Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications

Sebastian Poeplau*†, Yanick Fratantonio*, Antonio Bianchi*, Christopher Kruegel*, Giovanni Vigna*
*UC Santa Barbara
Santa Barbara, CA, USA
Email: {yanick,antoniob,chris,vigna}@cs.ucsb.edu
†University of Bonn
Bonn, Germany
Email: poeplau@cs.uni-bonn.de

***Abstract*—The design of the Android system allows applications to load additional code from external sources at runtime. On the one hand, malware can use this capability to add malicious functionality after it has been inspected by an application store or anti-virus engine at installation time. On the other hand, developers of benign applications can inadvertently introduce vulnerabilities. In this paper, we systematically analyze the security implications of the ability to load additional code in Android. We developed a static analysis tool to automatically detect attempts to load external code using static analysis techniques, and we performed a large-scale study of 1,632 popular applications from the Google Play store, showing that loading external code in an insecure way is a problem in as much as 9.25% of those applications and even 16% of the top 50 free applications. We also show how malware can use code-loading techniques to avoid detection by exploiting a conceptual weakness in current Android malware protection. Finally, we propose modifications to the Android framework that enforce integrity checks on code to mitigate the threats imposed by the ability to load external code.**

I. Introduction

Recent years have seen the Android platform gain more and more popularity, and a considerable number of mobile phones and tablet computers ship with Android. Specifically, Google announced at its developer conference Google I/O in May 2013 that 900 million Android installations have been activated since the launch of the system in 2008. The large number of devices running the Android operating system provides great opportunities for developers to reach a broad audience while only having to develop for a single platform. Unfortunately, the same economic incentives appeal to criminals as well. By targeting Android, they have the opportunity to conduct malicious activity on millions of devices. Accordingly, the amount of malware for and attacks against Android is steadily increasing [24], [34]. For example, malicious applications steal users' private information or use cost-sensitive functionality such as premium SMS to generate revenue for the attackers.

In order to counter the spread of malicious content in Android's main application store, the Google Play store,[1] Google introduced a vetting mechanism for applications in 2012 [21]. This system, called Google Bouncer, analyzes every application for malicious functionality that is submitted to Google's store. The analysis is performed offline, i.e., applications are analyzed in a centralized location before being admitted to the store. The alternative would be to conduct the application analysis online, i.e., directly on the users' devices. However, this is impractical due to the large importance of battery life for mobile devices and the access restrictions that the system enforces for all applications, including anti-virus software. The inner workings of the Bouncer are not precisely known to the public, but experiments by Oberheide and Miller indicate that Google uses dynamic analysis to examine applications [25].

A powerful way for malware to circumvent the Bouncer is by loading external code at runtime. For example, imagine an application that pretends to be a simple game. During the Bouncer's analysis, the application does not expose any malicious behavior, and even its code does not contain any malicious functionality. But once the application has been approved by the Bouncer, admitted to the store and installed by users, it starts to download and execute additional code that performs harmful activities. We show that using this technique, applications are able to evade detection by the Bouncer and several mobile anti-virus products.

We would like to stress that such evasion techniques that rely on the ability to load arbitrary code at runtime are more powerful than the ones presented in the past. In fact, this class of techniques exploits a conceptual problem of offline vetting mechanisms: First, these mechanisms can never be sure that they see all the code that an application will eventually execute on users' devices. The application could just download additional code from the Internet on any device and at any point in time. Secondly, techniques to load code at runtime are

[1] https://play.google.com/store/apps

NDSS '14, 23-26 February 2014, San Diego, CA, USA

http://dx.doi.org/doi-info-to-be-provided-later

often used by benign applications as well. Therefore, vetting systems cannot use the mere presence of such functionality as a feature to detect malware.

In this paper, we present a large-scale study that analyzes the use of code-loading techniques in applications on Google Play. We find a surprisingly large number of applications that use the techniques for a variety of legitimate reasons. In particular, the results emphasize that the mere ability of applications to load external code is not an indication of malicious intentions.

Unfortunately, the existence of mechanisms that allow applications to load code from external sources in Android introduces yet another problem: Benign applications use the techniques for legitimate reasons, but their use is prone to introducing vulnerabilities. In fact, we found that Android does not enforce appropriate security checks on the external code, and application developers are often unaware of the risks or do not succeed in properly implementing custom protection mechanisms.

As part of our experimental study, we apply static analysis techniques to automatically detect if a given application is able to retrieve and execute additional code and if it does so in an unsafe way. To our surprise, we find severe vulnerabilities both in Android applications and in third-party frameworks. We present two example exploits, which would enable an attacker to mount code injection attacks against more than 30,000 applications on Google Play, affecting several tens of millions of users. We notified the affected developers and collaborated with one framework vendor in an effort to mitigate the risk to the users as quickly as possible.

As a remedy to both types of attacks – malware escaping offline analysis as well as code injection into benign applications – we propose a modification of Android's Dalvik virtual machine. Our protection system aims to add the missing mandatory checks of code from external sources to the operating system. The checks are computationally cheap and do not require extensive changes to the Android system, while effectively protecting the users from attacks that are currently possible due to applications' ability to load external code.

Our main contributions can be summarized as follows:

1) We systematically analyze the Android platform for techniques that allow applications to load code at runtime, and examine their respective security implications.
2) We develop an efficient static-analysis tool that automatically detects problematic behavior associated with dynamic code loading in applications. Our tool can detect vulnerabilities in benign applications as well as support the analysis of potentially malicious applications.
3) We conduct an analysis of 1,632 popular applications on Google Play, each with more than one million installations, revealing that 151 (9.25%) of them are vulnerable to code injection attacks.
4) We propose and implement a modification to Android's Dalvik VM, which prevents all attacks that are possible due to external code loading.

