# *Grab'n Run*

## *Secure and Practical Dynamic Code Loading for Android Applications*


**Luca Falsina,**
**Yanick Fratantonio,**
**Stefano Zanero,**
**Christopher Kruegel,**
**Giovanni Vigna,**
**Federico Maggi**

# Dynamic code loading (DCL)

- Android system allows applications to *load additional code* from external sources at **runtime**.
- *Sources* can come from either:
  - a **local** code container stored on the device.
  - a **remote** URL location (first fetch, then execute the code).

**Minimize code memory footprint**

**Flexible code reuse (one library - many apps)**

**Live code update during execution**

# How to perform DCL in Android?

There are several ways to **load code dynamically** in **Android** (*Class Loaders, Context.createPackageContext, System.exec*).

In this presentation I will just focus on **Class Loaders**:

- Java objects allowing programs to load additional classes.

Added in API level 1

public class

PathClassLoader

extends BaseDexClassLoader

java.lang.Object
↳ java.lang.ClassLoader
↳ dalvik.system.BaseDexClassLoader
↳ dalvik.system.PathClassLoader

A ClassLoader implementation that operates on a list of files and directories in the **local file system**.

Added in API level 3

public class

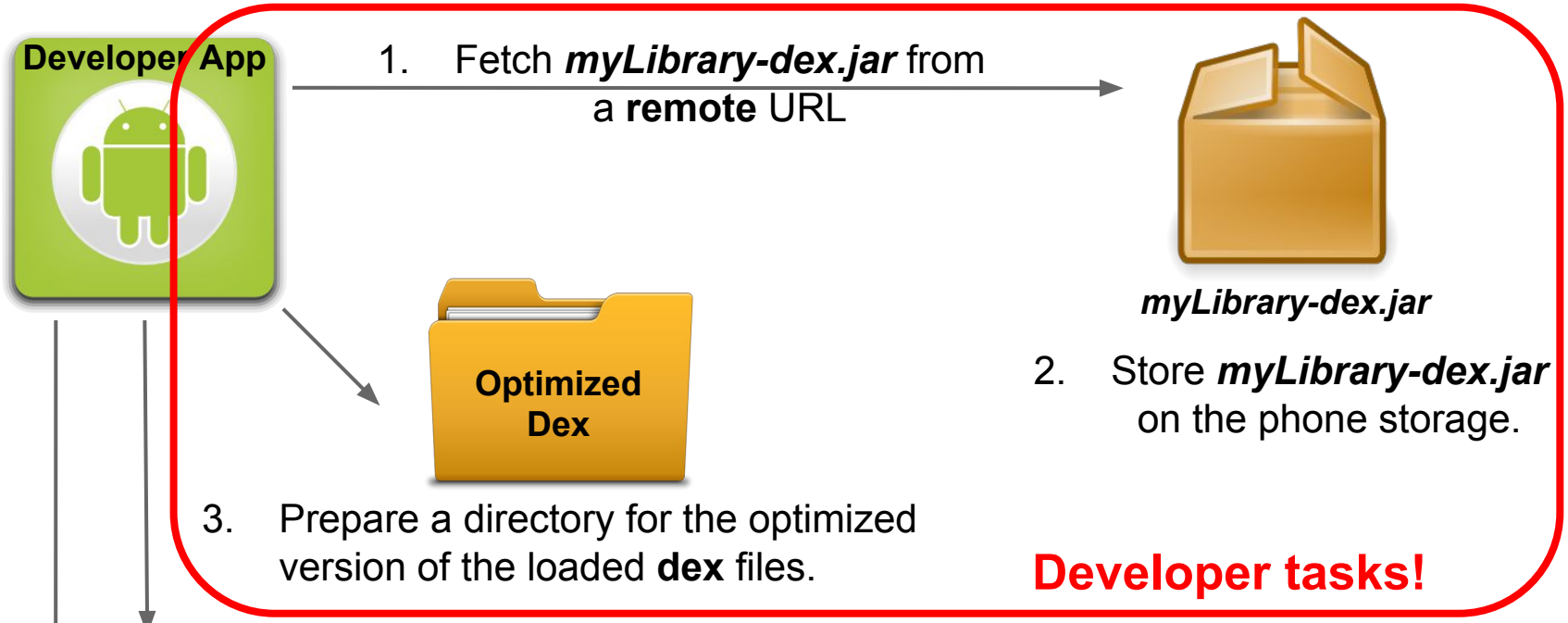
DexClassLoader

extends BaseDexClassLoader

java.lang.Object
↳ java.lang.ClassLoader
↳ dalvik.system.BaseDexClassLoader
↳ dalvik.system.DexClassLoader

A ClassLoader that loads classes from JAR and APK containers with a **classes.dex** entry.

# Typical remote DCL workflow using DexClassLoader API




4. Instantiate *DexClassLoader* object:

```
DexClassLoader mDexClassLoader = new DexClassLoader(    myLibJarPath,
                                                        dexOutputDir.getAbsolutePath(),
                                                        null,
                                                        getClass().getClassLoader());
```

5. Load *"com.example.ClassA"* class (by looking at the **optimized dex** files).

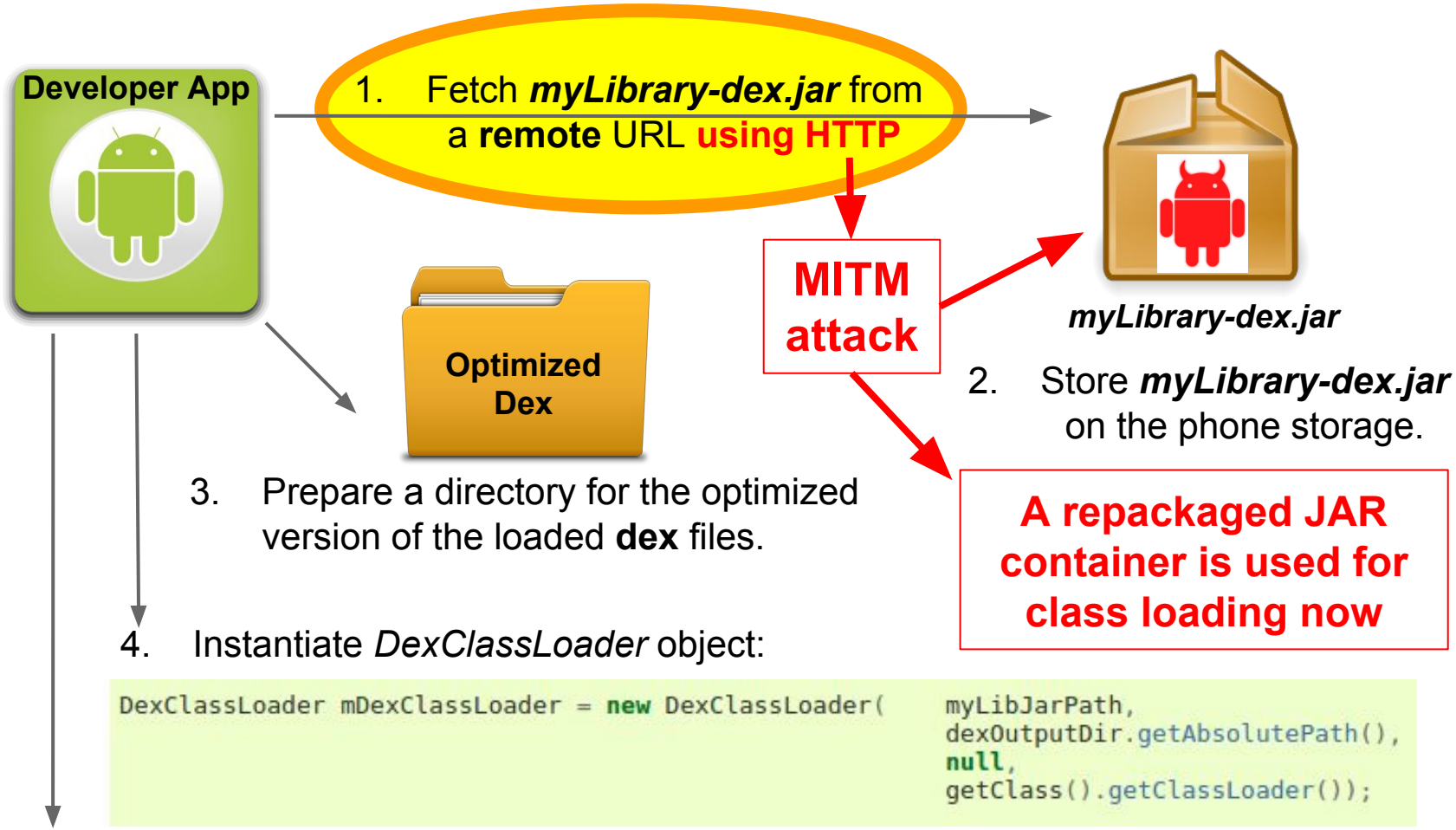
```
Class<?> loadedClass = mDexClassLoader.loadClass("com.example.ClassA");
```

# How to perform DCL badly in Android?

Here are the mistakes that a benign developer may introduce and make DCL completely **insecure:**

1. Developer fails to *retrieve* code in a *safe way*.
2. Developer fails to *store* code...
   a. ...by *saving the container* in a **modifiable** location.
   b. ...by *saving cached* ***dex*** *classes* in a **modifiable** location.

