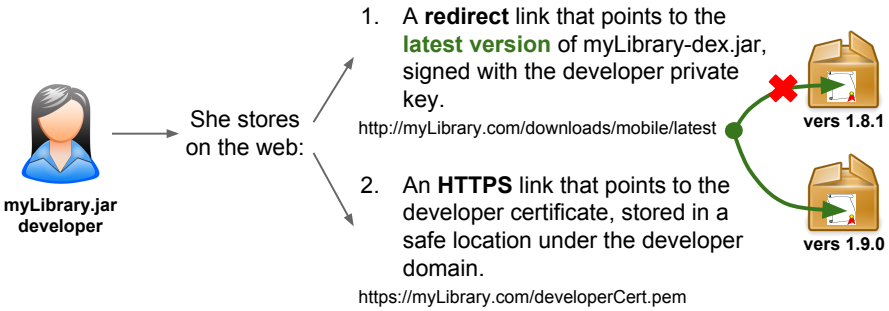


Figure 4.5: Modifications on library developer's side for silent updates. Firstly, she needs to provide a link to the latest version of the library (via both HTTP and HTTPS protocol) and, secondly, a link to the certificate to verify the library's container (via HTTPS protocol).

Listing 4.8: Code for silent updates on application developer's side.

```
Map<String, URL> packageNamesToCertMap = new HashMap<String, URL>();
packageNamesToCertMap.put( "com.example", new
     URL("https://myLibrary.com/developerCert.pem"));

SecureLoaderFactory mSecureLoaderFactory = new SecureLoaderFactory(this);
SecureDexClassLoader mSecureDexClassLoader =   mSecureLoaderFactory.createDexClassLoader(
        "https://myLibrary.com/downloads/mobile/latest",
        null,
        getClassLoader(),
        packageNamesToCertMap);

Class<?> loadedClass = mSecureDexClassLoader.loadClass("com.example.ClassA");
```

With this settled, the app developer can load the target classes from the latest version of the library's container. In particular, when the library developer releases a new version of her software, she simply needs to update the redirect link to reference the new container's version. On the app developer's side, nothing changes in the code. When the application starts the next time, GNR retrieves the latest version of the container and validates it against the usual trusted certificate; since also this new version of the library was signed with the same developer's private key, the signature verification process is going to succeed and, therefore, the app is going to load the latest version of the library classes.

4.1.8 Open-source release

We implemented GNR as a Java project starting and we made it publicly available on Github [10]. Our library is compatible with both the Android Development Tool (ADT) and the Android Studio (AS) IDE and it is a drop-in tool that a developer can easily insert in his Android projects. We also published GNR on Jcenter to make it compatible with tools, like Maven, that automatically handle library dependencies. In the repository, alongside the source code, a developer can find full and extensive documentation in a tutorial-like format to

learn quickly how to use GNR and even a simple example project, which shows some of the advantages in choosing our solution over native `DexClassLoader` API.

4.2 Signature verification: Implementation details

The signature verification is the core part of the implementation of Grab'n Run. In Subsection 4.2.1, we describe the `loadClass()` method of `SecureDexClassLoader`, which embeds the verification and is used to load new classes dynamically; in Subsection 4.2.2, we describe the implementation of this method; whereas in Subsection 4.2.3, we present the two strategies (i.e., lazy, or eager) used by `SecureDexClassLoader` to perform the operation.

4.2.1 Signature verification as a "black" box

After the initialization of `SecureDexClassLoader` (Subsection 4.1.4), the next step for a developer is querying this object to load dynamically new classes by invoking the `loadClass()` method, which, as its original counterpart, takes as a parameter the full name of the target class to load.

Listing 4.9: Signature of `SecureDexClassLoader`'s `loadClass()`.

```
public Class<?> loadClass(String className) throws ClassNotFoundException;
```

Also the return values of this method do not differ significantly from the ones of the native `DexClassLoader`: this method, in fact, returns a class instance, in case that an implementation for the target class is found into one of the code archives that successfully pass the signature verification; a `ClassNotFoundException`, whenever the code archives successfully overcome the security checks but none of them contains an implementation for the target class; and a `null` reference, in case that one (or more) of the security requirements of our model was not respected. Obtaining a `null` as a return value is a necessary but not sufficient condition to conclude that an attacker tried to compromise the user application. For this reason, we present the full list of the situations in which `SecureDexClassLoader` returns a `null` to the caller of the `loadClass()` method:

1. **Missing trusted certificate**. `SecureDexClassLoader` was not able to find a valid certificate to evaluate the container before loading the target class. This situation may happen for several reasons (e.g., the developer forgot to insert an entry that links the package name of the target class with a remote certificate URL, or he might have inserted a typo in the certificate URL, or, then again, he might have provided a URL pointing to an inexistent certificate, or, finally, the device might lack of connectivity therefore it was unable to fetch the remote certificate).

2. **Invalid trusted certificate**. `SecureDexClassLoader` was able to retrieve a certificate for the validation but this element was not a suitable one. For example, the certificate does not conform to the required X.509

standard, it is expired, or it is the Android Debug Certificate(i.e., the certificate used to sign applications in Android IDE for debugging purpose, which must be rejected at production phase).

3. **Unsigned source archive**. Although this may sound naïve, many developers of JAR libraries are not accustomed in signing their archives before releasing them. This bad practice makes impossible realizing whether a container is genuine or a repackaged one. For such a reason, `SecureDexClassLoader` always denies DCL from unsigned JAR containers.

4. **Archive verified only against untrusted certificates**. In this case, a code container satisfies the signature verification against a set of certificates, which, unfortunately, does not include the trusted one. Because of this absence, `SecureDexClassLoader` must reject dynamic loading for classes in this container.

5. **Invalid signature**. In this last case, `SecureDexClassLoader` checked a container, which resulted properly signed against the trusted certificate, except for one (or more) entry that does not match the expected signature. This is the most probable scenario in which an attacker tried to repackage a genuine container coming from a benign developer. In such a case, the attacker manipulated some of the entries in the archive but, since she does not own the private key of the developer, she is not able to resign the code container and that is why the manipulated entries do not match the signature anymore.

Notice that, whenever one of the latter three scenarios happens, `SecureDexClassLoader` deletes the container from the application-private cache folder. On the other hand, in the first two situations, `SecureDexClassLoader` keeps the cached version of the container on the device storage because the issue is missing a trusted and valid certificate and, therefore, the cached container should be considered genuine until we are able to prove the opposite.

One of the goals of our project is keeping GNR library as simple and easy-to-learn as possible for developers. For this reason, instead of modifying the `loadClass()` method signature by introducing custom exceptions that may result bothersome to learn and handle, we decide to simply return a `null` for failures. One may argue that this is a too simplistic way to handle so many different cases of failure; but, indeed, simplicity is exactly what we are looking for. Moreover, we dabbed this aspect by implementing all the classes of GNR to log their own key events, including outcomes of the steps of the verification protocol. Arguably, we think that this solution is a good trade-off between simplicity and security: in fact, those developers that are not interested in security details can simply perform a not-null check on the return value of the `loadClass()` method to verify whether they can load a class dynamically; on the other hand, developers, who want to figure out what is happening behind the curtains, can inspect the device's logs to have a clear look on the state of the library's classes and on the reasons of failures of loading operations.

4.2.2 Signature verification as a "glass" box

After having presented the input and output relationship of the signature verification process, we describe how we implemented the algorithm for signature

verification, previously outlined in Section 3.2.5. Appendix A.1 contains its Java implementation, named `verifyContainerSignatureAgainstCertificate()`.

Listing 4.10: Signature of `verifyContainerSignatureAgainstCertificate()`.

```
private boolean verifyContainerSignatureAgainstCertificate(
                                                    String containerPath,
                                                    X509Certificate verifiedCertificate);
```

The input parameters of the algorithm are a code container and a trusted certificate. Notice that these inputs has been already sanitized previously in the code, thus the first parameter is a path pointing to a code container (JAR or APK) with a `classes.dex` entry and the second parameter is a valid X.509 certificate. The return value of this method is a boolean, which assets whether the input container is properly signed against the trusted certificate. The algorithm is split into two parts: the first one is executed when the input container is an APK, whereas the second one is triggered either when the input is a JAR container or an APK that successfully passed the first part. Moreover, the method returns a true value only when the second part of the algorithm ends correctly; so, this method returns true when either a JAR container succeeds in the second part or an APK passes both the first and the second part; any other control flow returns false.

In the first part, the algorithm queries a `PackageManager` object to retrieve the array of the certificates used to sign the APK container. Notice that, in this case, Android API are inaccurate and even confusing since they call a "signature" what in fact is a certificate. Once the array of the certificates is available, the next step is reconstruct each one of these entries as a `Certificate` object, starting from its byte stream, check its validity, and finally compare all the reconstructed certificates against the trusted one.

```
// Recreate the certificate starting from this signature
inStream = new ByteArrayInputStream(sign.toByteArray());
certFromSign = (X509Certificate) certificateFactory.generateCertificate(inStream);

// Check that the reconstructed certificate is not expired..
certFromSign.checkValidity();

// Check whether the reconstructed certificate and the trusted one match
// Please note that certificates may be self-signed but it's not an issue..
if (certFromSign.equals(verifiedCertificate))
   // This a necessary but not sufficient condition to
    // prove that the APK container has not been repackaged..
    signatureCheckIsSuccessful = true;
```

If one of those matches the trusted certificate, then the APK container may have been signed with the proper private key so it is worthy to evaluate it also in the second phase; otherwise we can already reject the input APK. In other terms, passing this step correctly for an APK container is a necessary but not sufficient condition for succeeding in the signature verification.

In the second step, we can generalize each incoming container to be a JAR (and indeed this is correct because APK containers are just an extension of regular JAR). The algorithm passes the candidate JAR container, along with the trusted certificate, to a subroutine called `verifyJARContainer()`. This routine raises an exception in case that the JAR violates one (or more) of the security constraints. This is fine since the caller of this method catches any

raised exception and, in turn, reports that the signature process failed. The final step is analyzing how `verifyJARContainer()` works internally. Appendix A.2 shows its code.