II. Background and threat model

In this section, we introduce concepts and terminology related to Android security on which we will rely in the remainder of this paper, and we present the threat model that our work is based on.

A. Android

The following concepts of the Android system in general, and Android security in particular, are important in the context of our work.

1) Android permissions: Android restricts the access of applications to sensitive functionality by means of *permissions*. Permissions regulate access to sensitive APIs that can cause financial damage (e.g., SMS services and phone calls), compromise the user's privacy (e.g., record sound or video, use the Internet) or negatively affect device usability (e.g., lock the screen). Each application has to declare the set of permissions that it requires.

Whenever an application is installed, Android presents the user with a list of the requested permissions and asks for approval. The user can either accept all permissions or cancel the installation. It is not currently possible to selectively remove permissions from applications.

Internally, Android enforces permissions by using high-level checks in Java code as well as common Linux security concepts, e.g., file permissions, user and group IDs.

2) Application stores: Applications are commonly installed from *stores*, entities that offer large collections of applications for users to download. The main application store is the Google Play store – most Android devices have the application to access this store preinstalled. At a press event in July 2013, Google announced that the Google Play store offers more than one million applications.

Android provides the option to install applications that access alternative stores, such as the Amazon application store. Note that the system treats the store that the device manufacturer prefers differently from others: In order to install APKs from other stores, users have to enable the so-called *sideloading* setting, which allows the user to install APKs from arbitrary sources, while otherwise the system rejects anything that does not originate from the preferred store.

Since the amount of malware on Google Play was growing, Google introduced a system called Google Bouncer in February 2012 [21]. The Bouncer checks every application that is submitted to Google Play for malicious behavior. Its internal workings are not precisely known, but there have been successful attempts to circumvent or attack it, e.g., by fingerprinting the analysis system that the Bouncer is based on [25]. As mentioned previously, the experiments by Oberheide and Miller suggest that the main component of the Bouncer is dynamic analysis conducted in Qemu.

We will show that by loading code from the Internet at runtime an attacker can circumvent the checks imposed by the Bouncer.

3) Native code: Android applications are usually written in Java and compiled to Dalvik bytecode. For computationally expensive tasks, Android provides the option to run so-called *native code*, i.e., machine code for the device's processor. The common way of invoking native code on Android is through the *Java Native Interface (JNI)*, a standardized interface for interaction between Java and native code. At the system level, loading native code means that Dalvik, Android's virtual machine, loads a Linux shared object and allows Java code to make calls to the contained native functions (and vice versa).

Native code runs in the same sandbox as Java code. Specifically, the same permissions are enforced on native code as on Java code.

4) Application frameworks: When developing applications, many developers rely on *frameworks*. Such frameworks provide additional functionality, such as components that download and display advertisements within the application.

Different developers usually maintain frameworks and applications, so that their update cycles generally do not coincide. Updated applications can be published in application stores and pushed to users immediately. New versions of frameworks, however, first need to be included in all affected applications by the application developers before they can be installed on the users' devices with the next version of the application. This means that it is considerably more difficult for framework developers to deploy updates than it is for application developers.

B. Threat model

Before we examine the attacks and our proposed countermeasures in more detail, we outline the threat model that our work is based on. We consider two different attack scenarios:

1) Evasion of offline analysis systems: In this first scenario, an attacker creates a malicious application and tries to publish it in an application store such as Google Play. In order to avoid detection, she designs the application in such a way that it does not contain any clearly malicious code itself, but rather downloads additional code after being installed on a device. Since most of the application analysis mechanisms work in a central place, such as the various application stores, the malicious code can easily circumvent the checks. Note how it is conceptually impossible for offline analysis systems, such as the Google Bouncer, to detect the malicious functionality: The code that they analyze does not contain anything malicious, and they have no way of knowing what additional code the application might download at a later point in time. Since benign applications use code-loading techniques as well, the detection system cannot reject applications that load external code per se. An analysis of a large corpus of Android malicious applications conducted by Zhou and Jiang uncovered malware samples that already use this technique to evade detection [37]. For demonstration purposes, we designed an application that makes use of the same approach and uploaded it to Google Play to confirm that the application can be distributed without Google checking the additional code that might be downloaded later (see Section III-D1 for details).

2) Code injection against benign applications: The second scenario involves applications that load additional code for benign reasons (see Section III-B for a discussion of possible motives). As the operating system does not enforce security checks on the loaded code, an attacker could replace the original code with malicious one. At this point, a vulnerable application would load the malicious code, not recognizing that it is different from the intended one, and executes it. This gives attackers a way to run arbitrary code in the context of an application on the user's device (in particular, with the application's permissions and with full access to its data). In Section III-D2, we discuss different ways to exploit such vulnerabilities, and, in Section V-B, we present attacks against two popular application frameworks.

The protection system that we propose is designed to defeat both types of threats.

III. Investigation of code loading

In this section, we first examine different techniques that applications can use to load external code at runtime, and briefly outline why they can cause security issues. We then describe reasons for benign applications to load additional code. Finally, we point out typical mistakes of benign applications when using code-loading techniques.

A. Techniques to load code

We identified several ways to let Android load external code on behalf of an application. These techniques can be used by malware to avoid detection, and improper use can make benign applications vulnerable.

1) Class loaders: Class loaders are Java objects that allow programs to load additional classes, i.e., Java code. Android applications can use class loaders to load classes from arbitrary files. The Android class loaders support different file formats (APK, JAR, pure dex files, optimized dex files). They do not impose restrictions on the location or provenance of the file containing the code. For instance, an application can download an APK file from the Internet and use *DexClassLoader* to load the contained classes. By invoking the classes' methods, the application executes the downloaded code.