# 1. Developer fails to ***retrieve*** code in a *safe way*




```
DexClassLoader mDexClassLoader = new DexClassLoader(    myLibJarPath,
                                                        dexOutputDir.getAbsolutePath(),
                                                        null,
                                                        getClass().getClassLoader());
```

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

# 2a. Developer fails to ***store*** code in a *safe way*

## ..by *saving the container* in a **modifiable** location


1. Fetch ***myLibrary-dex.jar*** from a **remote** URL **using HTTPS properly.**

***myLibrary-dex.jar***


**Overwrite JAR before loading**

2. Store ***myLibrary-dex.jar*** on the phone storage **in a world-writable location (i.e. external storage)**.

**Code Injection**

3. Prepare a directory for the optimized version of the loaded **dex** files.

4. Instantiate *DexClassLoader* object:

```
DexClassLoader mDexClassLoader = new DexClassLoader(    myLibJarPath,
                                                        dexOutputDir.getAbsolutePath(),
                                                        null,
                                                        getClass().getClassLoader());
```

5. Load *"com.example.ClassA"* class (by looking at the **optimized dex** files)

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

# 2b. Developer fails to ***store*** code in a *safe way*

## ..by *saving cached dex* in a **modifiable** location


1. Fetch ***myLibrary-dex.jar*** from a **remote** URL **using HTTPS properly.**

***myLibrary-dex.jar***

**Code Injection**

2. Store ***myLibrary-dex.jar*** on the phone storage **in an application-private location**.

3. Prepare a directory **in a world-writable location** for the optimized version of the loaded **dex** files.

**Overwrite DEX classes before loading**

4. Instantiate *DexClassLoader* object:

```
DexClassLoader mDexClassLoader = new DexClassLoader(    myLibJarPath,
                                                        dexOutputDir.getAbsolutePath(),
                                                        null,
                                                        getClass().getClassLoader());
```

5. Load *"com.example.ClassA"* class (by looking at the **optimized dex** files)

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

# How frequent are these errors?

*Poeplau et al.* prove that DCL is heavily used in Android and that implementing DCL in a *secure* way is indeed challenging.

Considering the **top 50** free applications in August 2013:

- 31 (62%) of them make use of **DCL**
- 17 (34%) of them make use of **ClassLoaders**
- 8 (**16%**) of them were **flagged as vulnerable**

**DynamicLoadApk**, an open-source project to simplify the use of DCL (over 2000 stars, and 1000 forks on GitHub).

- Source containers are loaded from external storage.
- No integrity or authentication checks on containers.

# Threat model

Attacker's goal is *executing arbitrary code within the context of a target benign application*.

Attacker can exploit *vulnerabilities* in:

- **Device**. App execution with non-root privileges, read/write access on storage
- **Network communication channel**. MITM on all unencrypted connection
- **Remote server**. We assume the attacker is not able to compromise at least *one* server

# Goals

1. Design a **code-verification protocol** suitable to secure DCL.
2. Implement it as a *practical*, developer-friendly **Java library** to replace native API.
3. **Tradeoff** between efficiency, security, and simplicity.
4. Help developers to **migrate** existing applications effortlessly from the native API for DCL to the proposed one.

# Workflow to explain our code verification protocol


1. Fetch a **signed** copy of ***myLibrary-dex.jar*** from a **remote** URL **(HTTP / HTTPS).**

***myLibrary-dex.jar***


2. Store ***myLibrary-dex.jar*** on the phone storage **in an application-private dir**.

3. Save the optimized version of the loaded **dex** files **in an application-private dir**.

4. Retrieve developer certificate via **HTTPS (TRUSTED ELEMENT)** and store it **in an application-private dir**.


5. Verify the signatures of **all** the file entries of ***myLibrary-dex.jar*** against the **trusted certificate**.

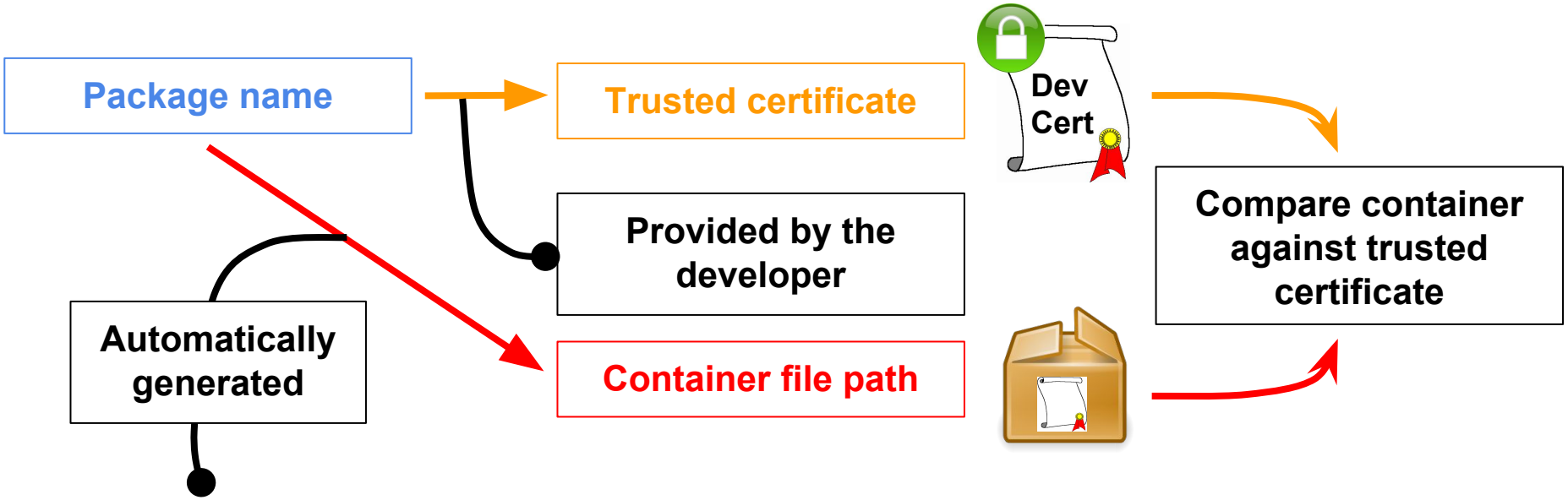
+


Load *"com.example.ClassA"* class.

Discard ***myLibrary-dex.jar*** copy in the application-private folder.

# How to associate containers and certificates?

Link **package name** to the *trusted certificate* and the *container location*.




Given a certain **container (i.e., APK, JAR)**, it is possible to retrieve the **package name(s)** associated to its classes.

# Features' Comparison Table

| DexClassLoader (Native API) | Features | SecureDexClassLoader (GNR API) |
|---|---|---|
| ✗ | **Fetch code from remote URL** | ✓ |
| ✗ | **Store code in app-private location** | ✓ |
| ✗ | **Container verification** *(integrity* and *developer authentication)* | ✓ |
| ✓ | **Dynamic code loading** | ✓ |

# SecureDexClassLoader API

**Listing 1:** DexClassLoader **code snippet.**

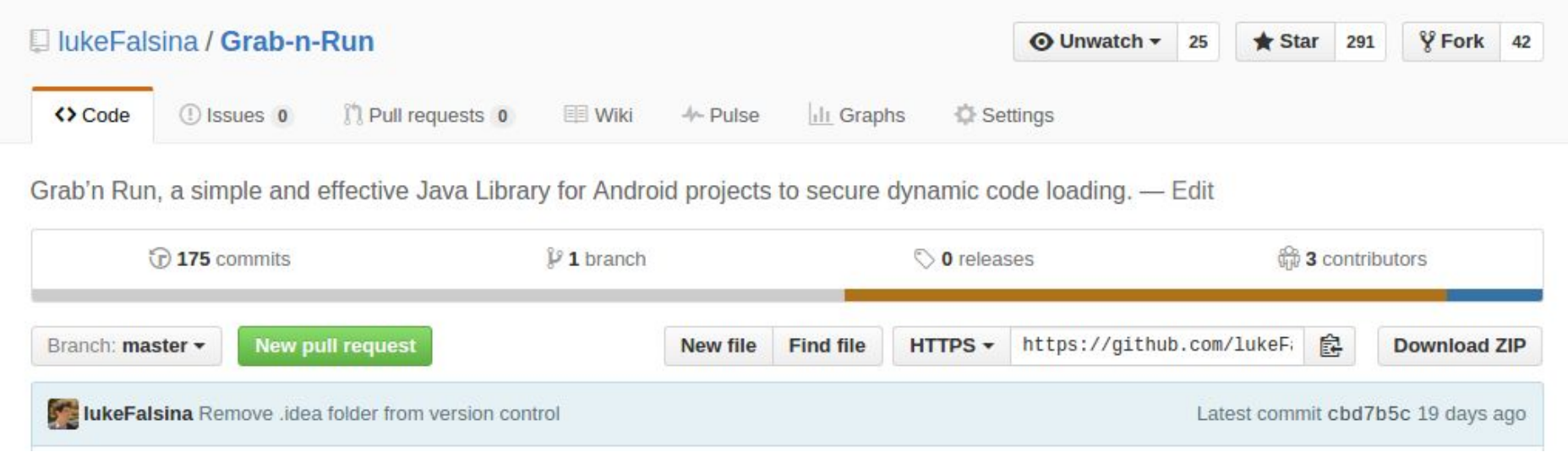
```
/*
 * Omitted steps, as they are ‘‘up to the developer’’:
 *
 *   - retrieval of the code container
 *   - storage of the container to jarContainerPath
 *   - creation of the dexOuptutDirPath
 */
DexClassLoader loader = new DexClassLoader(
   jarContainerPath, dexOutputDirPath, null, getClassLoader());

Class<?> klass = loader.loadClass("com.example.MyClass");
MyClass obj = (MyClass) klass.newInstance();
```