Listing 4.11: Signature of `verifyJARContainer()`.

```
private void verifyJARContainer(JarFile jarFile, X509Certificate trustedCert) throws
     IOException;
```

We implemented this method as a slight modification of the example code for JAR signature verification, provided by the Java Oracle documentation [8]. In particular, after the initial sanity checks for not null parameters, `verifyJARContainer()` looks for the JAR manifest. In case of a signed JAR, this file contains the list of the digests of all the file entries at the time of container's signing. JAR containers missing the manifest can be immediately rejected because they have never been signed.

```
// Ensure the jar file is at least signed.
Manifest man = jarFile.getManifest();
if (man == null) {
   throw new SecurityException("The container is not signed");
}
```

Next step is verifying that the digests of all the file entries in the archive match the digests in the manifest file. For this purpose, the algorithm retrieves the byte stream of each entry and use the read API method to parse it. While reading each entry, a `SecurityException` is raised whenever the digest of the current file entry does not match the corresponding digest stored in the manifest.

```
// Current entry in the jar container
JarEntry je = (JarEntry) entries.nextElement();

// Skip directories.
if (je.isDirectory()) continue;
entriesVec.addElement(je);
InputStream inStream = jarFile.getInputStream(je);

// Read in each jar entry. A security exception will
// be thrown if a signature/digest check fails.
while (inStream.read(buffer, 0, buffer.length) != -1) {
   // Don't care as soon as no exception is raised..
}

// Close the input stream
inStream.close();
```

This check ensures the integrity of all the entries in the JAR archive. As an additional check, digest values for the manifest file itself are recomputed and compared against the values recorded in the signature file. Once again, when an entry does not verify the signature, a `SecurityException` is raised. This second check grants that the signing process was performed by the private keys coupled with the certificates inside the signature block files. For this reason, the last step of this method is rebuilding, for each signed entry, the list of the certificates used to verify it and checking that, among those, there is also the trusted certificate. If this condition holds for all the entries, except folders and files in `META-INF` folder, which are not signed by convention, than the JAR container successfully verifies the signature.

```
// Every file must be signed except files in META-INF.
```

```
Certificate[] certificates = signedEntry.getCertificates();
if ((certificates == null) || (certificates.length == 0)) {
   if (!signedEntry.getName().startsWith("META-INF")) {
      throw new SecurityException("The container has unsigned class files.");
   }
} else {
   // Check whether the file is signed by the expected signer. The jar may be
   // signed by multiple signers. So see if one of the signers is ``trustedCert''.
   boolean signedAsExpected = false;

   for (Certificate signerCert : certificates) {
      try {
         ((X509Certificate) signerCert).checkValidity();
      } catch (CertificateExpiredException | CertificateNotYetValidException e) {
         // On Android a common practice is using certificates (even self signed)
         // with at least a long life span and so temporal validity should be enforced..
         throw new SecurityException("One of the used certificates is expired!");
      } catch (Exception e) {
         // It was impossible to cast the general certificate into an X.509 one..
      }

      if (signerCert.equals(trustedCert)) {
         // The trusted certificate was used to sign this entry
         signedAsExpected = true;
      }
   }

   if (!signedAsExpected) {
      throw new SecurityException("The provider is not signed by a trusted signer");
   }
}
```

4.2.3 Lazy vs eager strategy of verification

We implemented GNR such that two strategies (i.e., lazy and eager) are available for signature verification. The lazy strategy implies that `SecureDexClassLoader` evaluates each code container only at the exact moment in which the main application performs a call to the `loadClass()` method for dynamically loading a class in it. An ideal use case for this mode is when the developer sets up a `SecureDexClassLoader` instance with a considerable number of containers but the application has to load at runtime just a couple of classes, which may also vary from one execution to the other, and so validating all the containers in such a scenario can be a waste of time. As regards performance overhead, this strategy becomes less efficient than the eager strategy, presented later, when `SecureDexClassLoader`, during its life-cycle, has to evaluate almost all the code containers.

Operationally, we implemented the lazy strategy by moving the signature verification process of the single code container at the beginning of the `loadClass()` method. Thanks to the associative map linking package names to container that `SecureDexClassLoader` creates during its initialization, knowing which container to verify, given the full class name, is trivial. In a similar fashion, the certificate that must be used for the validation is obtained by querying the `PackageNameTrie` object, which, given a package name, returns whether a root package name with an associated valid certificate exists. If this test returns true, the algorithm performs the signature verification and propagates the outcome of the process to all the other package names pointing to the same container. With this method and by checking whether a package name has been already verified at the beginning of the `loadClass()` method, we grant that GNR evaluates each archive only once.

On the contrary, the eager strategy relies on verifying the signature of all the containers concurrently as soon as possible (i.e., at the end of the initialization phase before returning a new `SecureDexClassLoader` instance). This concurrent evaluation reduces, at run time, the execution overhead of the application from the sum of the times for signature verification of all the containers, to, more or less, the time for signature verification of the biggest code container. After this initial burden, any successive call to the `loadClass()` method will have execution time almost equal to the respective call on native `DexClassLoader`, thanks to the caching system on the results of signature verifications. Since eager strategy seems to fit better with general-purpose use cases of our library, we decided to have it as the default strategy. Anyway, a developer can decide to apply the lazy one by simply providing an extra `true` parameter to the `createDexClassLoader()` method invocation on a `SecureLoaderFactory` instance.

Operationally, we implemented the eager strategy by iterating over all the package names stored in the associative map linking package names to the related container. For each of these package names, the algorithm checks whether a related root package name exists by querying the `PackageNameTrie` object and, if this is the case, it adds an entry to a temporary associative map, where the key element is the container linked to this package name and the value is the root package name that holds a valid certificate. Next, the algorithm initializes a thread for each entry of the latter associative map; every running thread, then, performs the signature verification process, presented in the next paragraph, on its associated code container against the trusted certificate provided by the root package name and, in case of success, adds it to a set containing only the successfully verified containers. Once all the threads complete their job, the algorithm returns the `SecureDexClassLoader` instance to the developer. Now every time that the developer wants to load a class dynamically, `SecureDexClassLoader` will allow the operation if the package name of the target class is linked to a container that belongs to the set of the successfully verified ones.

4.3 Repackaging tool implementation

In the final section of this chapter, we present the implementation of the repackaging tool, which follows the approach outlined in Section 3.5. We implemented the repackaging tool as a Python script that relies on Androguard [5], an open-source Python tool for reverse engineering and static analysis of Android applications, and apktool [6], another tool for reverse engineering third-party, closed, binary Android applications. At first (Subsection 4.3.1), we introduce the settings that a developer can customize for the patching process, whereas in Subsection 4.3.2, we describe the most important details of the script's functioning. Finally, Subsection 4.3.3 provides insights on some technical challenges solved by our tool while patching smali code.

4.3.1 User settings

A developer can execute the script either by filling in a simple GUI or by launching from command line the script with an attached configuration file. For simplicity sake, we now describe the required inputs by presenting the different

screens of the GUI.

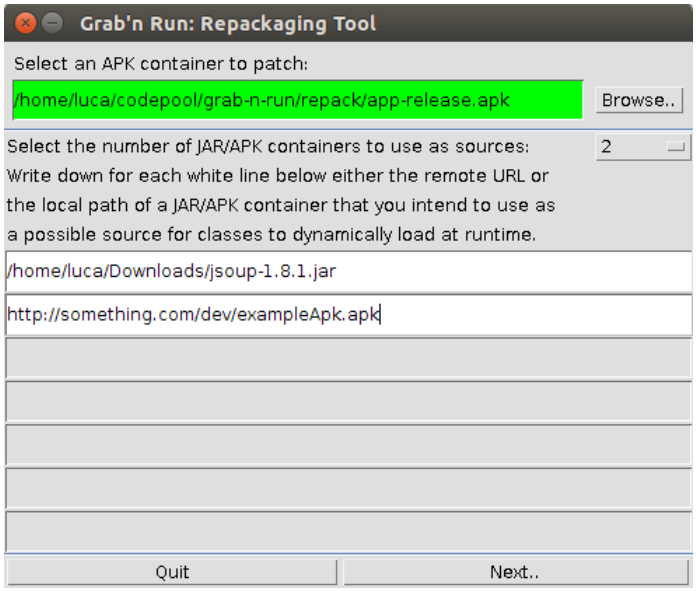
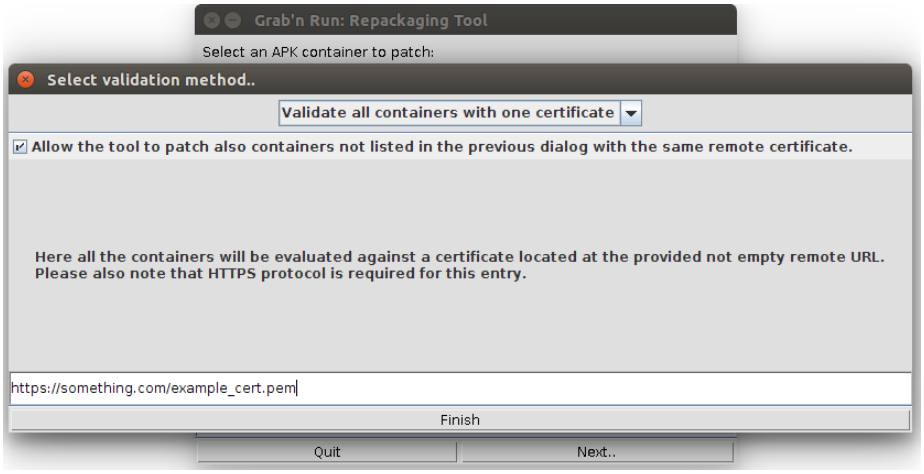
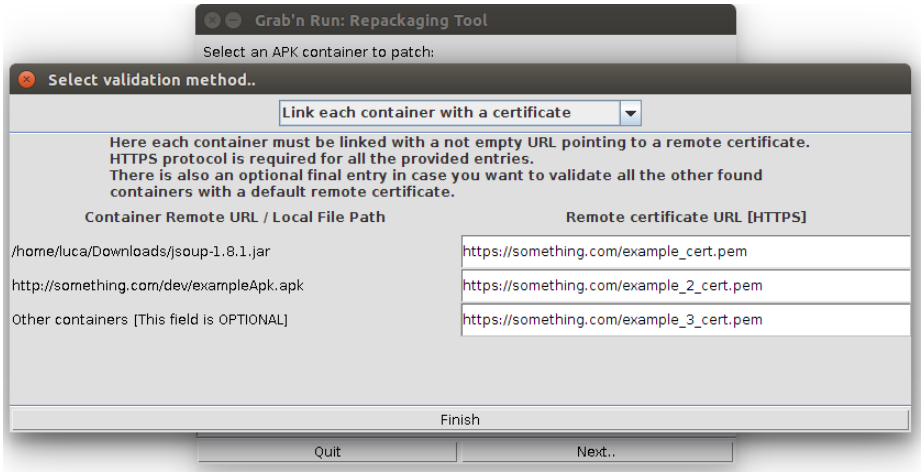


Figure 4.6: First screen of the repackaging tool GUI. In this window the developer can select the APK of the application that he wants to patch, alongside all the containers that will be used by the application as sources for the DCL operations.