Benign applications are at risk of code injection, i.e., an attacker can replace the file that is to be loaded with a malicious one, if it is stored in a writable location. The Android system does not check the integrity of class files in any way. In particular, running applications do not require a signature when loading APKs. In Section V-B1, we present an exploit that injects code into an application by hijacking the application's connection to the server that it downloads code from.

2) Package contexts: Whenever Android loads an application, it associates it with a *Context* object. The context, among other things, provides access to the application's resources, such as image files or code. Android provides applications with an API (*createPackageContext*) to create contexts for other applications that are installed on the system, identified by their package name. Using this API, an application can retrieve the context of another application, provided that it knows the other application's package name. By specifying certain flags when creating the package context, the application can cause the system not only to load another application's resources, but also create a class loader for its

code. This way, an application can load the Java classes contained in another application. Under certain circumstances (i.e., if both the flags *CONTEXT_INCLUDE_CODE* and *CONTEXT_IGNORE_SECURITY* are specified in the call to *createPackageContext*), the system does not verify that the applications originate from the same developer or that the application to be loaded satisfies any criteria at all. This means that an application can load and execute any other application's code.

Despite the alarming name of the necessary flag *CONTEXT_IGNORE_SECURITY*, we found a large number of applications using this technique, apparently ignoring the resulting security implications (see Section V for details).

Package contexts offer another way for attackers to inject code into benign applications: If an attacker manages to install an application with the same package name as the expected legitimate one and the loading application is careless in verifying integrity and authenticity, then the attacker's code is executed within the loading application (and with that application's privileges). We found many applications that are vulnerable to this attack, putting millions of users at risk. Even well-known frameworks do not take appropriate measures to avoid attacks. In Section V-B2, we show how to use this attack to exploit any application that is based on a very common app framework in versions from before July 2013.

3) Native code: As described in Section II-A, Android applications are generally written in Java, but they are allowed to execute native code (i.e., ARM assembly for most of the current Android devices) at any time. Using the Java Native Interface (JNI), the Java portion of an application can interact with the native portion, which is commonly developed in C.

As mentioned previously, Android enforces the same privilege checks on native code as on Java code. For instance, an application cannot open an Internet socket from native code without permission to access the Internet. However, there is one distinctive advantage for attackers when running native code: While they have to go through a well-defined API to load code into the Java environment, they can easily load and execute code from native executables in a variety of ways. The fundamental advantage for attackers is that there is no distinction between code and data at the native level, while Java requires an application to explicitly load a class file in order to run its code. This significantly complicates protection against malicious native code.

With respect to attacks against benign applications it should be noted that Android imposes no restrictions on location or provenance of native code that an application loads. So in theory, if attackers can modify the files that benign applications load, they can cause them to load and execute malicious code. We mention this attack only for completeness – in our analysis, we did not find an application that is vulnerable to such an attack, because applications usually only load native code from their own APK (which is not writable for applications) and use the other techniques described in this section if they need to load code from external sources.

4) Runtime.exec: The Java class *Runtime* provides a method *exec*, which allows applications to execute arbitrary binaries. Internally, *exec* is roughly equivalent to the Linux system calls *fork* and *exec*. Thus, it provides a similar functionality as the well-known C function *system()*, which gives a program access to a system shell. Again, the operating system does not check where the binary is located or where it comes from. A malicious application can therefore use system binaries, such as */bin/sh*, to execute arbitrary commands.

5) APK installation: On Android systems, the Package Manager Service is responsible for installing and removing applications. While this service is commonly used as backend for market applications, such as Google Play or Amazon Market, applications can also directly request the installation of APKs. In such a case, the package manager prompts the user whether or not to install the APK, displaying the requested permissions along with information about the application. If the user agrees, the application is installed on the system just like any application downloaded from an application store.

While not as stealthy as the techniques presented above, APK installations still provide attackers with a way to download and install malicious code at runtime. The package manager ensures that APKs carry a valid signature before it installs them, but there are no requirements as to the trust level of the signature. In fact, the package manager accepts any self-signed certificate for use in a package signature. Thus, the signature does not provide any guarantees about the origin of an application. It is only used to determine whether two applications originate from the same developer, e.g., to ensure that the same developer created an application and the alleged update. If an attacker is able to replace the APK that a benign application tries to install with a malicious one, the installing application does not detect the attack unless it implements a custom verification mechanism.

Note that the user has to enable sideloading in the system settings first in order to install applications from any source other than the preferred store of the device manufacturer (cf. Section II-A). However, any user who wants to use an alternative application store has to do the same. In an effort to assist users in the process of setting up their devices, the providers of such application stores usually offer detailed instructions on how to find the sideloading setting, often without warning about potential security implications. For this reason, we can assume that sideloading is enabled on a considerable portion of Android devices.

We found that APK installations are used by many well-known applications. For example, Facebook used direct APK installations rather than regular updates through Google Play in certain cases in April 2013 (further discussed below).

B. Motivation for loading external code

We found that many benign applications load additional code at runtime. There are legitimate reasons for this behavior, which we will present in this section.

1) A/B testing and beta testing: It is common for software manufacturers to test modified versions of their applications with a small subset of their users in order to evaluate user satisfaction in comparison to the established version. This approach is usually referred to as A/B testing. Until recently, Google Play did not offer a way for developers to roll out an update only to a portion of their users, so some developers used the techniques presented previously to achieve the same

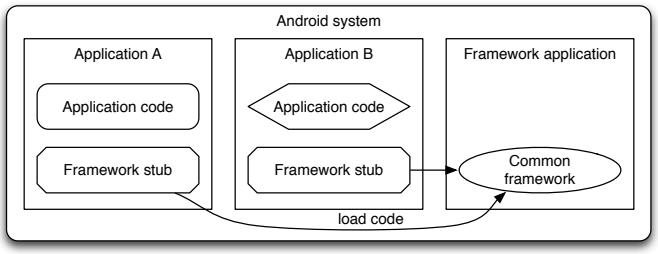

Fig. 1. Several applications load code from a common framework.