**Listing 2:** SecureDexClassLoader **code snippet.**

```
Map<String, URL> pToCert = new HashMap<String, URL>();
pToCert.put("com.foo", new URL("https://bar.com/cert.pem"));

SecureLoaderFactory factory = new SecureLoaderFactory(this);
SecureDexClassLoader loader = factory.createDexClassLoader(
    "http://something.com/dev/exampleJar.jar",
    null, getClassLoader(), pToCert);

Class<?> klass = loader.loadClass("com.example.MyClass");

if (klass != null) // Is signature valid?
  MyClass obj = (MyClass) klass.newInstance();
```

# Grab’n Run: Release

- We released *Grab’n Run* as a **open-source** library on *Github*.
- The project is online from *November 2014* at [www.grabnrun.org](http://www.grabnrun.org) and it is receiving attention from the community of the Android developers :)
- At the current state, it wraps and enhances *DexClassLoader* API.



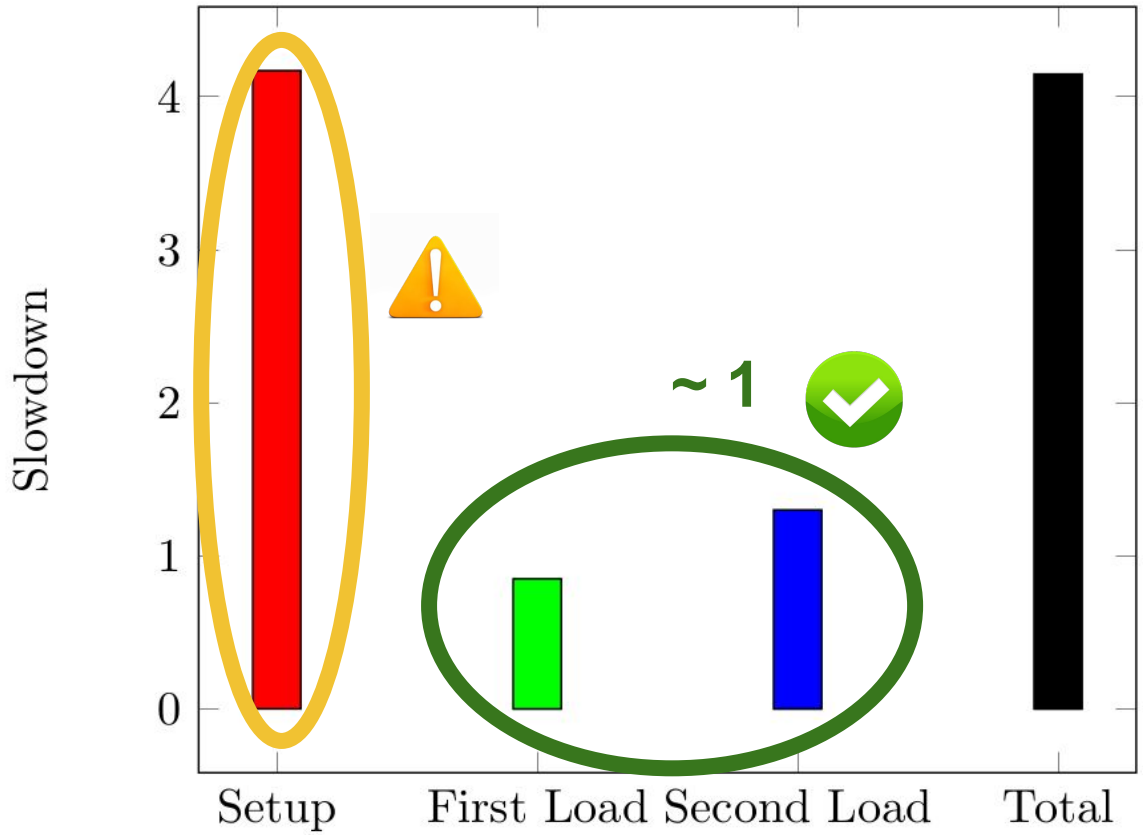
# Performance overhead

We measured the execution time of a simple profiling app using both *DexClassLoader* and GNR API for **local** DCL. We collected timestamps of 100 executions of this app on a Nexus 5.

| 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 |

# Performance overhead

$$Slowdown = \frac{Time\ of\ execution\ with\ GNR}{Time\ of\ execution\ with\ DexClassLoader}$$



# User study

We performed a user study on 12 Android developers.

Starting from a skeleton application, developers had to fetch a remote code container, store it on the phone, and load dynamically a class inside of it.

- **Phase 1**: Implement it using *DexClassLoader* API.
- **Phase 2**: Implement it using GNR API (*).
- **Phase 3**: Send their code and fill in a comparative survey on the two solutions.

(*) Not required to setup the certificate used to validate the code, nor the endpoint to store it.

Developers could consult any online resources (including the *official* Android documentation)

We explicitly asked the participants to treat this experiment as if they were adding a functionality to their own, very popular real application with millions of users.

| **Error** (***Triggering example***) | **% developers** |
|---|---|
| Fetch code in an unsafe way (*Use HTTP connection instead of HTTPS*) | 75.0% (9/12) |
| Store code in a world-writable area (*Save code container on external storage*) | 50.0% (6/12) |
| Store code in a world-writable area (*Wrongly initialize optimized cache folder*) | 00.0% (0/12) |
| Miss or fail to implement security checks (*Do not implement any custom integrity check*) | 100.0% (12/12) |

## Users evaluation: `DexClassLoader` (Native API) vs `SecureDexClassLoader` (Grab 'n Run API)

| **Average time for development**<br>*Average of the times in minutes that each developer declared as required to implement the DCL functionality:* | |
|---|---|
| Using `DexClassLoader` (Native API) | 139 min |
| Using `SecureDexClassLoader` (Grab 'n Run API) | 37 min |

| **Final evaluation of `DexClassLoader`**<br>*Please provide an average mark on your satisfaction after having used `DexClassLoader` (Native API).* | |
|---|---|
| 1 or 2 (Disappointing) | 6/12 |
| 3 | 5/12 |
| 4 or 5 (Excellent) | 1/12 |

| **Final evaluation of `SecureDexClassLoader`**<br>*Please provide an average mark on your satisfaction after having used `SecureDexClassLoader` (Grab 'n Run API).* | |
|---|---|
| 1 or 2 (Disappointing) | 0/12 |
| 3 | 0/12 |
| 4 or 5 (Excellent) | 12/12 |

| **Grab 'n Run learning overhead**<br>*Please quantify the effort in learning how to use Grab 'n Run API over the Native API (i.e., `DexClassLoader`).* | |
|---|---|
| 1 or 2 (Almost zero) | 12/12 |
| 3 | 0/12 |
| 4 or 5 (Extremely broad) | 0/12 |

| **Easy to implement**<br>*Look at the two applications you prepared, which one between the two was easier to implement?* | |
|---|---|
| First application (`DexClassLoader`) was easier to implement. | 0/12 |
| Second application (`SecureDexClassLoader`) was easier to implement. | 11/12 |
| Both of them were too difficult. | 1/12 |

| **Readability**<br>*Look at the two snippets of code you implemented for the applications, which one is easier to read and understand at a first glance?* | |
|---|---|
| First application (`DexClassLoader`) is way easier to read. | 0/12 |
| Second application (`SecureDexClassLoader`) is way easier to read. | 10/12 |
| They are more or less equally easy to understand. | 2/12 |
| Both of them are difficult to read. | 0/12 |

| **Flexibility**<br>*Between the two analyzed solutions, which one do you think offers more flexibility and features for your Android applications?* | |
|---|---|
| `DexClassLoader` (Native API). | 1/12 |
| `SecureDexClassLoader` (Grab 'n Run API). | 11/12 |

| **Code maintainability**<br>*If you decide to change the remote location of the APK used as source for DCL, or if you plan to perform DCL from a second remote APK, stored at a different URL, which one of the two applications would be easier to fix?* | |
|---|---|
| First application (`DexClassLoader`) would be easier to fix. | 1/12 |
| Second application (`SecureDexClassLoader`) would be easier to fix. | 10/12 |
| The amount of work would be exactly the same for both of them. | 1/12 |

| **Security**<br>*Which one between the two applications do you think is more secure?* | |
|---|---|
| First application (`DexClassLoader`) is more secure. | 0/12 |
| Second application (`SecureDexClassLoader`) is more secure. | 11/12 |
| They are the same, security-wise. | 1/12 |

# Repackaging tool

An easy way for developers to port existing applications to use GNR secure API.