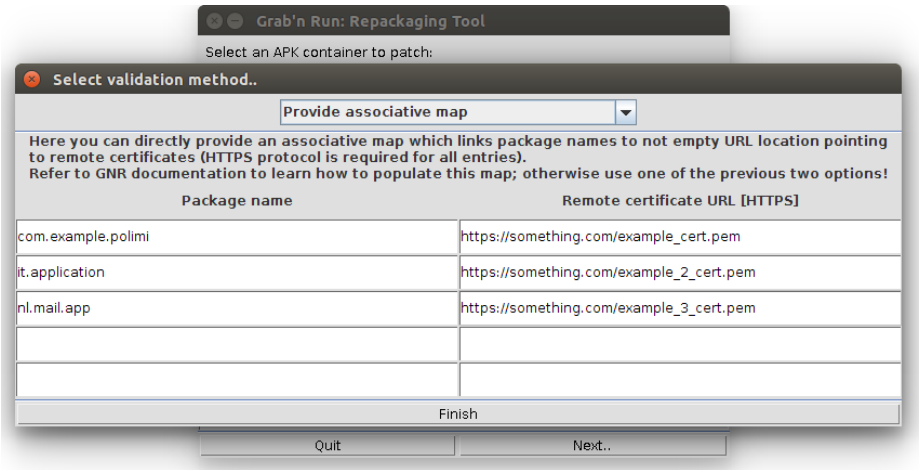
In the first screen (Figure 4.6), the GUI invites the developer to select the APK container that he wants to patch, alongside all the different code containers that the developer plans to use in his application as sources for DCL. These latter code containers can be, as usual, either APK or JAR with a `classes.dex` entry. Moreover, the user can provide both a path pointing to a container stored locally or a URL for remote containers. Once the user presses the OK button, as to help deciding the strategy to validate the previously inserted code containers, the GUI presents a drop-down menu with three options (Figure 4.7): (1) validate all the containers against a unique trusted certificate, whose remote URL is provided by the user (Figure 4.7a); (2) link each container with the remote URL of a different trusted certificate (Figure 4.7b); (3) provide directly an associative map linking package names to the remote URL of the trusted certificates (Figure 4.7c). Notice that this latter option is the equivalence of the `PackageNameToCertificateMap` that a developer would have to provide to initialize a new `SecureDexClassLoader` object in the application source code, therefore this option should be selected only by a developer, who knows how this map works in the GNR library. On the other hand, if a developer prefers an immediate and more easy-to-use solution, he should pick one of the first two options, which provide a simple way to directly connect containers to certificates and also offer the possibility to verify against a default trusted certificate all of those code containers, encountered at runtime, which were not directly listed by the developer in the settings customization phase.



(a) Validate all containers against the same trusted certificate.



(b) Validate each container against a different trusted certificate.



(c) Provide an associative map that links package names to remote certificates' location.

Figure 4.7: GUI screens of the three strategies to verify code containers in the patched application.

4.3.2 Functioning

After having presented the inputs required by the tool, we discuss how the script patches the input APK according to the user preferences. Figure 4.8 summarizes the main steps of the process explained in details in the next paragraphs.

Analysis and preprocessing As soon as the developer ends the settings customization phase, the first step of the script is assessing whether the input APK is valid. We delegate this operation to the `APK` class in the Androguard API, which provides a convenient `is_valid_APK()` method to perform the job. Next, the script performs static analysis on the input APK container to extract the sensitive points that are going to be patched later. Once again, we relied on Androguard for collecting the required pieces of information: In particular, the internal `androlyze` Python script provides an easy-to-use method, `AnalyzeAPK()`, which requires as input the path pointing to the APK and returns a triple of objects that can be queried to obtain useful pieces of information. Among those, the script extracts the set of the app permissions and, by comparing them with the ones required by the GNR library (i.e., `android.permission.ACCESS_NETWORK_STATE`, `android.permission.INTERNET`, and `android.permission.READ_EXTERNAL_STORAGE`), it can easily determine which permissions adding to the Android Manifest of the APK to patch. Moreover, during its analysis, Androguard traces the calls regarding `DexClassLoader` API (i.e., `DexClassLoader` constructor, and `DexClassLoader.loadClass()`) and it makes the results available though the two methods `analysis.is_dyn_code()`, which returns a boolean on whether function calls to the cited API are performed, and `analysis.show_DynCode()`, which shows exactly which classes and methods invoke `DexClassLoader` API functions, if any. Therefore, our script must at first call `analysis.is_dyn_code()`: If the call returns `false` than the APK does not need any patching so the script terminates with an explicative message; otherwise, Androguard has just found some relevant `DexClassLoader` function calls that the script needs to patch. In this latter case, the script parses the textual output of the `analysis.show_DynCode()` method to create its own data representation of the sensitive points. Notice that this preprocessing step on the sensitive points, although not being mandatory, is extremely useful to understand whether an APK needs to be patched, which classes needs to be fixed, and which ones can be skipped straightaway.

Decoding and adding missing permissions At the current stage, we have an APK container, which makes use of DCL, a list containing all the sensitive calls to the `DexClassLoader` API, and, finally, the set of the permissions to add to the Android Manifest of the APK. Now, the first step is disassembling the APK archive so that the script can patch it. For this purpose, our script invokes apktool with the decode option. This brings resources (including the Android Manifest) back to nearly original form and it disassembles the `classes.dex` entry into a hierarchical set of smali classes, which are human-readable and can be patched more easily compared to the Dalvik byte code. Once an APK is decoded, a programmer can modify both the resources and the smali classes; finally, apktool can rebuild the modified version of the container and, if the developer did not introduce any mistake, he obtains back a working and patched application. This is exactly what happens in our script. In particular,

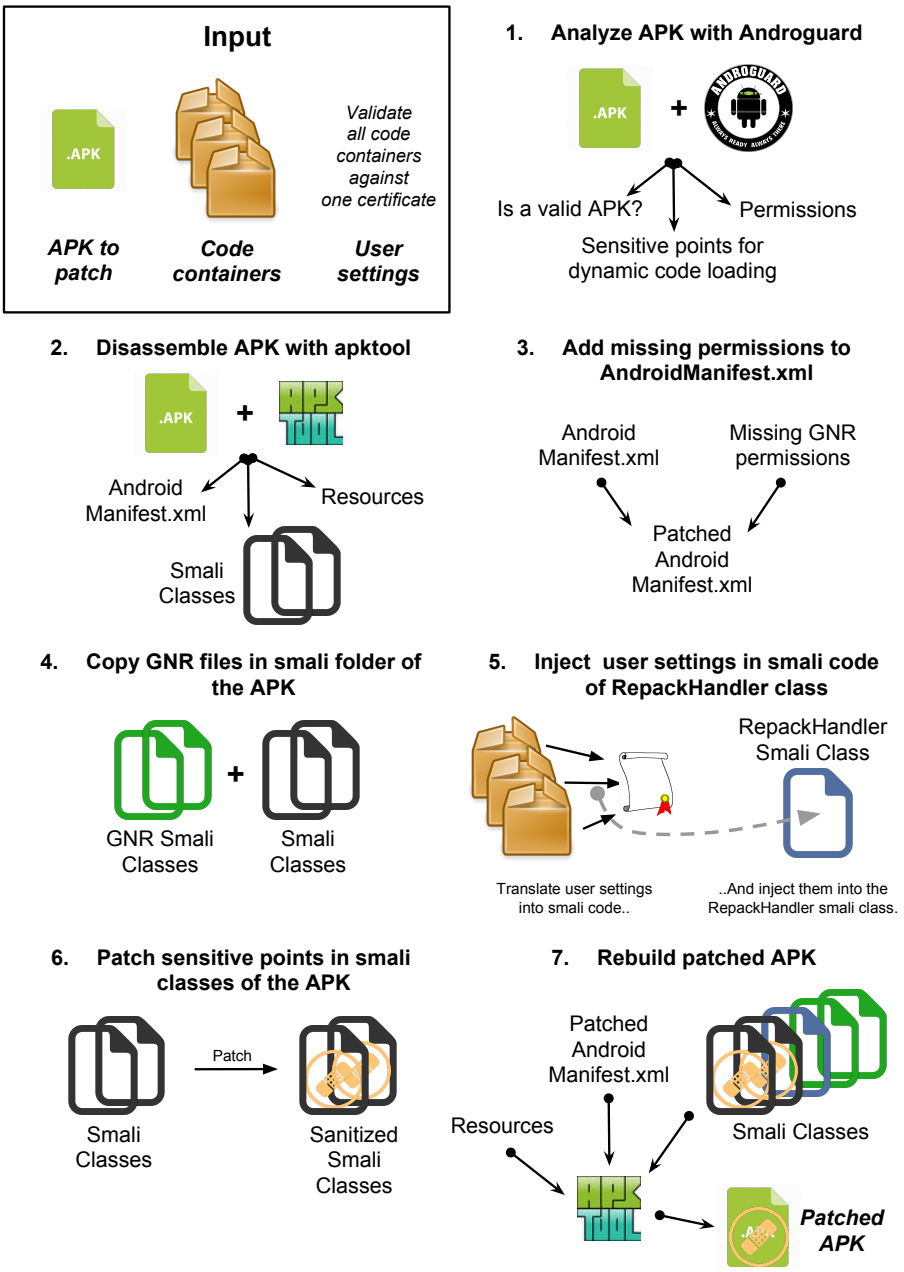


Figure 4.8: Repackaging tool functioning. This figure illustrates the main steps of the repackaging script, starting from the input provided by the developer till the return of the patched APK.

after that the APK is decoded by apktool, our script worries about adding to the Android Manifest the missing permissions necessary for running GNR. An Android Manifest is nothing more than an XML file and, for this reason, our

script simply parses it by leaning on the `xml.etree` Python module and adds extra nodes, one per each missing permission, to the XML root tree.