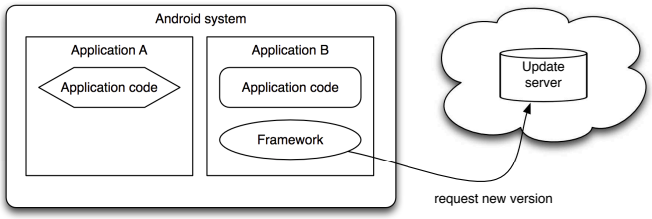

Fig. 2. Some frameworks implement self-update functionality, downloading and executing code from remote computers.

effect. Most notably, Facebook used APK installations starting in April 2013 to install updates on selected users' devices. There are also several frameworks that offer support for beta testing to application developers, often based on APK installations (see Section V). In June 2013, Google announced the introduction of A/B testing mechanisms for the Google Play store at the developer conference Google I/O. This will allow application developers to refrain from implementing custom (and potentially insecure) techniques.

2) Common frameworks: On most desktop operating systems, including Windows and Linux, it is common to install libraries centrally on the system rather than bundling them separately with every binary that uses them. This is useful to save disk space and to avoid conflicting versions of libraries on the same system. The same concept can be employed for Android applications. In this scenario, multiple applications are based on the same framework, which is installed on the device as a separate application. All the applications based on the framework contain stub code (usually provided by the framework's developers) that causes them to load code from the framework application. See Fig. 1 for an illustration.

If an attacker is able to install an application that pretends to provide the common framework, e.g., by convincing the user to install a seemingly benign application that internally uses the same package name as the framework, then applications based on this framework will load the attacker's code instead of the real framework. Without custom integrity checks, the applications will run the malicious code with their own permissions, and the attacker gains full access to their internal data.

The approach of loading a common framework was employed by a well-known company developing web and multimedia software. They used it for their multi-platform application framework until June 2013, when they began to bundle the framework with every Android application that uses it. In Section V-B2, we demonstrate a code injection attack against applications based on this framework.

3) Framework updates: As outlined in Section II-A, many current applications bundle various frameworks for additional functionality. Well-known examples are advertisement frameworks, which display advertisements to the user and function largely independently from the rest of the application. As mentioned previously, it is particularly difficult for framework developers to ensure that the latest version of their software is used by all applications. Therefore, some frameworks use the previously discussed techniques to load external code in an effort to implement a self-update mechanism that is independent from updates of the containing application. Fig. 2 illustrates the concept.

Our analysis shows that such update mechanisms are often implemented in a vulnerable way that allows attackers to replace the legitimate updates with malicious code. This effectively makes every application that is based on such a framework vulnerable to code injection attacks. In Section V-B1, we show an attack against an application that contains a vulnerable advertisement framework.

4) Loading add-ons: Some applications can be extended by add-ons that are installed as separate applications. Examples that we found during our analysis include an SMS application that loads design themes and a game that loads additional levels. In most cases, the additional content that applications loaded contained not only data, but also executable code.

When identifying and loading add-ons, applications have to ensure that the external code they execute is indeed part of a valid add-on. We found that such checks, if existent at all, are often implemented in an insecure way. Thus, malware can pretend to provide add-ons for legitimate applications, which those applications erroneously load and execute.

Clearly, these are legitimate reasons for benign applications to load external code. Thus we arrive at the conclusion that entirely removing the ability to load additional code at runtime would limit Android applications too much. For this reason, we develop a protection system that provides a secure way for applications to load code.

C. Policy change for Google Play

Google changed the content policy for Google Play in April 2013, adding the following sentence [16]:

> "An app downloaded from Google Play may not modify, replace or update its own APK binary code using any method other than Google Play's update mechanism."

The statement does not address all of the previously described code-loading mechanisms, but even if it did, there are several reasons why we believe that this change in policy would by no means constitute a solution to the issue: As discussed above, there are legitimate reasons for applications or application components to load and execute external code. If Google tried to prohibit the use of those techniques entirely, they would remove a valuable mechanism from application and framework developers. Furthermore, at the time of writing, the

new policy is not technically enforced. There is still a large number of applications on Google Play that load external code (see Section V). Finally, when we developed our detection tool for code-loading behavior in Android applications, we found that, in some cases, it is very challenging to determine reliably whether or not an application loads additional code (see Section VII).

D. Problems with loading external code

After having discussed the mechanisms that allow applications to load code and the reasons for benign applications to do so, we now show how the ability to load additional code can lead to severe security issues. Corresponding with the two threat scenarios defined in Section II-B, we present two types of attacks:

1) A malicious application that is able to evade detection by the Google Bouncer, so that it is made publicly accessible on Google Play.
2) Code injection attacks against benign applications that use code-loading techniques, affecting millions of users.

1) Code loading as evasion technique: Authors of malware can use code loading to evade offline analysis systems such as the Google Bouncer, as described in Section II-B.

We developed a proof-of-concept application to show the viability of this attack scenario: We demonstrate that an attacker can write a minimal application that passes the checks imposed by Google Play and downloads the actual malicious code only after it is installed on users' devices, i.e., after the store's vetting process.

Our demonstration application requests permissions to access the Internet and to write to external storage. It contains only one single activity with a download button and a log view. The download button causes the application to connect to our server and download an APK. It loads the APK and executes whatever code the file contains.

Note that our server has not offered malicious code at any point in time. It only ever answers requests from our own test devices, and the code that it serves just opens the browser on the system. Thus, our experiments did not impose any risk on users who might have installed our application from Google Play.