Prototype implemented as a Python script that relies on *Androguard* and *apktool*.

**Input:**

- An APK container to be patched
- The code containers used as sources for DCL
- The binding between each container and the trusted certificate that must be used for its signature verification

**Output:**

★ A fully-working patched APK that makes use of GNR API instead of native DexClassLoader one.

# Limitations

- Binding containers to certificate through **package name**
  - Good assumption with APK (unique).
  - Less suitable for JAR (a package name may be present in more than one JAR...).
- **Limited API coverage**
  - Some API could be included easily (e.g., PathClassLoader)
  - Some other would require new ad-hoc solutions
- **Challenges in code reuse**
  - Our protocol imports copies of container in app-private folders..

# Future works

1. Optimize **GNR**.
   a. Improve performance by adding extra caching strategies.
   b. Extend GNR to wrap other native API for DCL.

2. Refine and widen the evaluation of the **repackaging tool** (e.g., perform a user study asking developers to patch their own apps with our script).

# Conclusions

1. Propose and design a **novel code verification protocol** to prevent unsafe implementations of DCL
2. Implement this approach and release it as an **open-source library**
3. Evaluate extensively our work through a **user study** with 12 participants
4. Design and implement the **repackaging tool** to patch apps to use our newly-developed library

*Thanks for your attention!*
*Check out Grab'n Run:*
*www.grabnrun.org*

***Some bonus slides... Only for the braves ;)***

# Simple use cases of DCL


## Silent updates

- ➔ Keep updated **third-party libraries** used by an app (both on features and security).
- ➔ **Decoupling updates** on the application from updates of non-standalone libraries

## Extensibility

- ➔ In games or premium version of applications
- ➔ The main app downloads an **external** APK with the extra features purchased by the user and load this code.

# How to perform DCL in Android?

2. **Package contexts:**
   a. Whenever Android loads an app, it associates it with a *Context* object.
   b. It is also possible to create contexts for other apps (identified by package name) to load not only another app resources but even its classes -> Code Injection Party :(

public abstract Context **createPackageContext** (String packageName, int flags)

**Parameters**

| | |
|---|---|
| *packageName* | Name of the application's package. |
| *flags* | Option flags, one of `CONTEXT_INCLUDE_CODE` or `CONTEXT_IGNORE_SECURITY`. |

**Returns**

A `Context` for the application.

3. **Native code, Runtime.exec..**

# DexClassLoader Android API

## Public Constructors

public **DexClassLoader** (String dexPath, String optimizedDirectory, [redacted]nt)

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` |
| [redacted] | |
| [redacted] | |

## Public Methods

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.

**Full class name = Package name + Simple class name**
**= com.example + ClassA**
**= com.example.ClassA**

**Returns**

the `Class` object.

**Throws**

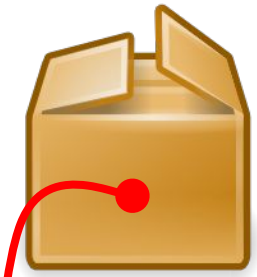
*ClassNotFoundException* if the class can not be found.

# DexClassLoader Android API: simple code example

```
ClassA classAInstance = null;
String myLibJarPath = getDir("containers", MODE_PRIVATE).getAbsolutePath() + "/myLibrary-dex.jar";
File dexOutputDir = getDir("dex", MODE_PRIVATE);

DexClassLoader mDexClassLoader = new DexClassLoader(    myLibJarPath,
                                                        dexOutputDir.getAbsolutePath(),
                                                        null,
                                                        getClass().getClassLoader());

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

        // Do something with the loaded object classAInstance
        // i.e. classAInstance.doSomething();

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

Full class name = Package name + Simple class name
= com.example + ClassA
= com.example.ClassA

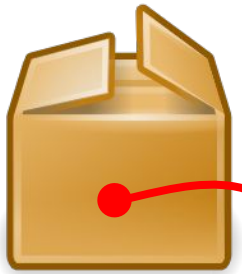
# *classes.dex*... How to add it to a library?



*myLibrary.jar*

**$ dx --dex --output=myLibrary-dex.jar myLibrary.jar**

```
processing archive myLibrary.jar...
processing META-INF/MANIFEST.MF...
processing com/example/ClassA.class...
processing com/example/ClassB.class...
processing .classpath...
processing .project...
processing .settings/org.eclipse.jdt.core.prefs...
writing .classpath; size 295...
writing .project; size 368...
writing .settings/org.eclipse.jdt.core.prefs; size 587.
writing classes.dex; size 884...
```



*myLibrary-dex.jar*

**$ jar xvf ../myLibrary.jar**
**$ tree**

```
.
├── com
│   └── example
│       ├── ClassA.class
│       └── ClassB.class
└── META-INF
    └── MANIFEST.MF

3 directories, 3 files
```

**When you export an APK container from the ADT or Android Studio, dx is called and so translation from Java Bytecode (.class) to Dalvik Bytecode (.dex) is automatically performed.**

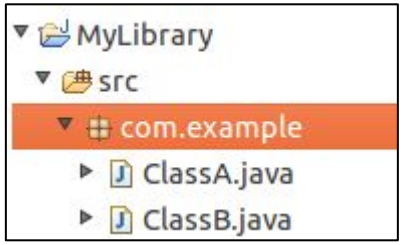
**$ jar xvf ../myLibrary-dex.jar**
**$ tree**

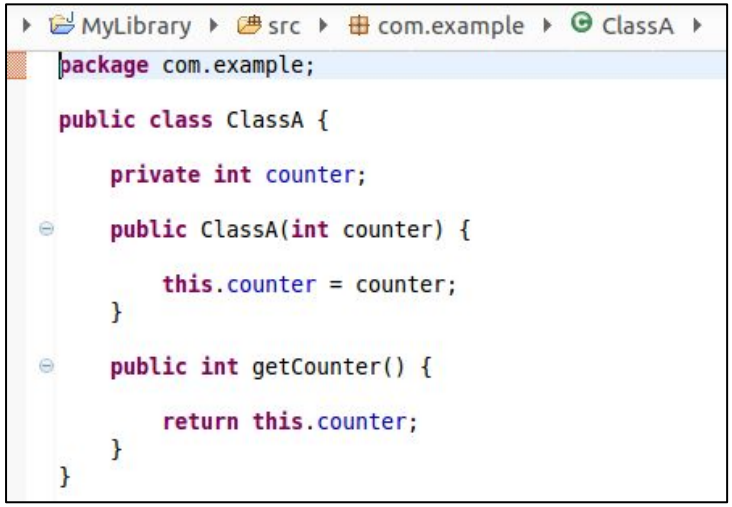
```
.
├── classes.dex
└── META-INF
    └── MANIFEST.MF

1 directory, 2 files
```

# Package name: what and where?

"A *package* is a grouping of related classes, interfaces and enumerations providing **access protection** and **name space management**."

- Every Java class is associated to a **package** (its name is recalled in the first line of the class).



# Package name in APK containers

- APK containers must have just **ONE** package name (chosen by the developer when a new application is created).
- Package name is stored in the *Android Manifest*.

AndroidManifest.xml ×

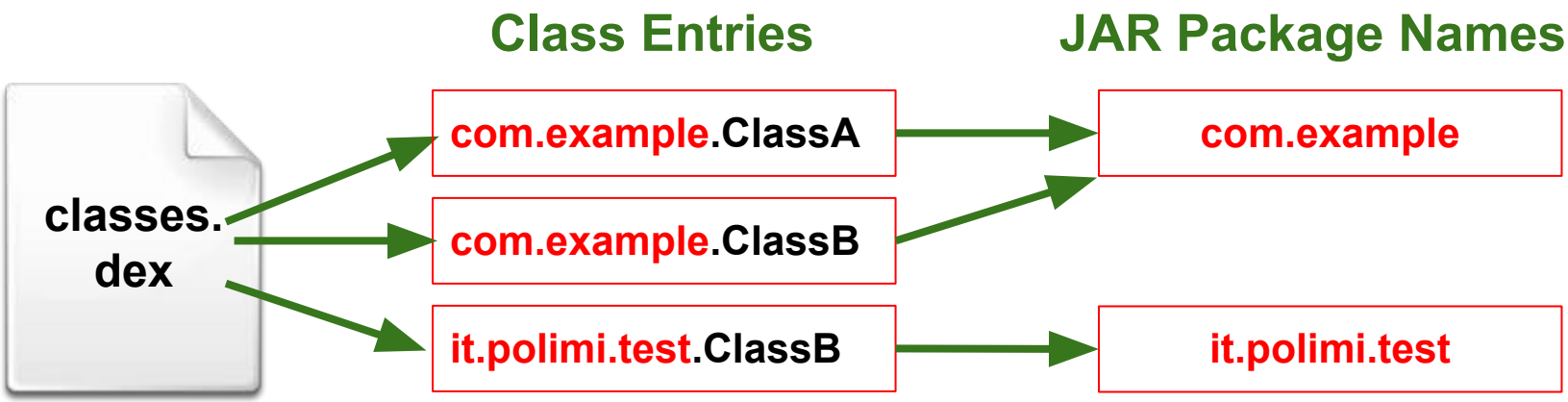
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.necst.grabnrun.profiler"
    android:versionCode="1"
    android:versionName="1.0" >
```