Patching smali classes Before rebuilding the decoded and modified APK through apktool, the final step of the script is patching the smali code. Our tool needs to modify two kinds of classes: (1) those ones that are or extends the `Activity` class, (2) those ones that performs calls to the `DexClassLoader` API. While the script has already collected the names of the classes in the latter category, as explained in Paragraph "Analysis and preprocessing," the script detects classes for the former category on-the-fly. The reason for detecting `Activity` classes is merely a technical implementation detail: Differently from native `DexClassLoader` API, when a `SecureLoaderFactory` instance is initialized in GNR, it requires a hook to a `Context` object (e.g., an `Activity` class) to be able to perform basic operations on a device, like opening a file on the local storage or checking the network state; for this reason, we need to keep track of all the not yet finished `Activity` objects to be able to provide a valid hook independently from where exactly the patched application executes DCL. The simplest solution is keeping a stack with all the `Activity` objects, from which popping out a reference every time that a new `SecureLoaderFactory` instance needs to be created. At implementation level, we inserted this stack and the method to push an extra activity reference in `RepackHandler`, a static class under our control used to ensure a correct patching. We also developed our script so to add a call in the patched application to this method at the beginning of each `OnCreate()` of smali classes extending the `Activity` one. With this modification, whenever a new `Activity` class starts at run time, it adds a reference to itself in the `RepackHandler`'s stack. Correspondingly, the application can easily identify at runtime activities for initializing `SecureLoaderFactory` by simply inspecting the stack for a running `Activity`.

Finally, the last required patch operation regards all function calls triggering API for DCL. For this step, our script uses a grep-based approach that analyzes all the smali classes that uses `DexClassLoader` API and, every time that a sensitive point is found, it perform sanitation by substituting with either a corresponding method invocation from the `RepackHandler` static class or a partial rewriting of the sensitive point. Table 4.2 lists all these points patched by our repackaging tool and, alongside the related sanitized code, it provides an explanation for each modification.

4.3.3 Further technical details on patching smali code

Patching the smali code presents a set of technical issues that are not trivial to solve. The biggest one is that smali requires, at the beginning of each class method, the declaration of the number of local parameters that are going to be used. Mishandling this number, for example, by declaring a smaller quantity of local parameters than the actual number involved in the method, makes the application unstable and leads it to crash at run time on the device. For this reason, we need extra care when dealing with smali patching, especially in our case, where the functions that we want to substitute (e.g., `loadClass()`) require for their invocation a consistent number of parameters, and thus of locals. The easiest trick to circumvent this issue is substituting all the interested function calls with signature-equivalent method calls to an external static class

under our control. This class will perform in its methods the same semantic functionalities of the initial dynamic calls but it will invoke, instead, equivalent methods belonging to the GNR library. Thanks to this technique, the script is able to substitute the native dynamic API and, at the same time, to keep the smali classes working properly since the number of local parameters is not changed. Another strength of this approach is that it does not require any previous knowledge on the code in the input APK because we do not alter the whole structure of the code but simply some specific function calls in it.

To make the substitutive methods callable, the tool has to copy into the "smali" folder of the decoded project the translation in smali of all the classes from the GNR library plus an extra one, `RepackHandler`, which is our static class, whose methods will replace the native API for DCL. Among its methods, `RepackHandler` provides a static one that takes as input the same parameters of native `DexClassLoader` constructor and contains all the boilerplate code needed for the instantiation of a `SecureDexClassLoader` instance, which is returned at the end of the method. A relevant question is how exactly `RepackHandler` populates the associative map linking package names to remote certificates' URL for `SecureDexClassLoader` setup and here it is where the developer preferences play a significant role. Indeed, our script customizes, during its run, the implementation of a `RepackHandler`'s method according to the preferences set by the developer. More in the details, our script translates user preferences into a set of assignments to internal data structures of the class that, when triggered at runtime, builds the corresponding associative map for the `SecureDexClassLoader` instantiation. As an example, let us consider the case in which a developer provides the repackaging tool directly with an associative map. In this scenario, the tool has to face a simple task since it just needs to inject in the method the corresponding smali code to populate the final associative map. A slightly more difficult case to handle is when a developer provides in the user preferences one (or more) container to link to a certificate. In this case, to avoid slow-down of the performance on the device at runtime, we force the script to retrieve a copy for all the code containers used as sources and, for each one of those, the tool computes the set of the contained package names, before injecting smali code for populating the associative map with the binding of each package name of the container with the proper certificate. Once the script has completed this injection step, when at runtime the application will invoke a method on the `RepackHandler` for the first time, an internal method is going to be triggered to generate the associative map according to the developer preferences and, from now on, used in the initialization of all the `SecureDexClassLoader` objects.

Patching of sensitive points in smali code	
Original point:	`invoke-super {VAR0,VAR1}, LCLASSNAME;->onCreate(Landroid/os/Bundle;)V`
Applied patch:	`invoke-static {VAR0}, Lit/necst/grabnrun/RepackHandler;->enqueRunningActivity(Landroid/app/Activity;V`
Explanation:	When a new `Activity` is created, its reference must be stored in the `RepackHandler`.
Original point:	`new-instance VAR0, Ldalvik/system/DexClassLoader;`
Applied patch:	Empty Line
Explanation:	Instantiation of new `DexClassLoader` objects must be removed.
Original point:	`invoke-direct {VAR0, VAR1, VAR2, VAR3, VAR4}, Ldalvik/system/DexClassLoader;-><init>` `(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/ClassLoader;)V`
Applied patch:	`invoke-static {VAR1, VAR2, VAR3, VAR4}, Lit/necst/grabnrun/RepackHandler;->generateSecureDexClassLoader` `(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/ClassLoader;)Lit/necst/grabnrun/SecureDexClassLoader;` `move-result-object VAR0`
Explanation:	The invocation of `DexClassLoader`'s constructor must be substituted with a call to the `RepackHandler`'s method to generate a `SecureDexClassLoader`. The returned object must be moved in the register originally holding the new instance of `DexClassLoader`.
Original point:	`invoke-virtual {VAR0, VAR1}, Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;`
Applied patch:	`invoke-static {VAR0, VAR1}, Lit/necst/grabnrun/SecureDexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;`
Explanation:	Each invocation to the original `loadClass()` method must be substituted with a call to `SecureDexClassLoader`'s `loadClass()`.
Original point:	`move-result-object VAR0`
Applied patch:	`if-nez VAR0 :sec_checkgnr_LABEL` `invoke-static {}, Lit/necst/grabnrun/RepackHandler;->raiseSecurityException()V` `:sec_checkgnr_LABEL`
Explanation:	After a call to the `loadClass()` method, the return value must be evaluated in the patched code: If it is null, a security exception must be raised by invoking the corresponding method on the `RepackHandler`.

Table 4.2: Patching of the sensitive points. This table correlates smali instructions that are marked as sensitive points by our tool, in the first row of each block, with the corresponding substitutive snippet of code, in the second row. The third row provides a summary explanation on each block.

Chapter 5

Experimental validation

In this chapter we describe the set-up and the outcomes of the experiments performed to evaluate GNR (Section 5.1) and the repackaging tool (Section 5.2).

More in the details, initially (Subsection 5.1.1) we present a case study that we performed on 9 Android developers to verify whether GNR is easier to use than `DexClassLoader` API and its learning overhead is modest for a developer.

Next (Subsection 5.1.2), we introduce a second experiment, whose goal is to estimate the runtime overhead of GNR over standard `DexClassLoader`, and we show that our solution adds only a negligible delay to the original API.

In the remaining part of this chapter (Section 5.2), we present the evaluation performed on our repackaging tool.

5.1 Grab'n Run validation

In this section we outline the two experiments that we used to prove that GNR is an easy-to-use, effective, and secure tool that does not impact the performance of DCL operations significantly.

5.1.1 User study

Goal The goals of this first experiment are to evaluate whether our library is easier to use than `DexClassLoader` API and to prove that GNR learning overhead for a developer is modest, once he knows how to use native API for DCL. Moreover, we would like to collect comparative results between the two solutions under several aspects, such as efficiency, code readability, security, and maintainability.

Setup To reach previous goals, we conducted a use case study on 9 developers. In particular, we selected these guys with different ranks of expertise in Android developing (e.g., some of them had developed only a couple of toy apps for fun, whereas some others use Android on a regular basis); still, all the participants had never used `DexClassLoader` API prior to this experiment.

For the study, the developer had to:

Error	Triggering example	Percentage of developers
Fetch code in an unsafe way	Use HTTP connection instead of HTTPS	66.7 % (6/9)
Store code in a world-writable area	Save code container on external storage	44.4 % (4/9)
	Wrongly initialize optimized cache folder	0.0 % (0/9)
Miss or fail security checks	Do not implement any custom integrity check	100.0 % (9/9)

Table 5.1: Developers' mistakes. This table lists the mistakes that a developer may introduce in his code using `DexClassLoader` API. Each one of these errors leads to a potential security vulnerability. The table correlates each error with an exemplifying triggering condition and with the number of developers that introduced it in our use case study.

