Grab 'n Run: Secure and Practical Dynamic Code Loading for Android Applications

Luca Falsina [1]
luca.falsina@mail.polimi.it

Yanick Fratantonio [2]
yanick@cs.ucsb.edu

Stefano Zanero [1]
stefano.zanero@polimi.it

Christopher Kruegel [2]
chris@cs.ucsb.edu

Giovanni Vigna [2]
vigna@cs.ucsb.edu

Federico Maggi [1]
federico.maggi@polimi.it

[1] Politecnico di Milano
Milano, Italy

[2] UC Santa Barbara
Santa Barbara, CA, USA

ABSTRACT

Android introduced the *dynamic code loading* (DCL) mechanism to allow for code reuse, to achieve extensibility, to enable updating functionalities, or to boost application start-up performance. In spite of its wide adoption by developers, previous research has shown that the secure implementation of DCL-based functionality is challenging, often leading to remote code injection vulnerabilities. Unfortunately, previous attempts to address this problem by both the academic and Android developers communities are affected by either practicality or completeness issues, and, in some cases, are affected by severe vulnerabilities.

In this paper, we propose, design, implement, and test Grab 'n Run, a novel code verification protocol and a series of supporting libraries, APIs, and tools, that address the problem by abstracting away from the developer many of the challenging implementation details. Grab 'n Run is designed to be practical: Among its tools, it provides a drop-in library, which requires no modifications to the Android framework or the underlying Dalvik/ART runtime, is very similar to the native API, and most code can be automatically rewritten to use it. Grab 'n Run also contains an application-rewriting tool, which allows to easily port legacy or third-party applications to use the secure APIs developed in this work.

We evaluate the Grab 'n Run library with a user study, obtaining very encouraging results in vulnerability reduction, ease of use, and speed of development. We also show that the performance overhead introduced by our library is negligible. For the benefit of the security of the Android ecosystem, we released Grab 'n Run as open source.

Categories and Subject Descriptors

[**Security and privacy**]: Software and application security–*Software security engineering*

ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA

http://dx.doi.org/10.1145/2818000.2818042.

General Terms

Security, Runtime Execution, Code Integrity

Keywords

Android; Dynamic Code Loading; DexClassLoader

1. INTRODUCTION

The wide adoption of smartphones pushed software developers to develop an incredibly high number of applications, and the official market for Android currently hosts more than one million apps [3]. The great success of the platform, which dominates the rivals with an 85% market share [11], and the ease in monetization contributed to create a market with a multitude of applications for each category of interest. Moreover, the fierce competition pushes developers to constantly add more features and functionalities.

To implement some of these functionalities in a reliable and performance-savvy way, applications make extensive use of *dynamic code loading* (DCL). Similar in spirit to the notion of shared libraries, DCL allows an application to load (and execute) code that is not part of its static, initial code base. This additional piece of code is loaded at runtime (thus the term "dynamic"), and it might not necessarily be present in the application package at installation time: In fact, the additional code could be retrieved, once again at runtime, from a remote network endpoint.

Unfortunately, it is well-known that DCL can be abused by malicious applications [7, 8, 9, 14], mainly to evade static and dynamic analysis systems. As documented by Poeplau et al. [9], this technique is also often used by benign applications, for a variety of reasons including code reuse, extensibility, self-upgrade functionality, and to boost the overall performance.

Although these are all legitimate motivations, Poeplau et al. also show that it is really challenging to implement DCL functionality in a *secure* way. Indeed, the authors identified severe vulnerabilities related to incorrect usage of DCL in very popular games, advertisement frameworks, and code platforms. According to their measurement on the Google Play Store, 16% of the top 50 applications were affected by similar vulnerabilities, potentially affecting several millions of users: In the worst case, a vulnerable implementation of DCL can lead to remote code injection, one of the most dangerous vulnerability classes.

Existing Solutions. Previous academic works have pro-

posed solutions to address this problem. For example, [9] proposed a series of modifications to the Android framework and the introduction of a "verification service" in charge of vetting any code component (both full-fledged APKs and additional code modules in form of raw Dalvik bytecode). Unfortunately, this solution is not very practical. In fact, this approach requires non-trivial modifications to the Android framework, and its adoption in the short-term is even less likely when considering the massive fragmentation of the different Android versions [13]. Another limitation is that the developers would need to go through a time-consuming vetting process for each of the additional components, thus defeating one of the main use cases of DCL — the ability for a developer to flexibly deploy additional components to her application. Another relevant work is by Vidas and Christin [12], who address the sub-problem of *secure code retrieval* by relying on a lightweight PKI-like infrastructure backed by the domain name system, which essentially uses the reverse of the package-name string as the lookup key to retrieve the signing certificate—referred by a TXT record. Although such work clearly constitutes a step forward, it is not comprehensive, because it does not prevent other classes of errors that might introduce a vulnerability (e.g., storing the securely retrieved code in an *unsafe location* on the file system). Also, their scheme is not always practical because of the package name restrictions. We discuss these limitations and their implications in more details in Section 8.

The high request for a practical mechanism to use DCL also pushed the open-source community to develop DynamicLoadAPK [10], a very popular library with more than one thousands "stars" on GitHub. Unfortunately, even if this library eases the implementation of DCL mechanisms, the risk for a developer to introduce severe security vulnerabilities is still really high. In fact, as a case in point, upon a quick investigation, we identified a severe security vulnerability in the library itself: DynamicLoadAPK stores the code to be loaded on the external storage of the device (e.g., SD card), which is writable by any application with the `WRITE_EXTERNAL_STORAGE` permission [6]. This opens the host application to code injection attacks. While fixing this specific bug might be trivial, there are still a number of aspects that a user of this library would need to take care of: For instance, it still leaves up to the developer the task of safely retrieve the code, and to perform the integrity checks required to verify whether it has been tampered with. Also, the discovery of such a vulnerability in the library itself reinforces the intuition that securely implementing a functionality based on DCL is challenging.