- It must be **unique** and must **not change** during the whole lifecycle of the app (Penalty: no admission to *Google Play Store*).
- It can be found easily by querying a *PackageManager* object:

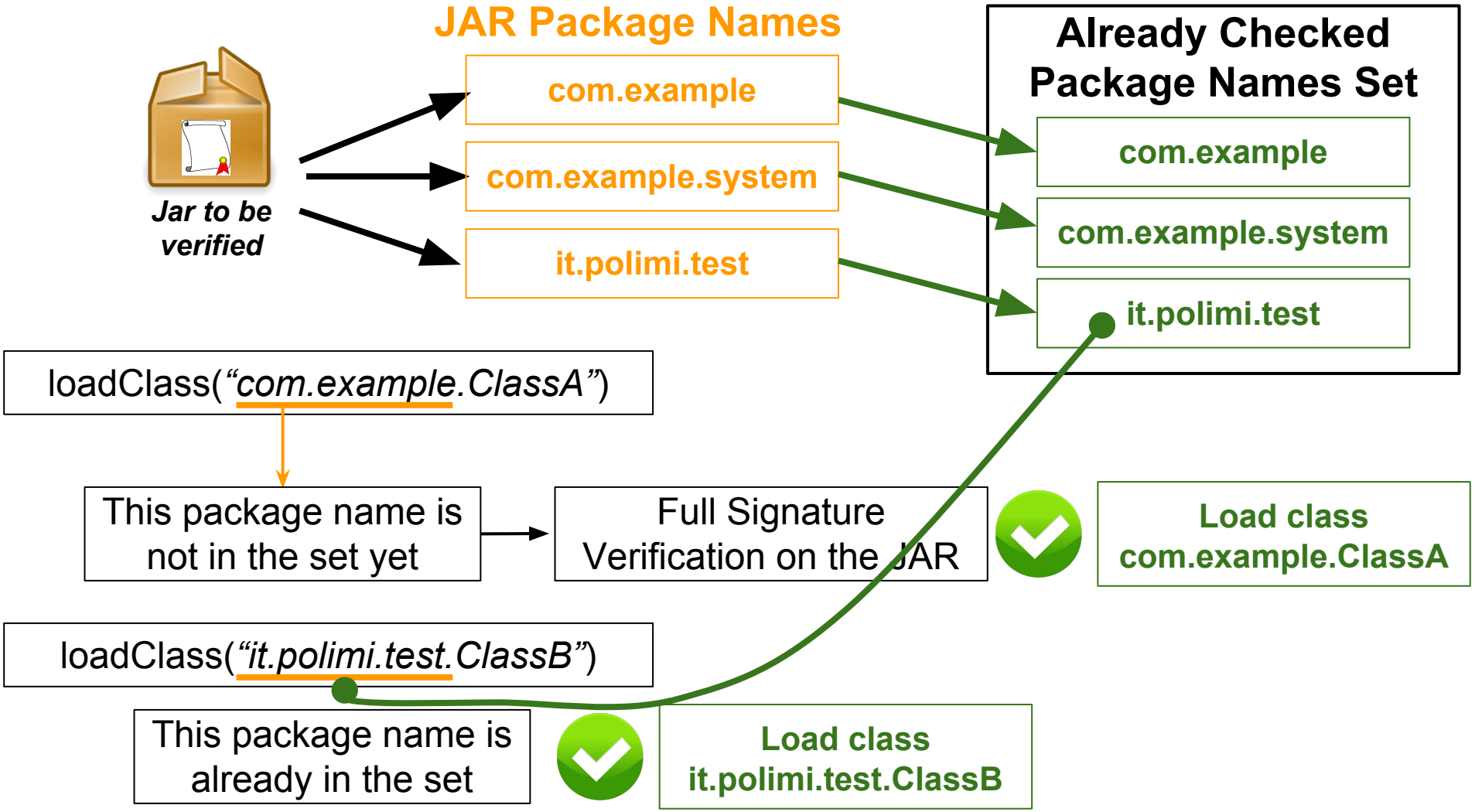
```
PackageManager.getPackageArchiveInfo(archiveFilePath, 0).packageName
```

# Package name(s) in JAR containers

- Every JAR container can have **many** package names.
- Potentially **n** *classes* may imply **n** *package names*.
- *No straightforward approach* to recover them
  - In this scenario a convenient solution is parsing *classes.dex* file and from class entries extract package names.



# Caching of signature verifications



# Grab’n Run (GNR)

- GNR is a **simple** *Java library* compatible with any Android project to secure *DCL* operations.
- GNR implements our *code verification protocol*.
- GNR solves **all** the presented **issues** and it grants also some extra features.

# SecureDexClassLoader API (1)

To instantiate this object you need to call a method on a helper *class* called **SecureLoaderFactory**:

**Method Detail**

**createDexClassLoader**

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

**Parameters:**

- *dexPath* - the list of jar/apk files containing **classes and resources**; these paths could be *either local* URLs pointing to a location in the device or URLs that links to a *resource* stored in the *web via* ***HTTP/HTTPS***.
- *packageNameToCertificateMap* - a map that couples each **package name** to a **remote** *URL* which contains the **certificate** that must be used to validate all the classes that belong to that package before loading them at run time.

**Returns:**

➔ a **SecureDexClassLoader** object which can be used to *load dynamic code securely*.

# SecureDexClassLoader API (2)

After having obtained a reference, you can invoke the usual ***loadClass*** method on the object:

**Method Detail**

**loadClass**

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

**Parameters:**

- *className* - the full class name to load.

**Full class name = Package name + Simple class name**
**= com.example + ClassA**
**= com.example.ClassA**

**Returns:**

➔ either a **class** to load at runtime accordingly to *className* if the *verification process succeeds* or a **null** pointer in case that *at least one of the security constraints* for secure dynamic class loading *is violated*.

**Throws:**

★ *ClassNotFoundException* - this exception is raised whenever *no security constraint is violated* but the **target class is not found** in any of the available source containers.

# Comparison among code snippets

***jarContainerPath*** contains the path to the *target container* for class loading.

```
String myLibJarPath = getDir("containers", MODE_PRIVATE).getAbsolutePath() + "/myLibrary-dex.jar";

File dexOutputDir = getDir("optimizedDex", MODE_PRIVATE);

DexClassLoader mDexClassLoader = new DexClassLoader(    myLibJarPath,
                                                        dexOutputDir.getAbsolutePath(),
                                                        null,
                                                        getClass().getClassLoader());

try {
    Class<?> loadedClass = mDexClassLoader.loadClass("com.example.ClassA");
    classAInstance = (ClassA) loadedClass.newInstance();
```

**Only LOCAL file path**

**DexClass Loader (native)**

**Both REMOTE URL and LOCAL file path**

```
String myLibJarPath = "http://myLibrary.com/downloads/mobile/myLibrary-dex.jar";

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( myLibJarPath,
                                                                                        null,
                                                                                        packageNamesToCertMap,
                                                                                        getClass().getClassLoader());

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

if (loadedClass != null)
        classAInstance = (ClassA) loadedClass.newInstance();
```

**Secure DexClassLoader (GNR)**

P.S. No **exception handling** is shown in these snippets because this code is exactly the same for both solutions..