1. **Phase 1:** Implement a functionality involving DCL using `DexClassLoader` API. Starting from a skeleton application, he had to fetch a remote code container, store it on the phone, and load dynamically a class inside of it.
2. **Phase 2:** Implement the same functionality but using our library instead.
3. **Phase 3:** Send us the source code of both implementations and fill in a comparative survey on the two solutions.

Notice that, although we provided a toy app, we explicitly recommended the participants to consider it as a real application, thus to implement it carefully and so to be secure at their judgment. Additionally, we left developers free to consult any on-line resource while performing the experiment, including the official Android documentation.

Outcome Table 5.1 quantifies the occurrences of the mistakes discussed in Section 2.3 using `DexClassLoader` API for implementing the DCL functionality in the toy app. We collected these data by manually inspecting the source code of all the submitted implementations. A positive result is that none of the developers failed in storing the optimized version of the container in a world-writable area of memory: The reason for this is that a warning on this specific security issue is present in the `DexClassLoader` API documentation and all the developers probably saw this best practice and applied it in their code. However, an alarming result is that not even a single developer thought about implementing custom checks for verifying the integrity or the signature of the fetched container.

This result is in line with the results obtained in [15] on the number of DCL-caused vulnerabilities found in Google Play apps. Thus, we can conclude that performing DCL in Android securely is not a naïve task. That is why solutions like GNR can be an helpful tool for developers.

Finally, Table 5.2, 5.3, 5.4, and 5.5 show a summary of the answers that participants provided in the comparative survey of the user study: For all the developers the learning overhead imposed by our library was little or almost null. Moreover, the average time spent for understanding and implementing the DCL functionality lowers from the 170 minutes of `DexClassLoader` API to the just 40 of GNR. Additionally, 7 out of 9 of the involved participants thought that the second implemented snippet of code, which relies on our library, would be easier to maintain and modify and it is actually simpler to read compared to the first one; 8 of them also asserts that GNR code was easier to implement and

that our solution is more flexible (i.e., automatically fetch remote containers, store them appropriately) and secure (i.e., perform integrity checks at run time on source code containers) than the native `DexClassLoader`.

5.1.2 Measurement of the runtime overhead

Goal The aim of this experiment is showing that the performance overhead of GNR is almost negligible over `DexClassLoader` API, after an initial one-time penalty.

Setup For supporting this claim, we prepared a simple profiling application that loads two classes from an APK container using `DexClassLoader` API or GNRs SecureDexClassLoader. In particular, we instrumented both the app and our library code to log initial and final time-stamps for relevant operations during the dynamic loading procedure. Since `DexClassLoader` is not able to fetch remote containers autonomously, we decided for both systems to use a code container stored on the device as the source for DCL, so to have a fair test. For this experiment, we defined two relevant test scenarios:

1. *No cached resource.* In this case, we want to simulate the first load operation performed on a source container, where the optimized version of it has not been already created and cached. From a performance point of view, this is the worst possible scenario since the system cannot rely on any cached resource but it has, instead, to prepare them for performance improvements in the next load calls. We achieved this result by erasing, between each run of the test application, all the cached resources in the private-application folders (i.e., ODEX or ELF cached files for `DexClassLoader`, and fetched remote certificates for GNR).

2. *Cached resources.* In this situation, we are addressing setup and load operations on a code container, from which classes have been loaded previously. This implies that `DexClassLoader` has already prepared and stored the optimized version of the container in the cache folder and GNR has fetched the certificate for the signature verification. Contrarily to the previous scenario, this is the best case for performance. In practice, we cached the resources and, then, started to collect time-stamps in the following executions.

Outcome We aggregate the measured time-stamps all the data by phase compute mean, median, and standard deviation for each phase. Table 5.6 and 5.7 show, respectively, the results of the performance comparison between the two studied solutions. In particular, for each of the two situations, we collected time-stamps over 100 iterations of the profiling application on a Nexus 5 device. Figure 5.1 shows the same results using bar charts, which graphically highlight the length of each separate phase.

Background Information on Participants	
Java programming experience *How long have you been developing in Java?*	
Between 0 and 1 year	0/9
More than 1 and less then 3 years	2/9
More than 3 years	7/9
Android programming experience *How long have you been developing Android apps?*	
Between 0 and 1 year	3/9
More than 1 and less then 3 years	4/9
More than 3 years	2/9
Android programming experience (2) *Which one of the following scenario is closer to your experience as an Android developer?*	
I played with Android just a couple of times in the past.	1/9
I develop apps just for fun in my spare time.	3/9
Programming in Android is a relevant part of my job and I use it on a daily base.	2/9
Other..	3/9
Java/Android expertise level *Rank between 1 (Novice) and 10 (Expert) your knowledge of the Java/Android ecosystem.*	
0 - 6	2/9
7 or 8	6/9
9 or 10	1/9

Table 5.2: Use case study: Background information on the 9 participants.

DCL with DexClassLoader API (Phase 1)	
Average time for completing Phase 1 in minutes:	170
Easy to learn *Was it easy to learn how to use DexClassLoader API given your previous knowledge of the Android framework?*	
1 or 2 (Very Difficult)	5/9
3	2/9
4 or 5 (Very Easy)	2/9
Simple to use *Did you encounter any difficulties in making your first toy app work fine? Did you succeed immediately?*	
1 or 2 (Many difficulties)	4/9
3	1/9
4 or 5 (No problem)	4/9
Official Documentation *Do you think that official Android API are good, complete, and clear enough to learn how to properly use DCL?*	
1 or 2 (Extremely Poor)	8/9
3	0/9
4 or 5 (Excellent)	1/9
Final evaluation *To sum up, provide an average mark on your satisfaction after having used DexClassLoader API.*	
1 or 2 (Disappointing)	6/9
3	0/9
4 or 5 (Excellent)	3/9

Table 5.3: Use case study: Summary of the evaluation of `DexClassLoader` API (**Phase 1**).

DCL with GNR API (Phase 2)	
Average time for completing Phase 2 in minutes:	40
Easy to set up *Was it easy to insert GNR library in the second toy project? Did you encounter issues?*	
1 or 2 (Very Difficult)	0/9
3	1/9
4 or 5 (Very Easy)	8/9
Easy to learn *Was it easy to learn how to use GNR given your previous knowledge of the Android framework and of DexClassLoader API?*	
1 or 2 (Very Difficult)	0/9
3	0/9
4 or 5 (Very Easy)	9/9
Simple to use *Did you encounter any difficulties in making your second toy app work fine? Did you succeed immediately?*	
1 or 2 (Many difficulties)	1/9
3	0/9
4 or 5 (No problem)	8/9
GNR documentation (Readme on Github + Tutorial on ReadTheDocs) *Do you think that the documentation is good, complete, and clear enough to learn how to properly use GNR API?*	
1 or 2 (Extremely Poor)	0/9
3	0/9
4 or 5 (Excellent)	9/9
Final evaluation *To sum up, provide an average mark on your satisfaction after having used GNR API.*	
1 or 2 (Disappointing)	0/9
3	0/9
4 or 5 (Excellent)	9/9

Table 5.4: Use case study: Summary of the evaluation of GNR API (**Phase 2**).

Comparison: DexClassLoader vs GNR API (Phase 3)	
GNR learning overhead *Please quantify the effort in learning how to use GNR API over native DexClassLoader API.*	
1 or 2 (Almost Zero)	9/9
3	0/9
4 or 5 (Extremely Broad)	0/9
Easy to implement *Look at the two toy apps you have prepared, which one between the two was easier to implement?*	
First toy app (DexClassLoader) was easier to implement.	0/9
Second toy app (SecureDexClassLoader) was easier to implement.	8/9
Both of them were too difficult!	1/9
Readability *Look at the two snippets of code you implemented for the apps, which one is easier to read and understand at a first glance?*	
First toy app (DexClassLoader) is way easier to read.	0/9
Second toy app (SecureDexClassLoader) is way easier to read.	7/9
They are more or less equally easy to understand.	2/9
Both of them are difficult to read!	0/9
Flexibility *Between the two analyzed solutions, which one do you think offers more flexibility and features for your Android app?*	
DexClassLoader API.	1/9
GNR API (SecureDexClassLoader).	8/9
Code maintainability *Let's say that you decide to change the remote location of your APK used as a source for DCL or that you want to add DCL also from a second remote APK, stored at a different URL. Which one of the two apps would be easier to fix?*	
First toy app (DexClassLoader) would be easier to fix.	1/9
Second toy app (SecureDexClassLoader) would be easier to fix.	7/9
The amount of work would be exactly the same for both of them.	1/9
Security *Which one between the two toy apps do you think is more secure?*	
First toy app (DexClassLoader) is more secure.	0/9
Second toy app (SecureDexClassLoader) is more secure.	8/9
They are the same, security-wise.	1/9