We submitted the application to the Google Play store in April 2013, where it was accepted within approximately 90 minutes. After that, the application was publicly accessible in Google's application store. The Google Bouncer did not detect the application's potential to download and execute malicious code. Since we logged all requests to our server, we know that the Bouncer did not even request the current payload at the time before admitting the application. The application is essentially free to download any code from our servers and execute it on the users' devices. Thus, an attacker could use this technique to circumvent the checks that Google imposes.

As of July 2013, the application has been installed 255 times by users of Google Play, although it is not advertised in any way. Even after it gained some popularity, it was not re-checked and removed from the store. Essentially, there is no way for the Bouncer to know that we do not distribute malicious code. So any malware could follow the same approach to evade detection. We believe that this is a conceptual flaw in Android's security model.

Furthermore, we evaluated the effectiveness of anti-virus software in detecting malicious code that is dynamically loaded by an Android application. Specifically, we developed a simple downloader application that, as soon as it is opened, downloads a malicious APK file from our server and asks the user to install it. The malicious APK that our server provided was a sample of the well-known Android malware family *PJApps*. We then used two meta-anti-virus scanners (VirusTotal[2] and AndroTotal[3]) to evalute the detection rate of several anti-virus applications on both the original *PJApps* sample and the downloader application we developed. When presented with the *PJApps* sample, 33 out of 46 anti-virus applications used by VirusTotal and all 7 used by AndroTotal correctly flagged the APK as malicious. However, no anti-virus was able to detect the malicious nature of our downloder application. The anti-virus software employed by VirusTotal did not perform any connection to our server to obtain the malicious sample. The scanners used by AndroTotal performed connections to our distribution server from three different IP addresses, but still were unable to identify the application as malicious.

Our proposed protection system (see Section VI) mitigates all of these threats by enforcing signature checks on *any* code – including any external code – so that attackers cannot escape analysis and detection by downloading code at runtime.

2) Code loading as source of vulnerabilities: Improper use of the code-loading techniques presented in Section III-A can introduce severe vulnerabilities in benign applications. There are many subtle aspects to consider, and our analysis of applications from Google Play shows that application developers often fail in their attempt to implement the mechanism securely (as we will describe in more detail in Section V).

The essential problem is that Android directs responsibility for checking the integrity and authenticity of external code to the application or framework developers. Those developers' main concern is the application or framework's functionality, and they are often unaware of the security implications of their applications' behavior. This leads to opportunities for attackers to replace the expected legitimate code with malicious one. Without proper checks, applications will not detect the attack and execute the malicious code.

The most common problems we found are the following:

a) Insecure downloads: Some applications download external code via HTTP. Since HTTP connections are vulnerable to man-in-the-middle attacks, it is possible for attackers to modify or replace the downloaded code. Fahl et al. showed that the use of HTTP and improper use of HTTPS are a widespread problem in Android applications in general [15].

b) Unprotected storage: We observed applications that download additional code (often in the form of APKs) and store it in the device's file system. In many cases, the storage location was improperly chosen in such a way that other

[2]https://www.virustotal.com/
[3]http://andrototal.org/

applications had write access to it (e.g., directories on the external storage of the device, often an SD card). This allows other applications on the device to tamper with the code. By modifying an APK before it is installed, for instance, attackers can gain full access to the new application's data.

c) Improper use of package names: Every application that is installed on an Android system is identified via its package name. The developer can freely specify the package name, but it must be unique on the device. Applications can use the package names of other applications to locate and load their code (see Section III-A for details). However, application developers often do not consider the possibility that a particular package name can be used by several applications, as long as they are not installed on the same device. In particular, the application that uses a given package name first, "reserves" it on that device. The package name is not displayed to the user during application installation, so that an attacker can choose a package name for her application that is normally used by a well-known application (such as a framework that many applications load code from). If a user installs such a malicious application, then any application that uses the well-known package name will load the malicious code instead. Note that the malicious application must be installed before the benign one, because the system does not allow the installation of applications with a package name that is already used on the device.

In Section V-B, we will present exploits against real-world applications using the above attack techniques.

IV. Automatic detection

In order to assess how widespread the previously described problems are among popular Android applications, we developed a tool that extracts information about the (mis)use of code loading techniques by means of static analysis.

At a high level, our detection tool receives an APK as input and looks for indications of problematic code loading. The analysis is performed directly on top of Dalvik bytecode, and does not require the source code of the application to be analyzed. The output of the tool consists of an informative report that shows whether the given application could possibly load additional code at runtime. If this is the case, the report also indicates the category of every detected code-loading attempt (as described in Section III-A).

In this section, we describe the design and implementation of our tool, while we will present the results of our analysis in Section V-A.

A. Basis for the tool

We developed a static-analysis tool for the detection of code-loading behavior.

We would like to note that parts of this work were developed for other, already published research [14]. We do not claim those parts of the tool as contributions, but in this section we will describe all the relevant details for completeness. In the following, we will clearly state which parts of the tool are novel.

The tool first uses Androguard [13] to disassemble the Dalvik byte code and to obtain information about classes, methods, basic blocks, and the individual Dalvik bytecode instructions. Based on this data, it transforms the code into *static single assignment (SSA)* form [10]. It then performs *Class Hierarchy Analysis (CHA)* and it builds the *control flow graph (CFG)* for each individual method.

B. Construction of the sCFG

On top of this, the tool constructs the *super control flow graph (sCFG)*, which represents possible method invocations between the different methods. In particular, an edge in the sCFG means that a given *invoke* instruction could possibly direct control flow to the given method's entry point.

Even though the construction of the sCFG is a well-studied task in the program analysis literature, it is not straightforward to build a precise call graph. The key difficulty is that Dalvik bytecode (like most object oriented languages, such as C++ and C#) heavily relies on the dynamic dispatch mechanism to determine which method a given *invoke* instruction will jump to. For the purpose of this work, we chose to apply a class-hierarchy-based algorithm that scales well while being reasonably precise at the same time. From a high-level point of view, the tool determines the possible targets for each *invoke* bytecode instruction by combining the information about the targets' types from the *invoke* instructions and the results provided by the class-hierarchy analysis.