# Hey! Hold on a sec.. Damn!

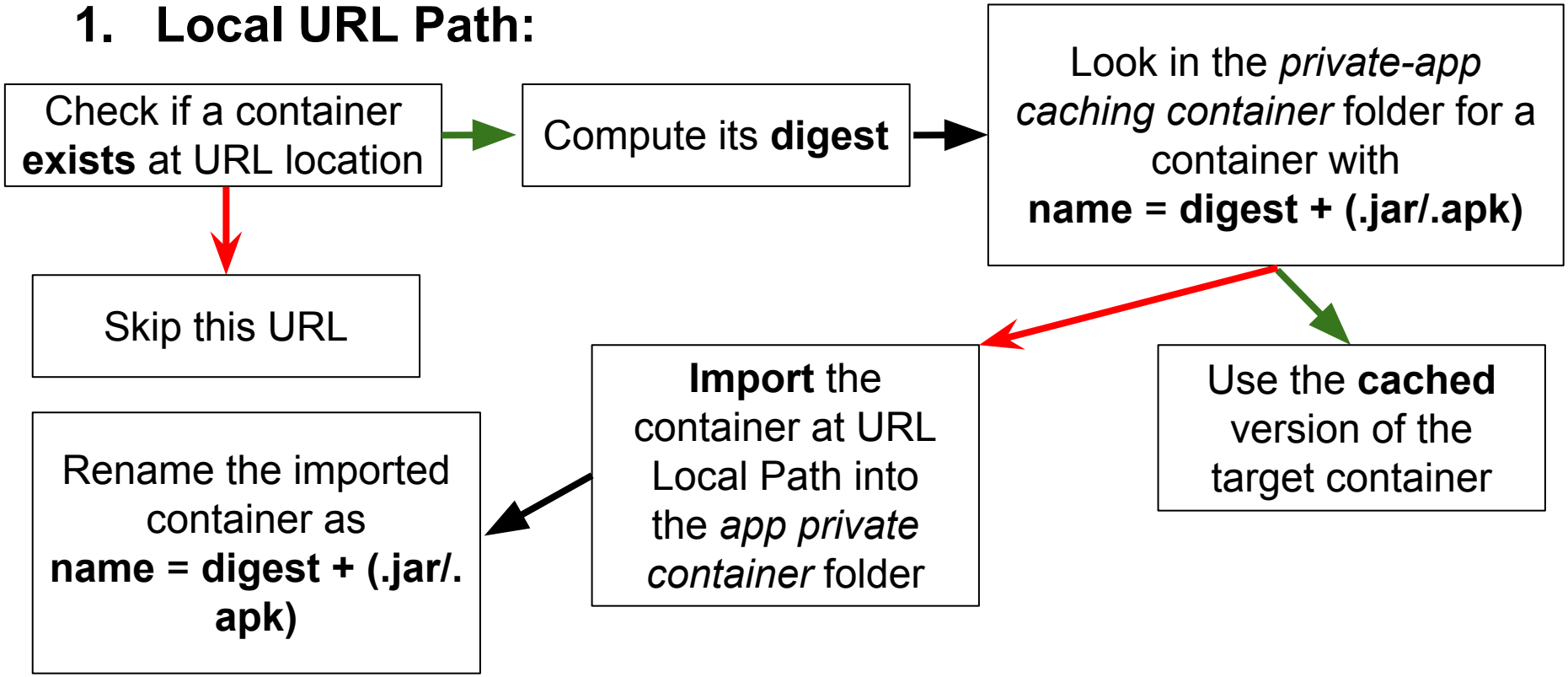
When you design systems for **mobile devices**, you must take into account **bad/missing Internet connectivity**.

- Performance mantra: **“Caching is the way..”**. Remember and apply this also to **remote containers** and **certificates**.
- Outcome: GNR can handle situations where the smartphone is **offline.**

# Caching for containers (APK/JAR)..

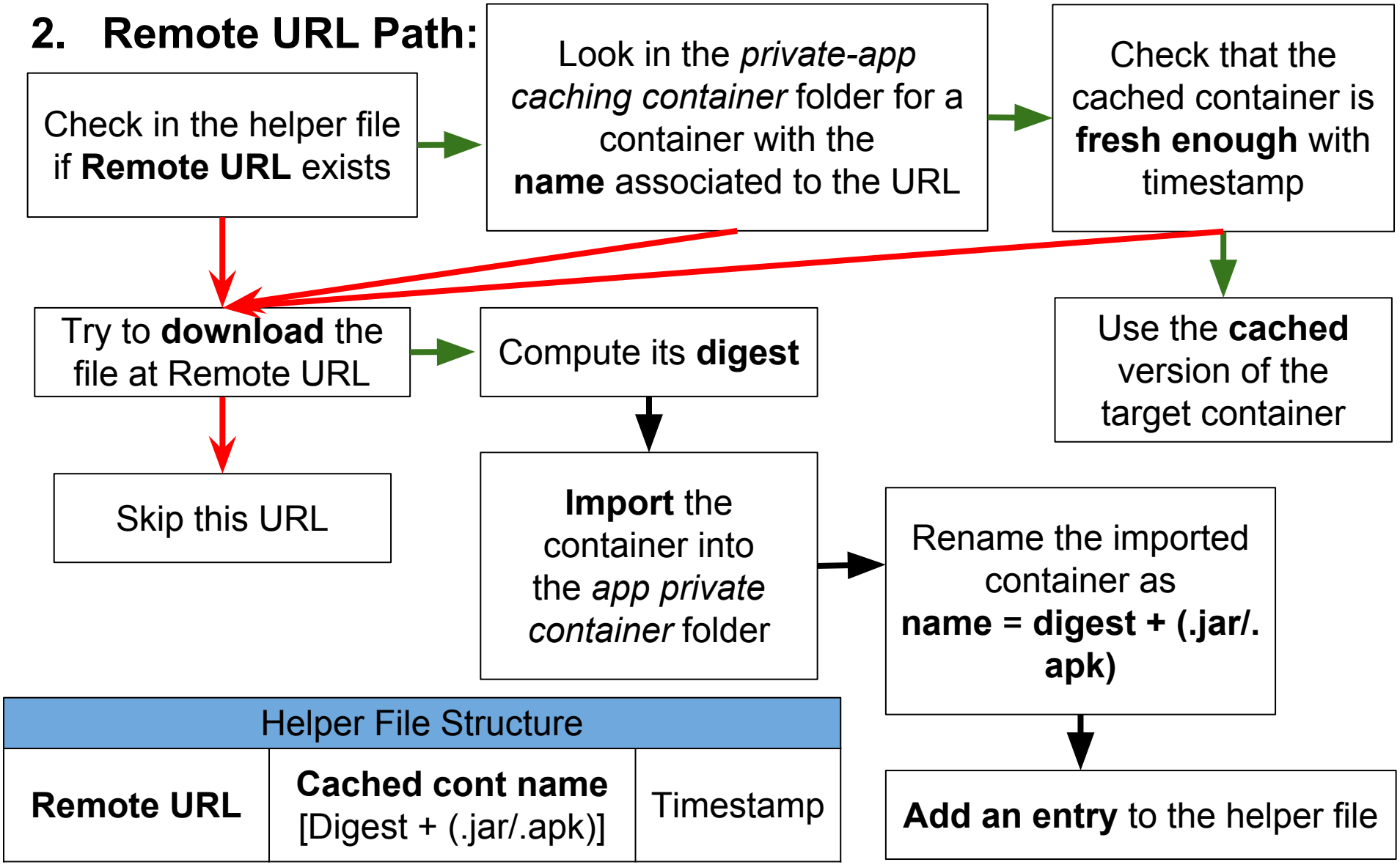
While **SecureLoaderFactory** is parsing *dexPath* string variable, each substring falls in one of these two cases:

1. **Local URL Path:**



# Caching for containers (APK/JAR)..

## 2. Remote URL Path:




| Helper File Structure | | |
|---|---|---|
| **Remote URL** | **Cached cont name** [Digest + (.jar/.apk)] | Timestamp |

# Interesting scenario/use case:

# Automatic silent updates for remote libraries

Through **SecureDexClassLoader** it is possible to handle this scenario in an easy *and secure* way.

**myLibrary.jar developer**

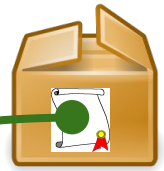
She stores on the web..

A **redirect HTTP** link which points to the **latest version** of **myLibrary-dex.jar,** signed with the developer private key.

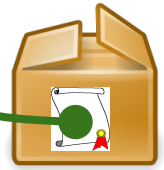
http://myLibrary.com/downloads/mobile/latest

An **HTTPS** link which points to the **developer certificate**, stored in a safe location under the dev domain.