Proposed Solution. With the goal of allowing developers to fully leverage the power of DCL in a secure and practical way, we designed, implemented, and tested Grab 'n Run, a novel code verification protocol and a series of supporting libraries, APIs, and components. To the best of our knowledge, this is the first toolset of its kind.

Grab 'n Run is *secure* because the design of its APIs removes from the developer the excessive burden of thinking about all the security aspects, by abstracting away all the implementation details that are known to be challenging to be securely implemented. In other words, the adoption of our APIs avoids, *by design*, the introduction of a broad class of security vulnerabilities related to the usage of DCL (e.g., the ones described in [9]).

We believe Grab 'n Run also to be *practical*. In fact, the library we developed is thought as a drop-in Java library for Android, which requires no modifications to the Android framework or the underlying Dalvik/ART runtime. Moreover, the exposed API is very similar to the native ones, with which the developers are already familiar. For these reasons, we believe that our library could immediately benefit the Android developers community thanks to its negligible adoption barrier. To lower the burden of adoption even more, we also developed an application-rewriting tool, which aims to assist a developer to port existing apps to use our newly-developed library. At its core, this component operates directly on Dalvik bytecode, and it therefore does not require access to the source code of the original app. Thus, our work allows a developer to automatically secure not only her code base, but also legacy, third-party, and untrusted libraries.

Experimental Evaluation. To evaluate Grab 'n Run, we first performed a user study to assess the security and practicality benefits brought to the developers by the Grab 'n Run library. To this end, we asked 12 Android developers to implement a functionality based on DCL, first by using the standard Android API, and then by using Grab 'n Run library. In both cases, we told the participants that we would have evaluated their performance under several aspects—mainly the security aspect. The results are strikingly different: When using the native APIs, all of the developers introduced one or more vulnerability. Instead, when using the Grab 'n Run API, all of them were able to implement the very same functionality in a secure way. Moreover, no developers expressed any concerns about the practicality of the Grab 'n Run library, even when we specifically asked for their feedback about this aspect. Finally, we measured the (negligible) speed overhead introduced by our library under several *caching* assumptions.

Original Contributions. In summary:

- We designed a code verification protocol, and then implemented it in a library and a series of tools, including an application-rewriting tool, that allow secure and practical implementation of a DCL functionality, easy porting of existing apps, and smooth adoption by developers.
- We conducted a user study with 12 Android developers and empirically showed the security benefits and practicality of Grab 'n Run. Indeed, none of the participants introduced security vulnerabilities when using our library, and all of them agreed on it being practical.
- Since we believe Grab 'n Run library to be the first of its kind, in the spirit of open science and for the benefit of the security of the Android ecosystem, we released it as fully open source [1], accompanied with documentation, tutorials, proof-of-concepts, and usage examples.

2. DYNAMIC CODE LOADING

This section provides background information on DCL, the common implementation patterns, and the security pitfalls highlighted by previous research.

2.1 Usage in Benign Applications

Benign applications often make use of DCL, for a variety of purposes [9]. Here, we list the principal ones.

[1] `http://grabnrun.org`

Code reuse. Applications often use third-party libraries (e.g., Adobe Air, codecs). As a technique to reduce the size of application packages (APKs), developers can opt to not embed those libraries in the APKs, but to store them in a common location, shared among different applications, and load them at runtime by means of dynamic code loading.

Improving App Startup Performance. Complex applications that comprise several libraries can rely on DCL to postpone the loading of (some of) such libraries to when they are actually used. This technique can increase the performance of the application when started.

Extensibility. Applications use DCL to modularly extend their functionality. For example, several games or apps offer the possibility to purchase additional levels or premium features via in-app billing: These additional components are often distributed as additional code that is dynamically loaded.

Self upgrade. Several libraries (e.g., advertising frameworks) wish their updates to be decoupled from the updates of the hosting application, so as to enable continuous deployment. Moreover, in Android, the default update-distribution mechanism conflicts with modern development practices, where small updates are released frequently: Even if the user sets the phone to install updates automatically, the phone would always provide notifications, which negatively affect the user's experience.

2.2 Implementation Patterns & Vulnerabilities

In this section we describe the three main conceptual steps needed to implement a generic DCL functionality, discussing the most common errors that developers commit for each step:

1. **Code Retrieval**. The application needs to retrieve the code component to be dynamically loaded. In the most general case, this component is fetched from a network endpoint. In case the code component is stored within the original application package or within an already-installed application, then this phase could trivially consist in retrieving the code component from its original bundle.

   Vulnerabilities. The most common mistake is to fetch the component over a plaintext protocol (most often, via HTTP as opposed to HTTPS). The problem is that an attacker could perform a man-in-the-middle (MITM) attack and actively tamper with the content of the transmission. This is not a real issue if the application properly verifies that the downloaded content has not been tampered with. However, Poeplau et al. [9] showed that most applications fail to do that.

2. **Code Storage**. The application needs to store the code component on the file system. This is necessary because of a limitation of the current Android framework API, which does not allow to load code directly from memory.