More specifically, we implemented the algorithm as follows. Given an invocation i of a method m of class c, we first use the class hierarchy to find all non-abstract subclasses of c. If c is an interface, then we locate all classes that implement the interface, and their subclasses. We place c and all classes found so far in the set X. Then we check for each class x in X whether the class implements a method with the same signature as m (i.e., the class overrides m). If this is the case, we accept the candidate as a possible target and connect it to the invocation i in the sCFG. Otherwise, we traverse the class hierarchy upward starting from class x, to check whether a compatible method is implemented in classes extended by x: the first method we encounter will be considered as the target of the invocation i.

Note that this algorithm produces an over-approximation of the sCFG: In other words, we introduce edges between *invoke* instructions and method entry points that might never be used at runtime. However, as we will discuss below, this does not pose a problem for the heuristics we developed for our analysis.

C. Backward slicing

The type of analysis we aim to perform requires the capability of program slicing. Based on the control flow information, the tool is able to compute slices of the analyzed application. We implemented a backward slicing algorithm that works on top of the sCFG, based on work by Weiser [35].

Given an instruction i and a register r used by the instruction, a slice $s(i, r)$ is a set of instructions that can possibly influence the value that register r contains when executing the instruction i. We compute the slice by starting at instruction i and walking back in the method's CFG, tracing the flow of data toward r in the SSA representation of the code. Whenever the

slicing algorithm reaches the beginning of a method, it uses the sCFG to locate all possible callers and recursively continues the analysis. Our implementation also keeps track of class and instance variables, so whenever such variables are used, the analysis recursively continues at all points in the code that assign values to the respective variables.

D. Heuristics to detect code loading

The following heuristics were implemented on top of the existing tool specifically for the work presented in this paper.

The goal of our analysis is to find uses of the loading techniques detailed in Section III-A. Therefore, the detection tool looks for invocations of methods that are associated with the respective techniques (e.g., *createPackageContext* to load code via a package context). If it detects a suspicious invocation, it uses the slicing mechanism described above to further analyze the call. In the previous example, a mere call to *createPackageContext* is not enough to conclude that the application loads external code. Instead, we have to make sure that the flags *CONTEXT_INCLUDE_CODE* and *CONTEXT_IGNORE_SECURITY* are passed as parameters (see Section III-A). We do so by computing a slice from the call instruction and the register that contains the *flags* parameter.

We implement several heuristics to detect issues with code-loading:

1) General detection: In order to identify candidate applications for further analysis, we detect the use of all code-loading techniques presented in Section III-A by identifying the associated method invocations. If a technique requires a method invocation with specific parameters, we check their presence by computing slices for the respective argument registers.

2) Storage location: For APK installations, class loaders and native code, we analyze the storage location of the code to be loaded. If the code is stored in a world-writable location such as the device's external storage, we consider the load operation to be vulnerable, because other applications can maliciously modify the code.

3) Code provenance: For the same set of techniques, we search for indications that the code is downloaded via HTTP. We flag such cases as problematic, because HTTP is vulnerable against man-in-the-middle attacks.

4) Package names: We consider code-loading based on package names a security risk at all times, because a given package name is not guaranteed to belong to the desired application (see Section III-D2). In particular, creating package contexts and using them to load code is a vulnerable operation, since the target application is always identified by its package name.

With the help of our tool, we identified a large number of vulnerable applications on Google Play, among them even very popular applications with millions of users (see Section V-A).

V. Large-scale analysis of benign applications

In the previous section, we described a tool for automatic detection of code-loading behavior. We applied the tool to various sets of real-world applications in order to assess the prevalence of code-loading techniques and the associated security issues. We found that the techniques are very popular and lead to vulnerabilities in a number of applications. In this section, we first present the results of our study and afterward detail two severe vulnerabilities that we found during the analysis.

TABLE I. Use of different code-loading techniques in 1,632 popular applications from Google Play.

Category	Applications in the category (relative to the whole set)	Flagged vulnerable (relative to the whole set)
Class loaders	83 (5.01%)	31 (1.90%)
Package context	13 (0.80%)	13 (0.80%)
Native code	70 (4.29%)	0
APK installation	155 (9.50%)	117 (7.17%)
Runtime.exec	379 (23.22%)	n/a
Total	530 (32.48%)	151 (9.25%)

A. Use of code-loading techniques

We applied the detection tool to the following sets of applications from Google Play:

1) A set of 1,632 applications chosen randomly in May 2012 from among those applications on Google Play with more than one million installations.
2) The 50 most popular free applications from Google Play in November 2012.
3) The 50 most popular free applications from Google Play in August 2013.

We ran our detection tool on the applications in those three test sets with a timeout value of one hour. In 10% of the cases, the runtime exceeded the timeout value. In the remaining cases, the mean runtime was 74.9 seconds per application with a standard deviation of 55.3 seconds. Taking all executions into account, including those that timed out, the median runtime was 69.8 seconds. The mean size of the analyzed code files was 3,303 KB, the standard deviation 2,152 KB.

Tables I, II and III show the use of code-loading techniques by applications and the number of vulnerabilities detected by our tool in the different test sets, respectively. Note that a single application can use multiple techniques. Also, note that code-loading using *Runtime.exec* does not usually lead to vulnerabilities, because it is commonly only used to execute system binaries. The surprisingly high numbers of applications containing code for this technique is partially due to multi-platform frameworks: *Runtime.exec*, as opposed to the other code-loading mechanisms presented in this paper, is not specific to Android. We found Java frameworks in the analyzed applications that exclusively use *Runtime.exec* when executed on non-Android systems.