https://myLibrary.com/developerCert.pem



**vers 1.8.1**



**vers 1.9.0**

# Interesting scenario/use case:

# Automatic silent updates for remote libraries

**Developer willing to use myLibrary-dex.jar**

1. Create a **SecureDexClassLoader** instance with the *redirect HTTP link* as *dexPath* and the *map package names* pointing to the *developer certificate*.
2. Load the *target classes* from the **latest** myLibrary.jar **version** :)

```
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( "http://myLibrary.com/downloads/mobile/latest",
                                                                                        null,
                                                                                        packageNamesToCertMap,
                                                                                        getClass().getClassLoader());

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

***Et voilá!*** Done..

It was pretty straightforward for both actors, wasn't it?

# Benefits from SecureDexClassLoader

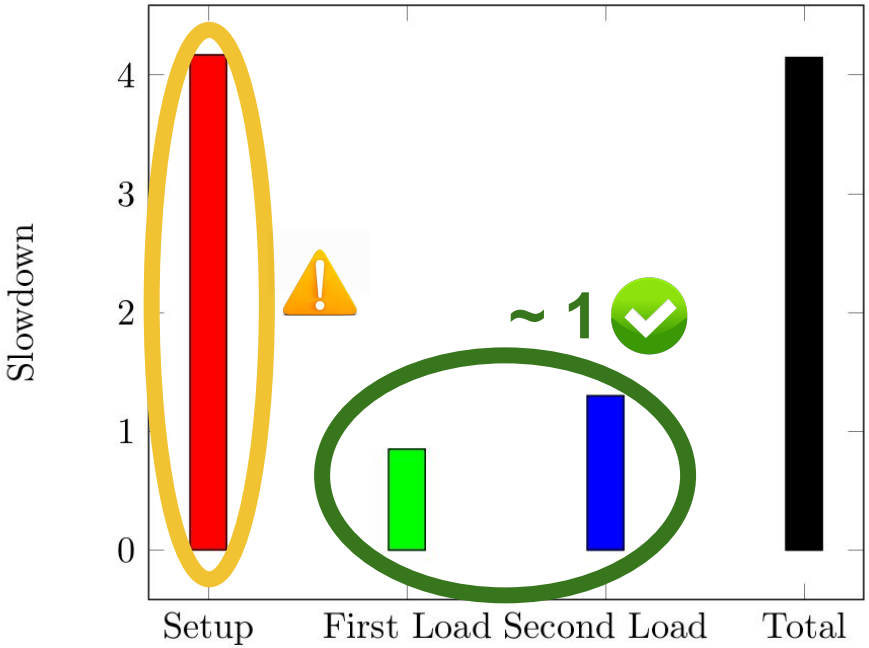
- This API grants both *integrity* and *developer authentication* on each used container.
- It is able to hide and *manage all the security issues* shown in DexClassLoader.
- SecureDexClassLoader is able to manage both **remote** and *local* containers.
- It is possible to *load concurrently* many classes (GNR is **thread safe**).
- *Performance overhead* is mitigated thanks to a **caching** systems on *containers*, *certificates* and previous *signature verification* on containers.
- **Caching** system makes possible to *partially use* the library also when the mobile is **offline**.

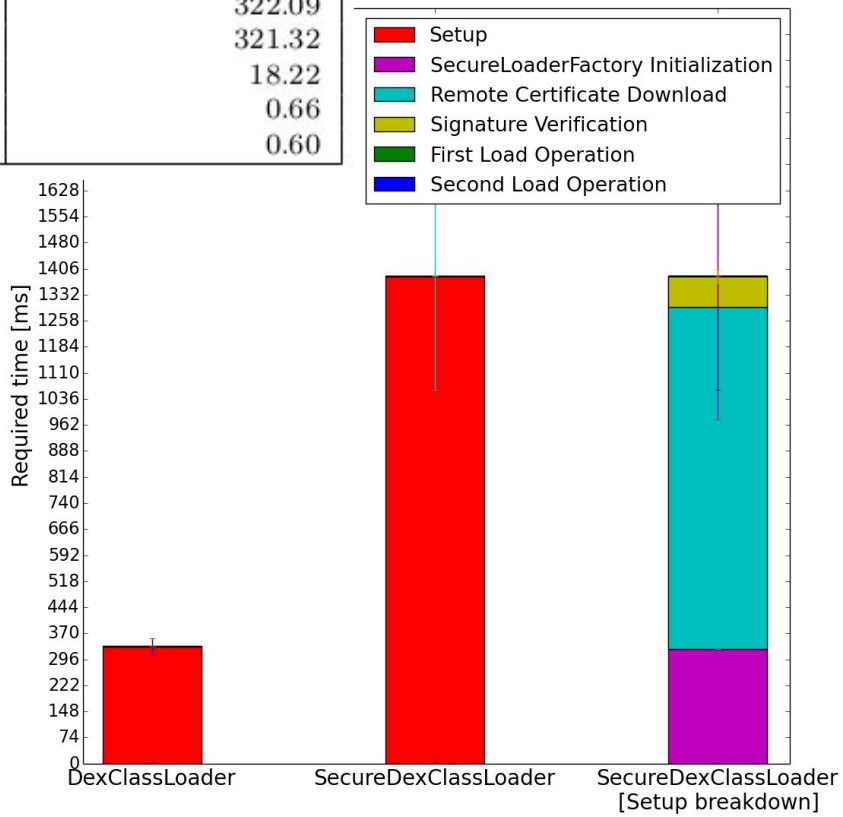
# Performance overhead

We measured timestamps of a simple profiling app using both DexClassLoader and GNR API. We executed it on a Nexus 5 for 100 times.

| 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 |

**No Cached Resources**



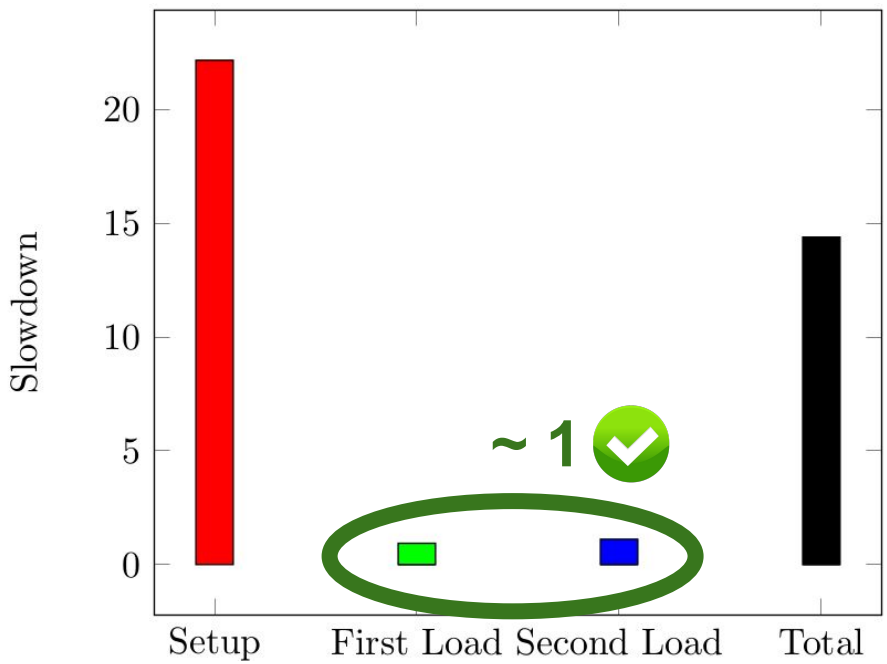
# Performance overhead

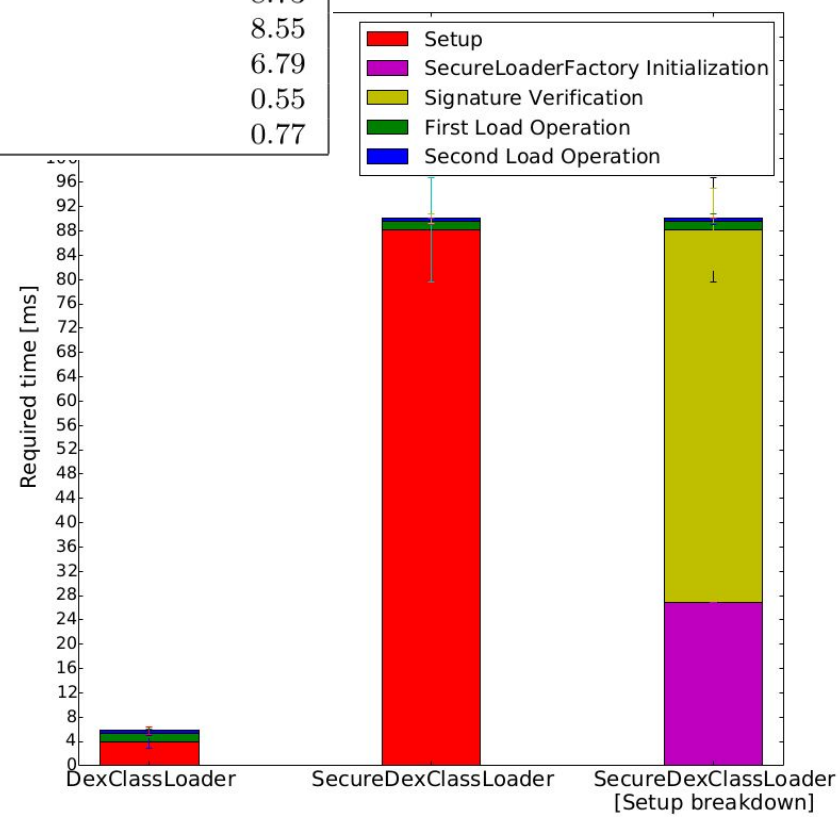
We measured timestamps of a simple profiling app using both DexClassLoader and GNR API. We executed it on a Nexus 5 for 100 times.

| 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 |

**Cached Resource**

1. **No DEX translation**
2. **No certificate fetch**



# Performance overhead (2)

We introduced several **caching improvements** to speedup GNR. Thanks to these strategies:

1. Each **remote** code container and certificate is *fetched only once* across the different executions.

2. Each **local** code container is *imported only once* into the application-private folder.

3. The **signature** of each code container is *checked only once* and the result of the verification is propagated to all the package names in that container.