   Vulnerabilities. Many applications often store the just-fetched component in a world-writable location on the file system (e.g., the SD card). This allows any malicious application (with the required permission) to modify the stored code component, and to subsequently execute arbitrary code within the context of the target application: the malicious code would hence obtain all permissions and data of the target application, and it would be able to execute actions on its behalf.

3. **Code Loading and Verification**. The application needs to initialize the Android framework library responsible to handle the loading operation, and it is then ready to actually load the code component. A security-conscious developer would also *verify*, through an integrity check, that the code about to be loaded is an exact copy of the component retrieved in the first step.

   Vulnerabilities. Ensuring end-to-end code integrity and authenticity is known to be a challenging problem. To make things worse, the current Android framework API does not require (nor even encourage) the developers to at least *think* about this critical aspect, thus causing many developers to implement DCL without even attempting to verify the integrity of the code they are about to load.

3. THREAT MODEL

In this section, we describe the threat model considered for this work, and we discuss what we assume an attacker can do, and what is instead outside our scope.

The goal for a generic attack against our framework is to *execute arbitrary code within the context of a target application.* To achieve this, an attacker can target and exploit vulnerabilities in the three different components of our scenario:

1. **Device.** We assume that the attacker *can* execute code on the user's device, by means of a previously-installed application under the attacker's control. We assume that such application has read or write access on the storage of the device (e.g., internal storage or SD card). However, we also assume that the application does not have root privileges, as that would make the attacker's goal trivially achievable.

2. **Network Communication Channel.** We assume that the attacker *can* mount arbitrary MITM attacks over all unencrypted connections (e.g., HTTP). At the same time, we assume that the attacker cannot tamper with encrypted connections (e.g., HTTPS).

3. **Remote Server(s).** We assume that the attacker *cannot* compromise a remote server controlled by the developer of the target application. As we will discuss later, if the application relies on more than one remote server, our framework is robust even if the attacker compromises *all but one* of them. This is because our system uses the remote server as the *root* (or, in other terms, trusted element) of a chain of trust to check the integrity of the code that the application dynamically loads.

4. GRAB 'N RUN OVERVIEW

In this section we discuss the goals of our work and the main challenges. Then, we present our code verification protocol, and we discuss how we used it as a basis to implement a library to perform DCL securely. Finally, we present the "repackaging tool," which allows a developer to adapt third-party, potentially-vulnerable libraries to use our library.

4.1 Design Goals & Challenges

The first goal of our work consists in designing a new library that, by design, ensures that developers of Android applications implement functionality based on DCL in a *secure* way. To this end, we aimed at designing a novel code

verification protocol, and at using it as a basis to develop a new library based on it.

As a second goal, we want this library to be *practical*, even for developers that are not security experts. For this reason, we aimed to design our library as a drop-in, developer-friendly Java library that replaces the standard Android API without requiring any complex code modification beyond trivial refactoring (e.g., function calls renaming). The development of such a library is challenging. First, it is well-known that, when striving for security, practicality is usually affected. Second, benign apps use DCL for a variety of purposes and in very different ways. Thus, our design needed to support a number of different use cases.

Our third goal is to make this library usable also when DCL functionality are implemented within third-party libraries. In other words, we want to help developers to migrate existing, potentially vulnerable code bases to use our library. This aspect presents the following challenges: the process needs to be automatic; developers might not be security experts; the source code of these third-party libraries is usually not available. To this end, we developed a "repackaging tool" that takes as input the Dalvik bytecode of an existing code base, and it is able to automatically rewrite it so that it uses our newly-developed API instead of using the less-secure native Android API.

4.2 Code Verification Protocol

Our verification protocol focuses on avoiding that, by design, a careless or inexperienced developer may make common mistakes when implementing a DCL-based functionality. For completeness, we consider the most generic case, which consists in an application that dynamically load code retrieved at runtime from a remote network endpoint. Our protocol is constituted by five different steps. Figure 1 shows a graphical representation of the five steps *in action*, by using an example of how our code verification protocol can be used by an application to dynamically load a library (in the example, located in a "Code repository," not necessarily under the control of the developer). The remainder of this section describes these five steps in details, and thoroughly discusses the motivation behind our design choices, their implications, and how they prevent the many possible attacks.

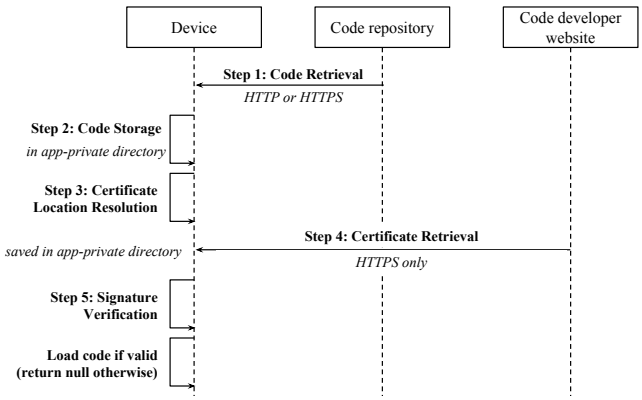

Figure 1: Sequence diagram of a simple use case of remote DCL. The diagram shows interactions between the application running on a device, the code repository, which hosts the dynamically loaded code, and the developer's website hosting the certificate.