Our results show that loading code at runtime is indeed a widespread phenomenon among benign applications. We found a surprisingly high number of potentially vulnerable applications. The analysis of the large test set indicates that 9.25% of the applications on Google Play are vulnerable to code-injection attacks (see Table I). The situation in the top 50 free applications at the time of writing is even more alarming: 16% of the applications contain vulnerable code (see Table III).

TABLE II. CODE-LOADING IN THE TOP 50 FREE APPLICATIONS AS OF NOVEMBER 2012. VULNERABILITIES MANUALLY CONFIRMED.

Category	Applications in the category (relative to the whole set)	Flagged vulnerable (relative to the whole set)
Class loaders	8 (16%)	1 (2%)
Package context	1 (2%)	1 (2%)
Native code	4 (8%)	0
APK installation	4 (8%)	2 (4%)
Runtime.exec	12 (24%)	n/a
Total	20 (40%)	3 (6%)

TABLE III. CODE-LOADING IN THE TOP 50 FREE APPLICATIONS AS OF AUGUST 2013. VULNERABILITIES MANUALLY CONFIRMED.

Category	Applications in the category (relative to the whole set)	Flagged vulnerable (relative to the whole set)
Class loaders	17 (34%)	2 (4%)
Package context	0	0
Native code	10 (20%)	0
APK installation	11 (22%)	7 (14%)
Runtime.exec	24 (48%)	n/a
Total	31 (62%)	8 (16%)

When comparing the top 50 free applications from May 2012 (see Table II) with the top 50 free applications from August 2013 (see Table III), we find a disquieting tendency: While only 3 out of the top 50 applications were vulnerable in November 2012, the share has since increased to 8 out of 50 in August 2013. We confirmed all vulnerabilities reported by our tool for these two test sets by manual analysis. Further manual analysis showed that the vulnerabilities are mostly due to frameworks. For example, several of the top 50 applications use frameworks that allow testers to install beta versions of the containing applications. We found that the two frameworks used by the top 50 applications from August 2013 both download beta versions in the form of APKs to the device's external storage, from where the APKs are installed. Since external storage is writable by any application with the appropriate permission in Android, any application on the device could replace the benign APK with malicious code.

The previous example illustrates an important property of our results: Applications that our tool marks as vulnerable contain vulnerable code, but the tool does not guarantee that the code is executed on all devices and in the default configuration of the application. In the previous example, a user must participate in the developers' beta-testing programs for the applications to expose vulnerable behavior. Nevertheless, we found sufficient evidence that many applications are vulnerable even in the default configuration and on any device, as the two sample exploits in Section V-B will show.

We manually analyzed some of the applications in the current top 50 that use code-loading techniques but were not flagged as vulnerable by our tool (see Table III). The reason is that the detection tool is rather conservative in classifying applications as vulnerable, so that the reported numbers of vulnerable applications can be interpreted as a lower bound. Three of the manually analyzed applications make use of a framework developed by the provider of a large application store and manufacturer of Android devices. We found that the applications contain stub code to load the framework, which is installed as a separate application. The code identifies the application by its package name, which most likely makes the applications vulnerable to an attack similar to the one presented in Section V-B2, thus increasing the real number of vulnerabilities above the conservative estimate of our tool.

B. Exploits against vulnerable applications

We now demonstrate how improper use of loading techniques can make benign applications vulnerable to exploitation by presenting attacks against vulnerable real-world applications (cf. the second attack scenario described in Section II-B). We found the vulnerabilities using our automatic detection tool during the study presented above.

1) Self-update of an advertisement framework: This is our first example of a code injection attack against a benign application. The application in question – a game with a number of installations between five and ten million according to Google Play – includes an advertisement framework. This framework has the capability to update itself via the Internet. Whenever the application is started, it checks with the servers of the framework developer whether a new version is available. If this is the case, it downloads the new version and loads it via *DexClassLoader* on subsequent application starts. The connection between the application and the web server uses HTTP instead of HTTPS. Since HTTP does not protect the integrity of the transferred data nor authenticate its source, it is possible to provide the application with a bogus update.

An attacker has several ways to mount an attack. She can, for instance, tamper with the DNS resolution of the victim's device (e.g., in an unencrypted WLAN) to redirect the connection to a server she controls, or execute a man-in-the-middle attack against the HTTP connection.

In our example exploit, we take over the connection and serve the application a custom file instead of the expected update. The application does not verify that the downloaded code originates from a trustworthy source. It only receives an MD5 hash from the update server along with the actual update file in order to detect transmission errors. We provide the application with a custom APK and the matching MD5 hash. The victim application does not detect the code injection and loads our APK the next time it starts. The only requirement for the malicious APK is that it contains a class with a specific name defining a certain method – both can be easily extracted from the application's Dalvik bytecode. As our APK meets the requirements, the application now runs the code that we injected, and we are free to use its permissions to our advantage and access its internal data.

By design, our exploit works against any application that uses the framework, and applications are vulnerable in the default configuration on every device with Internet access. According to AppBrain, this framework is included in 0.78% of all applications on Google Play and 3.21% of the top 500 applications [3]. It is likely that all those applications are vulnerable.

We informed the company behind the advertisement framework about the vulnerability in July 2013. They responded within a few hours and acknowledged the severe security issue. Moreover, they confirmed that their framework is used in more than 10,000 applications, putting millions of users

at risk. Within a few days, they published a new version of their framework without the vulnerable update component. We refrain from naming the framework or the company, because it will take time for the patch to be included in the affected applications and to be pushed to the users' devices.

2) Bootstrapping mechanism of a shared framework: The lack of verification of loaded code is not only a problem when downloading code from the Internet. In this next exploit, we attack applications that load code locally and fail to verify its integrity. The target application, downloaded from Google Play, is based on a framework by a well-known company in web and multimedia technologies.