# Performance overhead - No Cache

| Statistic Measure | Mean | Median | Standard Deviation |
|---|---|---|---|
| DexClassLoader | 334.20 | 332.00 | 23.50 |
| DexClassLoader Setup | 331.91 | 330.00 | 23.60 |
| DexClassLoader Load Operation | 1.55 | 1.00 | 0.70 |
| DexClassLoader Second Load Operation | 0.30 | 0.00 | 0.46 |
| SecureDexClassLoader | 1386.13 | 1237.00 | 322.05 |
| SecureDexFactory Preparation | 1384.13 | 1234.00 | 322.09 |
| Download Container | 0.00 | 0.00 | 0.00 |
| Download Certificate | 972.32 | 822.00 | 321.32 |
| Verify Signature | 86.17 | 82.00 | 18.22 |
| Load Operation | 1.32 | 1.00 | 0.66 |
| Second Load Operation | 0.39 | 0.00 | 0.60 |
| Wipe Cached Data | 0.00 | 0.00 | 0.00 |

# Performance overhead - Cache

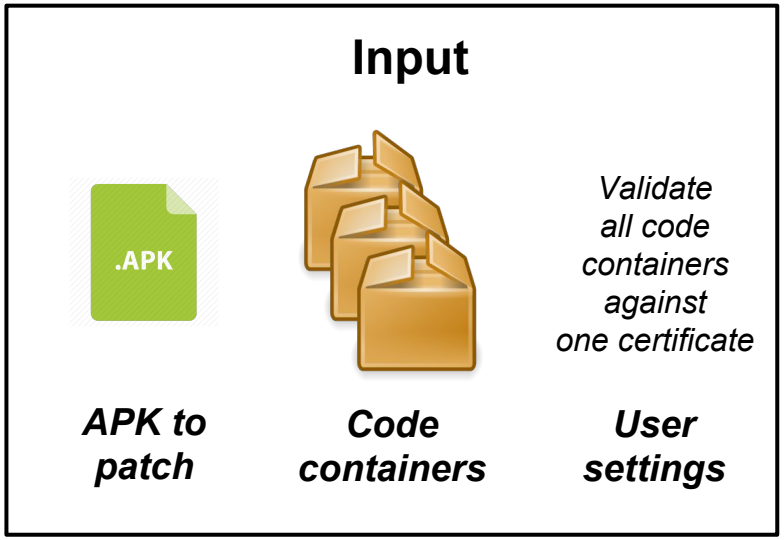
| Statistic Measure | Mean | Median | Standard Deviation |
|---|---|---|---|
| DexClassLoader | 6.28 | 6.00 | 1.39 |
| DexClassLoader Setup | 3.98 | 4.00 | 1.04 |
| DexClassLoader Load Operation | 1.42 | 1.00 | 0.70 |
| DexClassLoader Second Load Operation | 0.44 | 0.00 | 0.61 |
| SecureDexClassLoader | 90.42 | 90.00 | 8.73 |
| SecureDexFactory Preparation | 88.25 | 87.50 | 8.55 |
| Download Container | 0.00 | 0.00 | 0.00 |
| Download Certificate | 0.00 | 0.00 | 0.00 |
| Verify Signature | 61.39 | 61.00 | 6.79 |
| Load Operation | 1.33 | 1.00 | 0.55 |
| Second Load Operation | 0.49 | 0.00 | 0.77 |
| Wipe Cached Data | 0.00 | 0.00 | 0.00 |

# Grab’n Run: User study

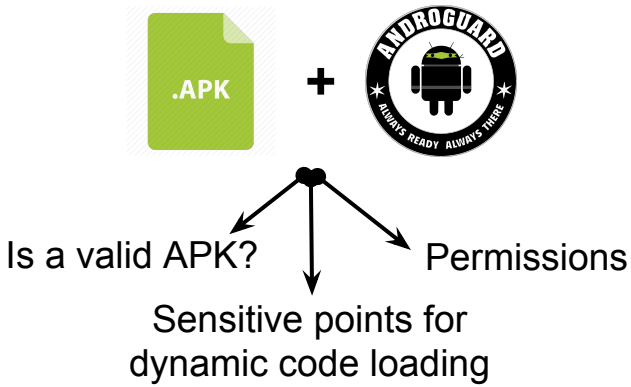
We performed a user study on 12 Android developers to verify whether:

1. GNR is easier to use than native *DexClassLoader* API.
2. GNR learning overhead for a developer is little, once he knows how to use native API.

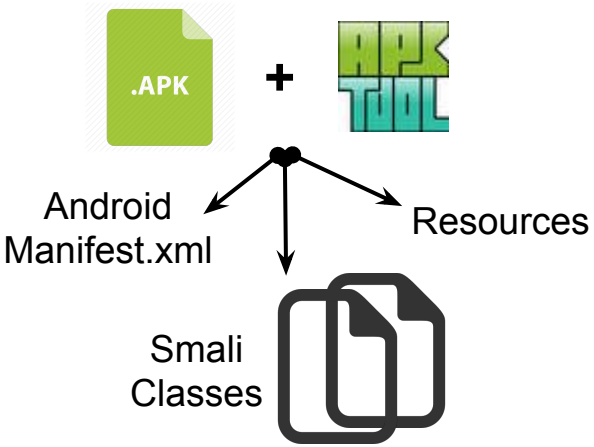
# Repackaging tool: Functioning (1)



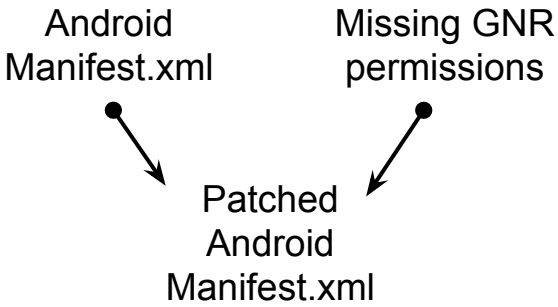

1. **Analyze APK with Androguard**




2. **Disassemble APK with apktool**

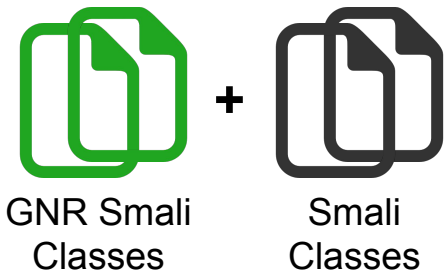



3. **Add missing permissions to AndroidManifest.xml**

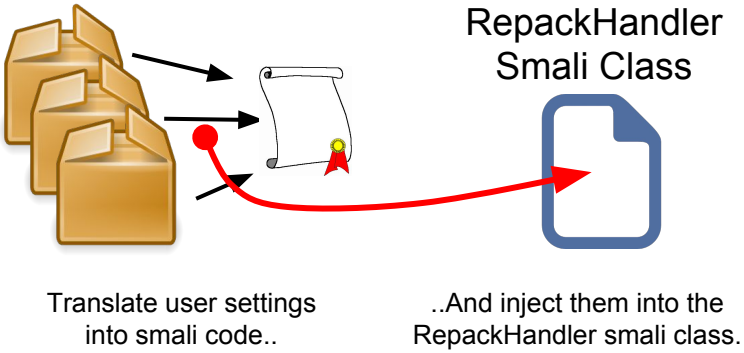


# Repackaging tool: Functioning (2)

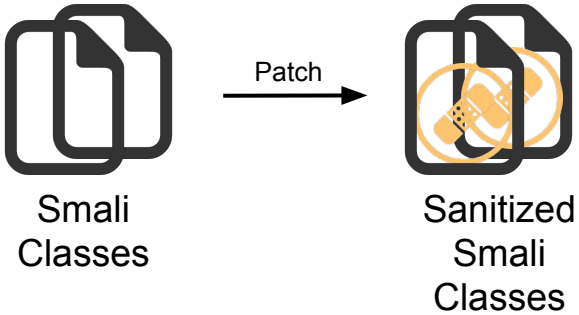
**4. Copy GNR files in smali folder of the APK**



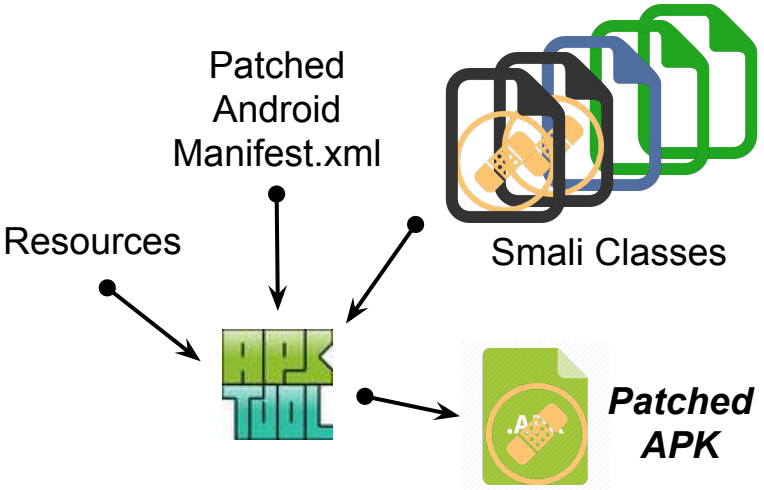

**5. Inject user settings in smali code of RepackHandler class**




**6. Patch sensitive points in smali classes of the APK**




**7. Rebuild patched APK**



# Related works in the field

1. **Remote Code Execution**

   [*Poeplau et al.*] Dalvik VM modification + external service to analyze dynamically loaded code components

2. **Code Verification**

   [*Vidas and Christin*] Public-key infrastructure backed by the domain name system