Step 1: Code Retrieval. The developer specifies the resource that the application needs to dynamically load. The developer can specify two kind of resources: local or remote. If the resource is *local*, the developer needs to specify a (local) file path, and the retrieval step is skipped (since the code component is already stored on the device). If the resource is *remote*, the developer needs to specify a URL, and the application will then retrieve the code component (via HTTP or HTTPS, depending on the URL). Note that our protocol allows the retrieval of code through HTTP. This provides flexibility to the developer and, in some cases, also performance improvements. Note that this choice does not introduce a security vulnerability, since the integrity of each component is verified before it is loaded and executed by the application (see Step 5). That is, even if an attacker succeeds in mounting a MITM attack, she will not be able to achieve arbitrary code execution within the context of the application. Moreover, note that the attacker cannot successfully mount an attack even in the scenario where she previously compromised the server hosting the code component.

Step 2: Code Storage. In order to *securely* store the retrieved code components, our protocol places them in an application-private directory, which an external, potentially-malicious application cannot tamper with. This is important since, even if the integrity of the retrieved code component is verified in Step 5, an attacker might exploit a race condition (between the *integrity check* and the *code loading* step) to invalidate the result of the integrity check. This choice also allows us to have a *trusted* cache, so that we do not need to perform an integrity check on code components that have already been verified.

Step 3: Certificate Location Resolution. To verify the integrity of the code components that need to be dynamically loaded, our library requires that each of these components is cryptographically signed. In turn, to establish the authenticity of this signature, our library needs to retrieve the certificate of the developer that signed and published a given code component. To specify the location of a certificate, the developer has two options: she can either (1) construct a URL by reversing the package name of the target class to load, or (2) she can specify a static mapping (through a configuration option) between the package name of the resource to be loaded and the remote location of the associated certificate. Although the first method is very simple, it might not grant enough flexibility in certain scenarios, since it requires the developer to satisfy tight constraints on web domain names. We believe our second option to be the sweet spot between flexibility and practicality: it requires a little extra effort from the developer, but it makes the deployment much easier. From a security point of view, both these methods are equivalent: In either case, the mapping is stored in the code and memory of the application. Thus, according to our threat model, an external, malicious application cannot tamper with this important component of our protocol.

Step 4: Certificate Retrieval. Since the certificate located during the previous step acts as the single *trusted element* of our system, we require it to be retrieved through a secure connection (e.g., through HTTPS), which, according to our threat model, a malicious attacker cannot tamper with. Once the certificate is fetched, our approach requires to store it into an application-private folder (similarly to

the downloaded code components), so that an attacker cannot tamper with it. Also in this case, we introduce caching strategies to avoid to repeatedly fetch the same certificate multiple times.

Note how our current solution proposes to store these certificates *remotely*, on a network endpoint, as opposed to store them *within* the application itself. On the one hand, this choice offers more flexibility to a developer, since she has the possibility to easily revoke and replace the *valid* certificates. On the other hand, this choice requires Internet connectivity to fetch the certificates. As a technique to minimize the impact of this choice, our approach fetches all the needed certificates just after the installation of the application — in case of a regular installation through Google Play, the device is guaranteed to have network connectivity. Nonetheless, we note that, in certain scenarios, storing all the certificates *locally* could be a viable solutions and, for this reason, we plan to implement support for this solution as well.

Step 5: Signature Verification. Our system checks the integrity of each code component before it is dynamically loaded. The integrity check is implemented by verifying the signature against the retrieved certificate. If the signature of any of the entries does not match, the entire code component is rejected, and our library aborts the operation. This effectively prevents the application to inadvertently load and execute *untrusted* code. Moreover, for performance reasons, we designed a caching strategy to benefit from the results of previous signature verifications, so that each container needs to be verified only once, independently from the number of load operations on it.

We note that our protocol assumes that it is the developer's responsibility to (1) properly sign the code to be dynamically loaded, and (2) define a proper mapping between code component and certificates. Since developers are already accustomed to the concept of *code signing* (as this is required by the Android release procedure of non-dynamically-loaded code), we believe this is a small burden for developers. Moreover, both the *code signing* and *certificate creation* steps can be easily automated and integrated into software development tools. This intuition is confirmed by the results of our user study, discussed in Subsection 6.1.

4.3 Repackaging Tool

In this section we present the high-level details of our app repackaging tool. This tool takes as input an application that implements a DCL-based functionality using the native API, and it returns, as output, a semantically equivalent application, in a way that all the DCL-based functionality are ported to use our Grab 'n Run library.

Our repackaging tool works in five main steps. Here we present the high-level goal of each of them, while we discuss the technical details in the next section.

1. **Unpacking.** The tool unpacks the application package (APK), parses its *manifest*, and disassembles its `classes.dex`, which contains the Dalvik bytecode of the application.
2. **Manifest Update.** The manifest is modified to ensure it includes additional permissions required by our library.
3. **Call Site Identification.** The tool identifies all the locations in the code that performs DCL. This is done by scanning the application's bytecode and by looking for the invocation (through the `invoke` bytecode instruction) of a DCL native API (e.g., `loadClass`). We refer to these code locations as *sensitive call sites*.
4. **Patching.** For each of the call site, the tool patches the bytecode such that our library is used instead. This patching follows the package-to-certificate mapping specified by the user. Note that it is possible to automatically and reliably patch the application bytecode as the interface of our API is very similar to the interface of the native one.
5. **Reassembly.** Finally, the tool reassembles the application bytecode and manifest in a new APK. This new application is semantically equivalent to the original one, but all its DCL-related functionality are now *guarded* by our code verification protocol.