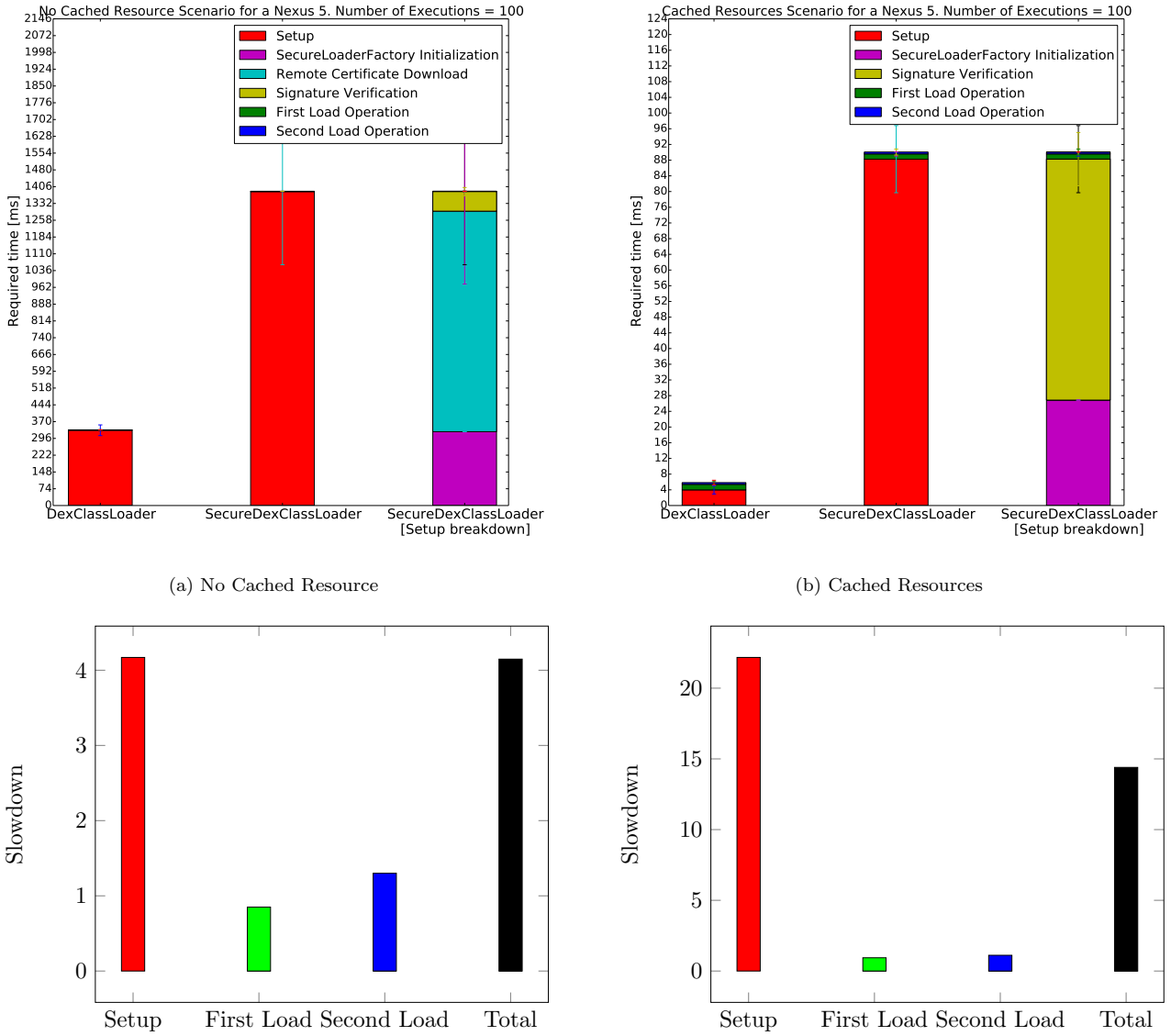
Table 5.5: Use case study: Summary of the comparative evaluation between `DexClassLoader` and GNR API (**Phase 3**).

Phase	Mean [ms]	Median [ms]	Std Deviation [ms]
DexClassLoader [Total Time]	334.20	332.00	23.50
— Setup	331.91	330.00	23.60
— First Load Operation	1.55	1.00	0.70
— Second Load Operation	0.30	0.00	0.46
SecureDexClassLoader [Total Time]	1,386.13	1,237.00	322.05
— Setup	1,384.13	1,234.00	322.09
—— Fetch Remote Certificate	972.32	822.00	321.32
—— Verify Signature	86.17	82.00	18.22
— First Load Operation	1.32	1.00	0.66
— Second Load Operation	0.39	0.00	0.60

Table 5.6: No cached resource. The table outlines mean, median, and standard deviation of each phase of the load operations, starting from the collected timestamps. The highest part of the table outlines results for `DexClassLoader` API, whereas the lowest part focuses on GNR API. All times are expressed in milliseconds. Notice that the time of the "Setup" phase of `SecureDexClassLoader` is composed by the times of "Fetch Remote Certificate," "Verify Signature," and "SecureLoaderFactory Initialization," which can be roughly estimated as a subtraction from the other phases and is not listed in this table.

Phase	Mean [ms]	Median [ms]	Std Deviation [ms]
DexClassLoader [Total Time]	6.28	6.00	1.39
— Setup	3.98	4.00	1.04
— First Load Operation	1.42	1.00	0.70
— Second Load Operation	0.44	0.00	0.61
SecureDexClassLoader [Total Time]	90.42	90.00	8.73
— Setup	88.25	87.50	8.55
—— Verify Signature	61.39	61.00	6.79
— First Load Operation	1.33	1.00	0.55
— Second Load Operation	0.49	0.00	0.77

Table 5.7: Cached resources. This table shows, in the same fashion of Table 5.6, results for the case in which both certificates and optimized versions of the code containers have been already cached. For this reason, time for fetching remote certificate is null thus not shown. Moreover, many of the other time entries are significantly smaller compared to the previous table because of the caching effect.



(a) No Cached Resource

(b) Cached Resources

Figure 5.1: Comparative bar charts on the execution time. In the two upper charts, each phase is identified by a color, the height of each bar depends on the mean of that step and the standard deviation is also represented using a thin solid line. The third bar is equal to the second bar but it highlights the breakdown of the "Setup" phase of `SecureDexClassLoader`. The lower charts show the slowdown introduced by `SecureDexClassLoader` over the original `DexClassLoader` on each different phase listed in Table 5.6 and 5.7. In (a), time of execution is higher for both solutions since neither the optimized source container is already available(bigger red bars), nor the remote certificate to fetch (cyan bar). On the other hand, in (b), having cached resources implies a reduction in execution time, as shown by a smaller red bar, and by the absence of the cyan one; the biggest overhead becomes the signature verification process, which, anyway, impacts only once during the initialization phase. Differently, load operations have always a comparable execution time.

From the results of this experiment we can conclude that, once the setup operations are completed, the difference of time implied for a load operation between the two systems is negligible. A non-obvious conclusion is that the signature verification process is indeed quite cheap (between 61 and 86 milliseconds) compared to other phases. The costly operations are: (1) the translation of the code container into an optimized version of it in `DexClassLoader` (more than 300 milliseconds), and (2) the retrieval of the remote container for the signature verification in `SecureLoaderFactory` (the "Fetch Remote Certificate" entry in the first table). In particular, while the former operation is a common denominator penalty for both the solutions, (remember that `SecureDexClassLoader` wraps native `DexClassLoader`, which performs this translation for performance improvements), the latter one is specific of GNR since `DexClassLoader` does not perform any signature verification on the code to load. Besides being expensive, fetching a remote certificate can be a troublesome bottleneck in the system because its timing depends directly on the network latency, which can be seen through the high standard deviation. However, after the setup phase, both the original and the GNR-secure solution perform the loading operations in the same time. Therefore, we conclude that the performance overhead introduced by GNR is negligible, after an initial overhead paid during the initialization step. In particular, whereas the signature verification process is executed every time that a `SecureDexClassLoader` object is initialized, the translation of the code container and the fetching of the remote certificate are executed only once, unless the cached resources are explicitly erased by the developer.

5.2 Repackaging tool validation

In this section, we present the evaluation performed to show the effectiveness of our repackaging script and we also explain the challenges that made this task hard.

5.2.1 Patching sample applications

Goal First, we want to show that the script always terminates and substitutes all the sensitive points successfully with a valid snippet of code using the `RepackHandler` class and the GNR API. Secondly, we want to show that the patched application is retained fully functional.

Setup For this experiment, we developed a simple application that uses DCL on a source container. We also selected a corpus of 15 applications fetched from the VirusTotal [17] dataset. We opted for VirusTotal since it can look for APKs with specific features: we retrieved *benign* APKs that were known for using dynamic code loading. We manually inspected each resulting APK.

Evaluation procedure For each APK, we apply the following procedure:

1. Install the target APK on an emulator and interact with the application to trigger the DCL functionality and save a copy of the code containers used as sources for DCL from the emulator.

Listing 5.1: Output produced by an `analysis.show_DynCode()` call on the original APK.

```
1 Lcom/google/android/gms/internal/j;->e(Landroid/content/Context;)V (0xa8) --->
     Ldalvik/system/DexClassLoader;-><init>(Ljava/lang/String; Ljava/lang/String;
     Ljava/lang/String; Ljava/lang/ClassLoader;)V
1 Lcom/google/android/gms/internal/j;->e(Landroid/content/Context;)V (0xbe) --->
     Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
1 Lcom/google/android/gms/internal/j;->e(Landroid/content/Context;)V (0xd6) --->
     Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
1 Lcom/google/android/gms/internal/j;->e(Landroid/content/Context;)V (0xee) --->
     Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
1 Lcom/google/android/gms/internal/j;->e(Landroid/content/Context;)V (0x106) --->
     Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
1 Lcom/google/android/gms/internal/j;->e(Landroid/content/Context;)V (0x11e) --->
     Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
1 Lcom/google/android/gms/internal/j;->e(Landroid/content/Context;)V (0x136) --->
     Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
1 Lcom/qq/e/v2/managers/plugin/PM;-><init>(Landroid/content/Context;)V (0x8c) --->
     Ldalvik/system/DexClassLoader;-><init>(Ljava/lang/String; Ljava/lang/String;
     Ljava/lang/String; Ljava/lang/ClassLoader;)V
```

2. When a code container is retrieved successfully, analyze and reverse it to gain knowledge of the certificate, if any, which was used to sign it and the name of the classes inside of it, which, once again, may be obfuscated.

3. With the knowledge gained from the previous three points, set up the configuration options of the repackaging script and then run it on the target APK. Check whether the script terminates with no error.

4. Inspect the patched APK against the original one with Androguard and verify that all the sensitive points have been substituted correctly. Listing 5.1 presents the output produced by the `analysis.show_DynCode()` method on an original tested APK, whereas Listing 5.2 shows the result of the same call on the correctly patched version.

5. Finally, install the patched APK on the emulator and trigger the DCL functionality again to verify that the application does not crash and possibly execute the loading operation as required.

Outcome At first, we patched our example application. The results were encouraging since the tool both substitutes all the sensitive points and it kept the patched APK fully-working. Later, we performed the same test but this time using a code container, whose signature was broken, as the source container for DCL. Also in this case, our tool performed the repackaging correctly: In fact, while in the original application the DCL operation was executed, in the patched version GNR correctly detected the incorrect signature of the code container, used as source, and prevented the DCL operation from happening.