5. GRAB 'N RUN IMPLEMENTATION

In this section we present the two tools that we developed as a proof-of-concept for our approach: the first one is a library, which implements our code verification protocol, as described in Subsection 4.2, whereas the second one is the repackaging tool that follows the procedure outlined in Subsection 4.3.

5.1 The SecureDexClassLoader API

We implemented Grab 'n Run in an open-source Java library, compatible with both the Android Development Tool (ADT), and the Android Studio (AS) IDE, as a drop-in tool that developers can easily plug-in in their Android projects. Our library is online since Nov 26, 2014, with the goal of helping developers to implement DCL securely. Since then, more than 250 GitHub users have already put a "star" on our repository, and more than 30 developers have forked it.

This section describes an implementation of the code verification protocol. In our proof-of-concept library, we implemented a new version of `DexClassLoader`, which we called `SecureDexClassLoader`. We chose this API since it is one of the most commonly used to load code dynamically (as documented by [9], 5.01% out of 1,632 popular applications with more than one million downloads made use of this API), as well as one whose use is often misunderstood by developers (the same source measured that 37,35% of the applications in the previous group were affected by a security vulnerability).

Original API. More in the details, `DexClassLoader`, which was introduced in APIv3 (and is still present unmodified as of APIv21), can load classes from external DEX (Dalvik EXecutable) files stored in APK or JAR archives. As exemplified in Listing 1, the `DexClassLoader` constructor requires a list of URIs pointing to such code containers stored on the device's file-system. Next, this API retrieves the code and caches its optimized version (i.e., ODEX in case of Dalvik, ELF in case of ART) into a local directory also provided by the developer as input, on which no restriction on the accesses is enforced. Finally, by invoking the `loadClass()` method, the developer can dynamically load any class implemented within the code container. Note that, by design, `DexClassLoader` does not perform any integrity or authentication checks. Note also that the retrieval and storage of the code container (that usually precede the actual code loading), is "up to the developer", and this is why we omitted these steps in Listing 1.

Grab 'n Run API. At its core, our library wraps the `DexClassLoader` class to add the missing security checks

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();
```

introduced by our protocol. Table 1 maps each step of our verification protocol to the corresponding classes of the Grab 'n Run library that execute them. 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 API.

We can logically map the main functionalities of `SecureLoaderFactory` to the ones of the constructor of `DexClassLoader` and the `loadClass()` method of `DexClassLoader` to the corresponding one in `SecureDexClassLoader`. These two classes are used as follows by the developer:

1. **Initialize `SecureLoaderFactory`.** At first, the developer initializes an instance of `SecureLoaderFactory`, which requires a reference to the current `Activity` object.
2. **Initialize `SecureDexClassLoader`.** The method `createDexClassLoader()` implemented by the `SecureLoaderFactory` object, returns an instance of `SecureDexClassLoader`. Alongside the usual parameters required by `DexClassLoader`'s constructor, the developer must pass an extra argument, which is the package-to-certificate associative map that links package names to the URL of the remote certificate to verify each code signature.
3. **Code Loading.** Next, the developer uses the `loadClass()` method on `SecureDexClassLoader` to load a given class, identified by its full name. `SecureDexClassLoader` returns a class object only if the class is implemented within a successfully-verified JAR or APK code container.

Listing 2 shows an example of this procedure. Note that this listing implements the same exact functionality as in Listing 1, and it also additionally retrieves and stores of the code component to be loaded. Listing 2 also shows how to properly handle the scenario where it is not possible to verify the code component's integrity.

Compared to the native `DexClassLoader`, `SecureDexClassLoader` securely fetches and stores the code containers to be loaded, and caches them to minimize the retrieval of remote resources (i.e., code containers, and certificates for the signature verification). The signature is verified concurrently in case of multiple code containers. Finally, Table 2 summarizes the main differences between the original API and the one proposed by Grab 'n Run.

Table 1: Mapping between the verification protocol's steps (Subsection 4.2) and classes in our library. Classes in brackets embed those that actually implement the corresponding step.

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

5.2 Repackaging Tool

We implemented the procedure outlined in Subsection 4.3 in Python. The prototype relies on Androguard [1], an open-source reverse-engineering and static-analysis suite for Android application analysis, and `apktool` [2], which offers APK repackaging functionality.

Configuration. As input, the developer provides two configuration options along with the APK to be patched: (i) the code containers used as sources for DCL (either with a local reference on the file-system, or with a remote URL pointing to each archive), and (ii) the binding between each container and the trusted certificate that must be used for its signature verification (provided via a remote URL with HTTPS protocol).

Unpacking and Reassembly. Using Androguard, we perform static analysis on the target APK to determine whether it uses `DexClassLoader` API. If this is the case, we query Androguard to obtain the list of the points to patch, and the required permissions of the APK. We use `apktool` to unpack the APK and disassemble the classes inside of it: The result is a collection of decoded resources (including the Manifest) and a set of classes written in Smali — an intermediate, and human-readable intermediate representation of Dalvik bytecode. If missing, we append three extra permissions required by the Grab 'n Run library to the Manifest: `ACCESS_NETWORK_STATE`, and `INTERNET` for retrieving code containers, and certificates from remote endpoints, and `READ_EXTERNAL_STORAGE` to transfer code containers stored on the SD into an application-private folder prior to the actual verification. Once we obtain the patched bytecode,

Table 2: Features comparison.

Feature	Original	Grab 'n Run
Fetch code from remote URL	✗	✓
Store code in app-private dir.	✗	✓
Code verification (integrity and developer authenticity)	✗	✓
Dynamic code loading	✓	✓

we use `apktool` once again to reassemble the package.

Patching. Patching a Smali class is, in the general case, quite challenging. Our solution is based on the following two observations: 1) adding a new class is trivial (in fact, in Smali, each class is defined in a separate file); 2) substituting an invocation to a method M with the invocation to a different method M' is trivial, as long as the two methods M and M' have the same exact signature (i.e., number and type of arguments and return value). Thus, our system first creates a new static class that implements a series of methods that have the same signature of the methods we need to patch (e.g., the `loadClass()` method). Functionally, each of these new methods constitute a wrapper for their respective method that implements the required security checks and pass the required additional arguments. Once these methods are defined, it is then trivial to patch the original Smali code to redirect each invoke to a relevant method to its respective wrapper method.

6. EVALUATION

In this section, we report on our effort to evaluate the security and practicality aspects, when actually used by non-security-conscious Android developers. In particular, we evaluated these aspects by performing a user study with 12 Android developers[2]. Our results show that our library provides tangible benefits under both the security and practicality aspects. As the last step of our evaluation, we also present the results we obtained while measuring our library's performance overhead, which, as expected, resulted to be negligible.

We also attempted to collect a corpus of insecure real world applications to patch with the repackaging tool. Unfortunately, almost all of these applications make use of unsigned code containers that we were not in a position to sign, and/or retrieve from a secure endpoint (e.g., the unsigned container is embedded in the firmware of the phone by the vendor). Thus, even if the patching process with the repackaging tool completes successfully, we were not able to test whether the functionality of the application was preserved since Grab 'n Run prevents DCL from unsigned code containers.

6.1 User Study

In this section we describe the user study we conducted, and we discuss the results we obtained.

Participants Recruitment. For our user study, we first tried to recruit as many participants as possible by posting an announcement on several public Android-related mailing lists, as well as on mailing lists internal to the authors' affiliations. In total, we were able to recruit 12 participants. Ideally, we would have successfully recruited many more participants. However, we required the participants to have at least a minimal prior knowledge in Android application development, and we also did advertise that the experiment could have taken up to few hours: While we would have probably recruited more participants if we were not specifying these requirements, we believe these were necessary to obtain meaningful results.

These participants had different ranks of expertise in Android developing: some of them had developed only a couple of toy applications, whereas some others develop Android applications on a regular basis. It is important to note that none of the selected participants had never used `DexClassLoader`, or any other API for DCL before this experiment.

[2]IRB approval was obtained by our institution.

Table 3: Security bugs introduced by the developers of our user study using `DexClassLoader` API. The table correlates each error with a triggering example and the number of developers that introduced it.

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)

Experiment Setup. We gave each developer access to a skeleton application that we developed specifically for this project. Then, we asked them to perform two tasks. First, we asked them to implement a functionality involving DCL using the native Android API, `DexClassLoader`. In particular, we asked them to fetch a remote code component, store it on the device, and finally dynamically load a class defined in the downloaded component. As the second task, we asked them to implement the very same functionality, but, this time, using our `SecureDexClassLoader`. At the end of the experiment, the developers had to send us the source code of both implementations, and compile a form that asked questions about the two different APIs. In particular, this form collected feedback from the participants related to several aspects, such as efficiency, code readability, security, and maintainability.