On the other hand, after a first analysis on the corpus of 15 benign APK, we found out that all of them used DCL to include third-party ads library to inject banner in their `Activity` classes. In particular, 14 apps embeded the Google Mobile Ads SDK (package name: `com.google.android.ads`), whereas one used a different library (package name: `com.jnm.adlivo.androidsdk`). Our script terminates with errors on 2 of these applications because of an internal exception

Listing 5.2: Output produced by the same call on the APK patched by the repackaging tool. Since this call traces all the invocation of `DexClassLoader` API, the fact that none of the sensitive points, detected in Listing 5.1, is present anymore is a proof that our script detects all sensitive points and substitute them with snippets of code using GNR API. The result of this is that the only invocations to `DexClassLoader` API are the ones included in our `SecureDexClassLoader` class.

```
1 Lit/necst/grabnrun/SecureDexClassLoader;-><init>(Ljava/lang/String; Ljava/lang/String;
     Ljava/lang/String; Ljava/lang/ClassLoader; Landroid/content/ContextWrapper; Z)V (0x1a)
     ---> Ldalvik/system/DexClassLoader;-><init>(Ljava/lang/String; Ljava/lang/String;
     Ljava/lang/String; Ljava/lang/ClassLoader;)V
1 Lit/necst/grabnrun/SecureDexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
     (0x9a) --->
     Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
1 Lit/necst/grabnrun/SecureDexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
     (0x108) --->
     Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
1 Lit/necst/grabnrun/SecureDexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
     (0x1f8) --->
     Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
```

raised by apktool while decoding the resources of their APK. As regards the remaining 13 applications, all the sensitive points were properly substituted. However, none of the applications injected banner classes at runtime anymore because they were all using unsigned JAR containers as sources for DCL and, thus, GNR prevented the execution of those loading operations.

Roadblocks As the previous results showed, there may be some peculiar cases, where our tool is not able to patch completely an incoming APK. This outcome can be explained because of different roadblocks:

- **Impossibility to find package names of the source containers**. Since the repackaging tool needs to set up an associative map, linking package names to certificates' remote URL to initialize `SecureDexClassLoader` instances, it is important to know all the package names of the source containers prior to the execution of the script. Unfortunately, this may become a problem in case that developers decide to use tools, like Proguard, to obfuscate their JAR libraries used as sources for DCL.

- **Impossibility to find a valid copy of the code containers**. An alternative way to avoid the previous issue is providing the repackaging tool with a valid copy of the code containers used by the patched application. With such an input, the tool can extract package names automatically from a container and link them to the proper certificate. However, also this requirement sometimes can be difficult to accomplish (e.g., the application may erase the code container immediately after the load operation, or the archive may be fetched only in unlikely conditions, difficult to trigger in the application).

- **Impossibility to trigger the dynamic loading behavior at run time**. In some applications that we tested, it was impossible to trigger the

dynamic loading behavior, while executing the application on the emulator. This is a well-known problem in the security research field, especially for the detection of malicious behaviors of malwares using dynamic analysis. In our scenario, this can be troublesome because it usually prevents the tested app from fetching the remote code containers for DCL.

- **Source containers are not signed**. This situation happens whenever the patched application uses as sources JAR containers, which were not signed by the library developer. Incurring in such a situation implies that, although fully-working, the patched app will prevent these containers from being loaded at runtime. This issue can be solved easily by requiring developers to sign their code containers before publishing them, as they are accustomed to do for static code.

Not all of these roadblocks prevented our script from finishing its execution or from patching all the sensitive points, but they leaded to not fully-working applications after the patch was applied. Anyway, it is important to underline that these roadblocks occurred when we tried to patch unknown applications. In particular, none of them, except the last one, should be a relevant issue for a developer, who wants to patch his own application: This developer, in fact, knows exactly how the app works, therefore he can provide easily all the pieces of information needed for the operation (i.e., the package names or the code containers, the locations of the remote certificates).

Chapter 6

Limitations and future works

6.1 Limitations

After having presented both GNR library and the repackaging tool, in this chapter we outline the main limitations of our current solution.

Firstly, at the current implementation stage, we realized our library to associate only one source container per package name. This means that, if a developer decides to create an instance of `SecureDexClassLoader` with two source archives that contain an equal package name, attempting to load a class with this specific package name, will generate an unpredictable behavior since `SecureDexClassLoader` associates that package name with just one of the two containers. Although this may seem a strong limitation, indeed it is not: APK containers, in fact, requires a unique package name for being published on the Google App Store therefore this issue is reduced only to JAR library containers. We considered having two source JAR containers with one, or more, identical package name a rare possibility and that is why we leave this check on package names conflicts as a responsibility for the applications' developers.

Secondly, our remote protocol accepts as legitimate for signature verification self-signed certificates as long as they are formally valid and not expired. This design choice derives from what Google suggests in its overview on application security [1], which is trusting and using self-signed certificate for code signing, since "Android currently does not perform CA verification for application certificates". For this reason, given that the use of self-signed certificate is the common practice for signing in Android [16], the current implementation of GNR does not take into account certificate chain validation during signature verification.

Finally, let us consider the repackaging tool. As it was stated at the end of Section 5.2, the real issue is finding a good method to evaluate the complete effectiveness of our script. In fact, the evaluation that we applied in this work was able to prove that the repackaging script terminates and substitutes all the sensitive points of the original APK, but it didn't show, except for our test application, that applications patched by our tool are still fully-working at the end of the process. However, it is conceptually very hard to write a testing

oracle that tells whether an arbitrary, unspecified program has computed the function that the developer wrote it for.

6.2 Future works

In addition to addressing the aforementioned limitations, we foresee some future directions for our work.

As a first future work, after having released our library and the repackaging tool, we are thinking about implementing a plug-in for Android Studio (AS). The aim of this tool would be to assist developers by warning them when the native, unsafe `DexClassLoader` API are used and, in case, by managing the automatic refactoring of all their snippets of code to port them to use GNR API. An obstacle to the immediate realization of this plug-in is that, since Android Studio has just recently become the new official IDE for applications' development, resources for the implementation of AS plug-in are few and modest, at the moment, and so we prefer to wait for a more stable release, before designing and realizing this plug-in.

Moving back to GNR, we reasoned about extending it to wrap other native API for DCL. More in the details, including `PathClassLoader` in our library should be a quite naïve task, since this API works in a similar way to `DexClassLoader` and, therefore, reapplying the very same verification protocol should be sufficient. On the other hand, `android.content.Context.createPackageContext` would probably require a completely different study because the functioning of this API varies significantly from the `DexClassLoader` case and, thus, it will be necessary to design an ad-hoc solution.

We may also introduce further improvements under the performance point of view: In particular, by storing the results of previously executed signature verifications into an application-private folder, the library can avoid to perform an extra signature verification every time that `SecureDexClassLoader` attempts to load a class from an already-imported source container, which has been already evaluated in the past. Such a performance enhancement would impact by removing the yellow bar in the cached resources scenario, as presented in the right bar chart of Figure 5.1.

Finally, to obtain a more complete and useful evaluation of the repackaging tool, we may think about performing a different use case study: We may try to collect a set of users, who has already developed an Android application that makes use of `DexClassLoader` API and ask each one of them to patch their own application with our repackaging tool to obtain a final working APK using GNR API. In this way, we could benefit from the experience of each developer to properly tune the settings of the script and, thus, avoid almost all the challenges presented in Section 5.2. With this, it should be way easier to evaluate whether or not our tool keeps patched APK fully working.

Chapter 7

Conclusions

In this thesis we presented GNR, an easy-to-use, drop-in Java library for Android applications that helps developers to make remote DCL secure by default. GNR is based on a protocol that we proved to be secure not only in the initial threat model, where a MITM attacker exploits security vulnerabilities introduced by benign applications that do not validate dynamically-loaded code properly, so as to execute arbitrary code, but also to an extension of it, where the attacker is able to compromise the server that hosts the code to load. At the current stage of implementation, the library wraps native `DexClassLoader` API and enhances it both in functionalities (e.g., it handles fetching of remote containers automatically) and, especially, in security since it prevents inexperienced or inaccurate developers from introducing security vulnerabilities in their applications because of insecure code fetching, storage, or lack of integrity and authentication checks.

Compared to previous related research works, our solution has the remarkable advantage of being really easy to deploy since it is a simple library that can be plugged into any Android project. This is a plus since, while providing a sufficient level of security against the main threats outlined in Chapter 2, our library is an intermediate solution that does not require complex setup cost, like a partial rewriting of the Android framework and a consequent update of the operative system for all the devices. Moreover, our solution results easy to adopt for all the main actors involved in the process: Google does not need to modify anything in the current Android framework, application developers need just to remember to import and use GNR in their projects, and third-party library developers must remember to sign their code and make available on a domain under their control the trusted certificate for the signature verification.

Along with this library, we have implemented a repackaging tool, in the form of a Python script, which takes as an input a not obfuscated APK using `DexClassLoader` API and some user settings, and automatically patches it by enforcing the use of GNR API, without the need for the source code or for the developer to write even a single line of code.

We evaluated the performance of GNR by measuring in a profiling test application the overhead introduced by our library over native `DexClassLoader` API: The results are encouraging since, except for a one-time penalty during setup phase, which can be significantly reduced by the caching strategies of GNR, the overhead introduced by our library on load operations is negligible

compared to the native solution. We also setup an use case study on 9 Android developers to prove that: (1) GNR is an effective and useful solution since it removes by design all the security vulnerabilities that these developers inserted in their code inadvertently, while using native `DexClassLoader` API (i.e., 6 of them failed in fetching code in a safe way by using HTTP connections instead of HTTPS ones, 4 of them failed by storing the code in a world-writable location, and all of them forgot to implement custom integrity checks on the fetched code); (2) GNR requires only a little learning effort for a developer, who already knows native API (indeed, all the participants quantify this learning effort as "little" or "almost null"); (3) GNR is a more powerful, flexible, and easier to use solution than `DexClassLoader` (developers agreed also on this point since 7 of them state that our solution, in the proposed experiment, would be easier to maintain and it is simpler to read; whereas 8 of them find our library easier to use, more flexible, and more secure than native `DexClassLoader` API).