During the experiment, the developers were left free to consult any online resources, including, obviously, the official Android documentation. Moreover, 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. This was done to encourage the participants to think about the security aspect, among many others (e.g., efficiency, code readability, and maintainability). We note that the participants were not required to setup the certificate used to validate the code, nor the endpoint to store such a certificate. Although this reduces the complexity of the task, we believe these two steps represent one of the most common use cases, especially in a scenario where a developer wants to securely load dynamic code from a third-party container.

Security Benefits Results. Table 3 reports our study's results related to the security aspect. In particular, it shows how many developers committed which kind of security-related error when using the native Android API, `DexClassLoader`. This data was collected by manually inspecting the source code of all the submitted implementations. As the table shows, 75% of the developers failed to fetch the remote code container securely (i.e., they used an HTTP connection, as opposed to a HTTPS one); half of them failed to securely store the fetched code component (i.e., they stored it on the external storage). An alarming result is that not even a single participant thought about checking the integrity of the downloaded component, or the developer's identity of

the fetched container. Thus, all 12 developers introduced a severe security vulnerability, even when implementing a *simple* functionality, and even after telling them to take into account the security aspect. This confirms, once again, that securely implement DCL-based functionalities is really challenging.

Another very interesting aspect is that none of the developers unsafely stored the optimized version of the code component in a world-writable portion of the file system. We believe this is the case thanks to the fact that this security-related aspect is specifically mentioned in the official documentation of the `DexClassLoader`, and all developers opted to adopt this best practice. We believe this is interesting for two reasons: First, it shows that the developers of our user study did indeed care about the security aspect; Second, it shows that proper documentation is indeed effective in helping developers to not introduce security vulnerabilities.

As the second part of the experiment, the participants then used our newly-developed API, `SecureDexClassLoader`, to implement the very same functionality. In this case, none of the developer introduced a security vulnerability. While this is not entirely surprising (since our library prevents this class of vulnerabilities *by design*), it does confirm that the adoption of our work would directly benefit the Android developers community.

Practicality Benefits Results. To evaluate the practicality aspect of our work, we asked all participants to compile a questionnaire, which contained questions related to many aspects of the experiment. Table 4 shows a summary of the answers we received from our participants. Overall, our work received very positive feedback. First, all developers felt that the overhead of learning to use our library (starting from the native API) was negligible; Second, 83% of the involved participants thought that the second snippet of code, which relies on our library, would be easier to maintain and modify, and that it is actually simpler to read compared to the first one; Third, 91% of developers also asserts that the code snippet using Grab 'n Run was easier to implement, looks more flexible (e.g., automatically fetch remote containers, store them appropriately), and secure (e.g., perform integrity checks at run time on code containers) than the native API. We believe these are really encouraging results.

Finally, we also noted that the average estimate of time spent for understanding and implementing the dynamic code loading functionality is about 139 minutes when the developers used `DexClassLoader`, while it dropped to just 37 minutes when using `SecureDexClassLoader`. While this is also a very good result, we acknowledge that our experiment setting might have significantly influenced this outcome. In fact, we asked the developers to implement the required functionality with our library only after they have already implemented it by using the native API. That is, it might be possible that the striking difference in the average time is due to the fact that the developers acquired part of the required knowledge during the first phase of the experiment.

Table 4: Summary of the feedback provided by the participants of our user study.

Users evaluation: `DexClassLoader` (Native API) vs `SecureDexClassLoader` (Grab 'n Run API)	
Average time for development *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` *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` *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 *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 *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 *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 *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 *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 *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

6.2 Performance Evaluation

Experiment Setup. To measure the performance overhead introduced by `SecureDexClassLoader`, we developed a simple profiling application that dynamically loads two classes from an APK container using first `DexClassLoader`, and then `SecureDexClassLoader` API. In particular, we instrumented both the application and our library code to log the initial and final timestamps of the operations related to the DCL step. Since `DexClassLoader` cannot automatically fetch remote code components, we decided to use a local container as the source for DCL for both systems, so to have a fairer comparison. Moreover, as part of the experiment, we also considered the most common scenario in which classes have been already loaded from the source container during a previous run of the application. In other words, we considered the scenario where `DexClassLoader` has already prepared and stored the optimized version of this archive in the cache folder, and `SecureDexClassLoader` has already fetched the certificate used for the container signature verification. The measurement was performed by executing and profiling the application over 100 iterations on a Nexus 5 device.

Performance Overhead Results. Table 5 reports our results and a comparison between the native API and ours. We aggregated the measured time-stamps depending on the experiment phase, and we computed mean, median, and standard deviation over all the iterations. The measurements outlines that `SecureDexClassLoader` introduces an overhead during its "Setup" phase that is mostly due to the presence of the signature verification step, which is missing in native `DexClassLoader`. However, thanks to the caching mechanism,

Table 5: Performance evaluation results.

Phase	Mean [ms]	Median [ms]	Std Deviation [ms]
`DexClassLoader`	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`	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

this costly verification is performed only once, during the app initialization. After this setup phase, the overhead introduced by our library is negligible.

7. LIMITATIONS

In the previous sections, we described how we believe our approach to be feasible, secure, and practical. Nonetheless, our work is affected by a few limitations, which we enumerate in this section.

Uniqueness of Package Name. The current implementation of our library assumes that a package name uniquely identifies a code container. This implies that our library does not currently allow an application to securely load two code containers that have the same package name. We believe that this limitation does not introduce any significant practicality issue. In fact, the Google Play Store already enforces a similar policy: a given package name uniquely identifies an application on the market, and two different applications are not allowed to have the same package name. Moreover, in the unlikely scenario where a developer needs to load two code containers that, incidentally, have the same package name, one simple workaround is to simply change the package name of one of the two containers.

Limited API Coverage. The work presented in this paper mainly focuses on providing a secure implementation of the existing `DexClassLoader` native API. However, there are several other APIs for dynamically loading code in Android. For example, an application might load additional code through the `PathClassLoader` or the `android.content.Context.createPackageContext` APIs. Even if our library does not currently provide a secure replacement for these APIs, we believe our work can be extended. However, we acknowledge that, for some of these APIs, the modifications might be non-trivial.

Challenges in Code Reuse. One use case of DCL is to allow for code reuse across different applications. However, our protocol relies on safely storing each code container in an application-private folder, and it makes the implementation of code reuse more challenging. As a possible solution, a developer that wants to *share* code components among two of its applications, could sign the two applications with the same private key: in this case, the Android framework would assign to the two applications the same Linux user, and the two applications would consequently have access to the same folders and code components, thus making code reuse possible. An alternative solution would be to set the permissions of the dynamically-fetched code component to *read-only* for other applications. However, it is not clear how this could be implemented through the current Android permission system.

8. RELATED WORK

This section describes the relevant related work with a brief discussion of the differences with Grab 'n Run.

Remote Code Execution. Poeplau et al. [9] were the first to study the security problems introduced by DCL. After reviewing the mistakes made by the developers of the vulnerable apps among the top 500 on the Google Play Store, they proposed a series of modifications to the Android framework. They also introduce the notion of an external service in charge of analyzing any dynamically loaded code component. As for [5], this solution is valid, but it is affected by practicality issues. In fact, the required modifications are not trivial, and they involve low-level implementation details of the Dalvik VM. Moreover, the massive fragmentation of the different Android versions make these modifications less likely to be adopted in the short term [13]. Another aspect that might incur into practicality issues is that the proposed solution requires that the developers to go through a time-consuming vetting process for each of the additional components, making this solution unideal when a developer would need to flexibly deploy additional components.

In contrast, Grab 'n Run leaves the underlying framework untouched and wraps the security-sensitive APIs, exposing a cleaner, simpler API to the developers. Differently from [9], we focused our work on the code-verification protocol, the implementation of Grab 'n Run in an open-source library and patching tool, leaving the porting to DCL of native (e.g., ARM, x86) code to future works. While we acknowledge that our solution does not protect against DCL usage in a malicious scenario (e.g., an application that loads only at runtime the malicious payload), we believe our work makes a significant step forward to prevent benign applications to contain vulnerable components.

Code Verification. Vidas and Christin [12] propose a simple mechanism that alleviates the specific problem of verifying the authenticity of an app, to protect the user from repackaged apps that contain malicious code. Their approach is based on creating a simple public-key infrastructure backed by the domain name system. To allow Android to verify the authenticity of a given app, the developer must place the signing certificate on the DKIM or TXT record of their own domain name. Such domain name is required to match the reversed string extracted from the Java package name of the app itself. Thus, if a developer wants to execute `MalwareMainActivity.evil.com` or want to repackage a benign application with malicious code, she would have to possess the signing certificate and control the authoritative name servers accordingly.

Differently from Grab 'n Run, their scheme is not always practical because of the package name restrictions. This is mainly due to the fact that their approach is tailored to protect against repackaging. However, because of this limitation, the approach cannot be easily ported to address the DCL-related security vulnerabilities discussed in Section 2. In fact, a developer could not securely load code with a package name of a domain they do not own, and this would not allow them to "seal" third-party apps or libraries.

More importantly, in their work, the signature is checked only upon installation, and not when the code component is actually used. This design choice is mainly motivated by the fact that there might not be Internet connectivity at runtime. However, this implies that any code change between

installation and execution, which would break the signature, would pass unnoticed. Differently, our system first makes sure the code component is stored in a safe location, and it then verifies the integrity of the component just before it

Vulnerable Usage of Security-related APIs. Our work is motivated by the observation that apps' developers, when using security-related APIs or functionalities, often make severe mistakes and, as a consequence, introduce severe security vulnerabilities. Unfortunately, this aspect seems to affect not only DCL-related APIs, but also other kind of APIs. For example, Egele et al. [4] have recently studied the use of cryptographic APIs in Android applications and found out that the developers that publish on the Google Play Store made at least one mistake in 88% of the apps (they reviewed a sample of 11,748 apps).

Despite the abundance of cryptographic libraries, which strive to abstract away the details and remove the burden on the developers' side, many security flaws are still due to inaccurate use of such libraries or naïve practices. In fact, according to a recent empirical measurement by Fahl et al. [5], 17.2% of a sample of 13,500 Google Play apps failed to verify the SSL certificate, of which 1,070 include critical code, 790 accept all certificates, and 284 accept all hostnames. Similarly to [9], the authors of this work propose a patch to the Android framework to close the gap that caused the errors.

9. CONCLUSIONS AND FUTURE WORK

In this work, we proposed, designed, implemented, and tested Grab 'n Run, a novel code verification protocol and a series of supporting libraries, APIs, and components, that aim to address the pressing security issues related to the *unsafe* implementation of functionality based on *dynamic code loading*. We extensively evaluated our work through a user study with 12 participants: Our results show that our library is both *secure*, since many security vulnerabilities are prevented *by design*, and *practical*, since Grab 'n Run is thought as a drop-in Java library, very similar to the already-existing Android API. Our results also show that it is really challenging to securely implement DCL-based functionality by using the current Android API. Moreover, we also implemented a "repackaging tool," which can automatically *patch* potentially-vulnerable Android applications so that they use our newly-developed library. Finally, in the spirit of open science and to benefit the Android developer community, we released our library as fully open source, and we already received very positive feedback. As part of our future efforts, we will implement support for new APIs, and we will perform additional user studies to understand how to improve our work and to increase its adoption.

10. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback. This work is based on research sponsored by DARPA under agreements number FA8750-12-2-0101 and FA8750-15-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The work was also supported by the Office of Naval Research (ONR) under grant N000140911042, the Army Research Office (ARO) under grant W911NF0910553, by the National Science Foundation under grant CNS-1408632, Secure Business Austria, and MIUR FACE Project No. RBFR13AJFT.

References

[1] Androguard, project home page. URL `https://code.google.com/p/androguard/`.

[2] Apktool, project home page. URL `https://code.google.com/p/android-apktool/`.

[3] AppBrain. Number of available Android applications. `http://www.appbrain.com/stats/number-of-android-apps`.

[4] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[5] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking SSL development in an appified world. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[6] Google. Android Permissions. `http://developer.android.com/reference/android/Manifest.permission.html#WRITE_EXTERNAL_STORAGE`.

[7] D. Maier, T. Muller, and M. Protsenko. Divide-and-Conquer: Why Android Malware Cannot Be Stopped. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, 2014.

[8] D. Maier, M. Protsenko, and T. Muller. A Game of Droid and Mouse: The Threat of Split-Personality Malware on Android. *Computers & Security*, 2015.

[9] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[10] singwhatiwanna Github User. Dynamic Load Framework for Android. `https://github.com/singwhatiwanna/dynamic-load-apk/blob/master/README-en.md`, 2015.

[11] Strategy Analytics. Android Captures Record 85 Percent Share of Global Smartphone Shipments in Q2 2014. http://prnewswire.com/news-releases/strategy-analytics-android-captures-record-85-percent-share-of-global-smartphone-shipments-in-q2-2014-269301171.html, August 2014.

[12] T. Vidas and N. Christin. Sweetening android lemon markets: Measuring and combating malware in application marketplaces. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[13] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[14] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2012.