Our personal hope is that this work will highlight the need for the Android project and, thus, for Google to impress a significant change: In fact, although DCL is a useful tool for Android app developers, such that even Google itself makes use of it in its libraries (e.g., Google Mobile Ads), native API for DCL are difficult to understand and use, and it is even easy for a developer to introduce inadvertently serious security vulnerabilities. Moreover, compared to the average high-quality standard provided by the Android project, documentation for these API is extremely poor and no tutorial is present at all. This is both a pity, given the usefulness of the feature, and a security concern and that is why we hope that this thesis, along with similar research works, may sensitize Google to provide a revised version of native API for DCL, which should be more secure, easier to use, better documented, and, why not, maybe partially inspired on what we have developed for GNR.

Bibliography

[1] Android. *Application Signing in Android*. URL: https://source.android.com/devices/tech/security/overview/app-security.html#application-signing.

[2] Android. *ART and Dalvik*. URL: http://source.android.com/devices/tech/dalvik/index.html.

[3] Android. *DexClassLoader API Reference Page*. URL: http://developer.android.com/reference/dalvik/system/DexClassLoader.html.

[4] *Android 5.0 Behavior Changes*. Oct. 2014. URL: http://developer.android.com/about/versions/android-5.0-changes.html.

[5] Desnos Anthony. *Androguard, Project Home Page*. URL: https://code.google.com/p/androguard/.

[6] Brut.alll. *Apktool, Project Home Page*. URL: https://code.google.com/p/android-apktool/.

[7] International Data Corporation. *Smartphone OS Market Share, Q4 2014*. URL: http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[8] Java Oracle Documentation. *Example Code for JAR Signature Verification*. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/MyJCE.java.

[9] Java Oracle Documentation. *Jar File Specification: Signed Jar File*. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html#Signed_JAR_File.

[10] Luca Falsina, Federico Maggi, and Yanick Fratantonio. *Grab'n Run Repository on GitHub*. Nov. 27, 2014. URL: https://github.com/lukeFalsina/Grab-n-Run.

[11] Michael Grace et al. "Unsafe Exposure Analysis of Mobile In-App Advertisements". In: *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks(WiSec)*. 2012, 101112. ISBN: 9781450312653.

[12] Wenhui Hu et al. "Duet: Library Integrity Verification for Android Applications". In: *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks(WiSec)*. 2014, pp. 141–152. ISBN: 9781450329729.

[13] Joshua J.Drake et al. *Android Hacker's Handbook*. Ed. by Wiley. 2014. ISBN: 9781118608647.

[14] OpenSignal. *Android Fragmentation Visualized*. Aug. 2014. URL: http://opensignal.com/assets/pdf/reports/2014_08_fragmentation_report.pdf.

[15] Sebastian Poeplau et al. "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2014, 2326. ISBN: 1891562355.

[16] *Signing Considerations for Android Developers*. URL: https://developer.android.com/tools/publishing/app-signing.html#considerations.

[17] VirusTotal team. *VirusTotal, Project Home Page*. URL: https://www.virustotal.com.

[18] T Vidas and N Christin. "Sweetening Android Lemon Markets: Measuring and Combating Malware in Application Marketplaces". In: *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2013.

Appendices

Appendix A

Implementation details of relevant parts

This appendix reports the full Java implementation of two key methods for signature verification, presented in Subsection 4.2.2, while explaining the internals of GNR library.

A.1 Verify container signature against a trusted certificate

```
// Given the path to a jar/apk container and a valid certificate
// instance this method returns whether the container is signed
// properly against the verified certificate.
private boolean verifyContainerSignatureAgainstCertificate(String containerPath,
     X509Certificate verifiedCertificate) {

   // Check whether the selected resource is a jar or apk container
   int extensionIndex = containerPath.lastIndexOf(".");
   String extension = containerPath.substring(extensionIndex);

   boolean signatureCheckIsSuccessful = false;

   // Depending on the container extension the process for
   // signature verification changes
   if (extension.equals(".apk")) {

      // APK container case:
      // At first look for the certificates used to sign the apk
      // and check whether at least one of them is the verified one..

      PackageInfo mPackageInfo = mPackageManager.getPackageArchiveInfo(containerPath,
           PackageManager.GET_SIGNATURES);

      if (mPackageInfo != null) {

         // Use PackageManager field to retrieve the certificates
         // used to sign the apk.
         Signature[] signatures = mPackageInfo.signatures;

         if (signatures != null) {
            for (Signature sign : signatures) {
               if (sign != null) {

                  X509Certificate certFromSign;
                  InputStream inStream = null;

                  try {

                     // Recreate the certificate starting from
                     // this signature.
                     inStream = new ByteArrayInputStream(sign.toByteArray());
                     certFromSign = (X509Certificate)
                          certificateFactory.generateCertificate(inStream);

                     // Check that the reconstructed certificate
                     // is not expired..
                     certFromSign.checkValidity();

                     // Check whether the reconstructed certificate
                     // and the trusted one match.
                     // Please note that certificates may be self-signed
                     // but it's not an issue..
                     if (certFromSign.equals(verifiedCertificate))
                        // This a necessary but not sufficient condition
                        // to prove that the apk container has not been
                        // repackaged..
                        signatureCheckIsSuccessful = true;

                  } catch (CertificateException e) {
                     // If this branch is reached certificateFromSign
                     // is not valid..
                  } finally {
                     if (inStream != null) {
                        try {
                           inStream.close();
                        } catch (IOException e) {
                           e.printStackTrace();
                        }
```

```
                }
             }
          }
       }
    } else {
       Log.i(TAG_SECURE_DEX_CLASS_LOADER, "An invalid/corrupted signature is associated
            with the source archive.");
    }
 } else {
    Log.i(TAG_SECURE_DEX_CLASS_LOADER, "An invalid/corrupted container was found.");
 }
}

// JAR container OR successfull APK container case:
// This branch must be taken by all JAR containers and by those APK
// containers, whose certificates list contains also the trusted
// verified certificate.
if (extension.equals(".jar") || (extension.equals(".apk") && signatureCheckIsSuccessful))
      {

   // Verify that each entry of the container has been signed properly
   JarFile containerToVerify = null;

   try {

      containerToVerify = new JarFile(containerPath);
      // This method will throw an Exception whenever
      // the JAR container is not signed with the trusted certificate
      // N.B. APK are an extension of JAR containers..
      verifyJARContainer(containerToVerify, verifiedCertificate);

      // No exception raised so the signature
      // verification succeeded
      signatureCheckIsSuccessful = true;

   } catch (Exception e) {
      // Signature process failed since it triggered
      // an exception (either an IOException or a SecurityException)
      signatureCheckIsSuccessful = false;

   } finally {
      if (containerToVerify != null) {
         try {
            containerToVerify.close();
         } catch (IOException e) {
            e.printStackTrace();
         }
      }
   }
}

   return signatureCheckIsSuccessful;
}
```

A.2 Verify a JAR container signature

```
private void verifyJARContainer(JarFile jarFile, X509Certificate trustedCert) throws
     IOException {

   // Sanity checking
   if (jarFile == null || trustedCert == null)
        throw new SecurityException("JarFile or certificate are missing");

   Vector<JarEntry> entriesVec = new Vector<JarEntry>();

    // Ensure the JAR file is at least signed.
    Manifest man = jarFile.getManifest();
    if (man == null) {
      Log.i(TAG_SECURE_DEX_CLASS_LOADER, jarFile.getName() + "is not signed.");
      throw new SecurityException("The container is not signed");
    }

    // Ensure all the entries' signatures verify correctly
    byte[] buffer = new byte[8192];
    Enumeration<JarEntry> entries = jarFile.entries();

    while (entries.hasMoreElements()) {

      // Current entry in the JAR container
       JarEntry je = (JarEntry) entries.nextElement();

       // Skip directories.
       if (je.isDirectory()) continue;
       entriesVec.addElement(je);
       InputStream inStream = jarFile.getInputStream(je);

      // Read in each JAR entry. A security exception will
      // be thrown if a signature/digest check fails.
      while (inStream.read(buffer, 0, buffer.length) != -1) {
          // Don't care as soon as no exception is raised..
      }

      // Close the input stream
      inStream.close();
    }

    // Get the list of signed entries from which certificates
    // will be extracted..
   Enumeration<JarEntry> signedEntries = entriesVec.elements();

   while (signedEntries.hasMoreElements()) {

      JarEntry signedEntry = (JarEntry) signedEntries.nextElement();

      // Every file must be signed except files in META-INF.
      Certificate[] certificates = signedEntry.getCertificates();
      if ((certificates == null) || (certificates.length == 0)) {
         if (!signedEntry.getName().startsWith("META-INF")) {
            Log.i(TAG_SECURE_DEX_CLASS_LOADER, signedEntry.getName() + " is an unsigned
                 class file");
            throw new SecurityException("The container has unsigned class files.");
         }
      }
      else {
         // Check whether the file is signed by the expected
         // signer. The JAR may be signed by multiple signers.
         // So see if one of the signers is 'trustedCert'.
         boolean signedAsExpected = false;

         for (Certificate signerCert : certificates) {

            try {
               ((X509Certificate) signerCert).checkValidity();
            } catch (CertificateExpiredException
                 | CertificateNotYetValidException e) {
               // Usually expired certificate are not such a relevant
```

```
            // issue; nevertheless on Android a common practice is
            // using certificates (even self signed) but with
            // at least a long life span and so temporal validity
            // should be enforced..
            Log.i(TAG_SECURE_DEX_CLASS_LOADER, "One of the certificates used to sign " +
                  signedEntry.getName() + " is expired");
            throw new SecurityException("One of the used certificates is expired!");
          } catch (Exception e) {
            // It was impossible to cast the general certificate
            // into an X.509 one..
          }

          if (signerCert.equals(trustedCert))
            // The trusted certificate was used to sign this entry
            signedAsExpected = true;
        }

        if (!signedAsExpected) {
          Log.i(TAG_SECURE_DEX_CLASS_LOADER, "The trusted certificate was not used to sign
                " + signedEntry.getName());
          throw new SecurityException("The provider is not signed by a trusted signer");
        }
      }
    }
}
```