The framework allows application developers to create applications for several different platforms. The developer essentially designs a Flash file, which the device-specific framework runtime can execute on a variety of systems. The Android version of the runtime is installed as a standalone application that any application based on the framework has to load at start-up. The code that loads the framework is generated automatically for the application developer. It uses *createPackageContext* (see Section III-A) with a hard-coded package name to load the framework runtime into the current application. However, the loading code does not verify the integrity of the loaded application, so that any package with the right name is accepted. This allows us to mount a code injection attack as described below.

We install a custom application with the required package name and the expected class on a test device. When we launch an application based on the framework, it loads our application (which it mistakes for the framework runtime) and executes our code. Note that in order for the exploit to work, the attacker has to be able to install an application on the device *before* the user attempts to install the real framework runtime. However, this application does not need any permissions at all – the code "inherits" all permissions from the exploited application, so that it is relatively easy for an attacker to lure users into installing her malicious code. Again, verification of the integrity and authenticity of loaded code could have avoided this attack.

In May 2013, the company that develops the framework announced that future versions will package the runtime directly with each application rather than load it from a shared package, so that this exploit will not work with applications based on later versions of the framework. However, our exploit works against any application that is built on top of the framework up to the version that was published in June 2013. It can attack those applications in their default configuration. According to AppBrain, the framework is used in 2.13% of all applications on Google Play and 2.81% of the top 500 applications [3]. It is likely that a large portion of the applications that use the framework has not been updated to the latest version and is thus vulnerable.

We notified the company in July 2013 and provided details on the vulnerability we discovered. As of August 2013, we have not received any response.

VI. Design of the protection system

In the previous sections, we described the problematic behavior of loading code dynamically at runtime, the risks arising from it, and a tool that analyzes existing applications for such behavior, detecting a large number of vulnerable applications. We now focus on a possible solution. We start with a high-level overview of the protection system before examining details of the implementation.

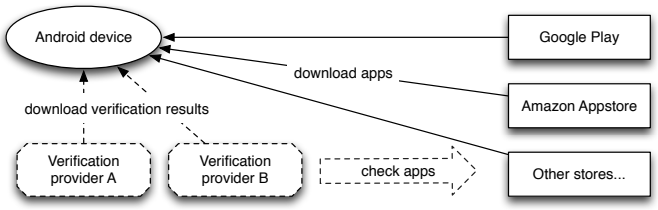

Fig. 3. Overview of the infrastructure that we propose. The dashed parts are our addition to the current system.

A. High-level overview

In general, it is a bad decision to delegate responsibility for system security to application developers if protection can be implemented by the operating system itself. The protection system that we propose adds a missing verification mechanism to Android, making it mandatory for all applications. Checking the integrity of the code before executing it can mitigate all attacks resulting from the ability to load external code. For instance, the advertisement framework that we attacked (see Section V-B1) could have used SSL to make sure that the update originates from a trustworthy source and was not tampered with by an attacker. Similarly, applications based on the exploited multi-platform framework (see Section V-B2) could check the integrity of the common framework application before executing its code, e.g., by means of signature verification. However, Android leaves the burden of implementing such checks to the application or framework developer.

At its core, our modification enforces that an authority that the user trusts approves every piece of code that an application loads. We envision different *application verifiers* that analyze applications, each according to a custom set of criteria and with different algorithms. If an application verifier deems an application benign, it issues the equivalent of a signature for the application and makes it accessible to the public. Application verifiers are independent from application stores – the store offers applications for download, while the verifier provides approvals for applications. Although store providers are free to act as applications verifiers, this approach has the advantage that there is no need for store providers to change anything. We only add the verifiers to the ecosystem. Fig. 3 illustrates the changes that we envision.

Users can choose which verifiers to trust. We see several advantages in this design decision:

Different users may have different priorities, so application verifiers can focus on different sets of criteria when analyzing applications. For example, enterprises will employ different evaluation criteria for applications on their employees' devices than home users do for their private phones or tablets. Users get the freedom to choose verifiers according to their priorities.

Furthermore, users do not depend on a single verifier, as is the case in the iOS ecosystem (where devices can typically only run software approved by Apple).

Finally, by decoupling application verification from application stores, we achieve the same level of security for all applications regardless of the store distributing them. At the moment, users of alternative stores to the one preferred by the device manufacturer (usually Google Play) are inherently at risk, because Android only distinguishes two modes: "allow only applications from the preferred store" and "allow *any* application" (the sideloading setting, as described in Section II-A). This means that users of alternative stores have no choice but to open their devices to *all* APKs, regardless of their origin. On the other hand, our solution focuses on individual binaries rather than stores, so that it protects users of all available stores equally.

Note that the Java Virtual Machine (JVM), the standard Java environment, uses its Security Manager to deal with the tasks of authenticating and verifying untrusted code. Android, in contrast, accomplishes most of the Security Manager's tasks (except code authentication and verification) by means of its permission system already. The JVM leverages signatures for code verification, which is similar in spirit to our approach. The difference is that we trust code after a verification check, whereas the JVM trusts code if it originates from a developer that presents a certificate.

We implemented our protection system as a modification to Android 4.3, the most recent version of Android at the time of writing.

B. Detecting attempts to load code

Before we describe our implementation, we make the following crucial observation: In a Java application, it is not directly possible to execute externally loaded Java code. In order to make it executable, applications have to ask the Java runtime environment to load it (e.g., using a class loader). This provides us with the unique opportunity to impose checks on the code that is to be loaded, whereas in other scenarios, such as raw machine code, we could easily miss the fact that an application is loading code. Therefore, our system is implemented as a modification of the Dalvik VM, the component of Android that executes Java code. This means that our protection system does not require changes to the operating system as a whole and can be easily applied.

Whenever an application asks the Dalvik VM to load code, we check the integrity of the code from within Dalvik (details on the nature of our checks follow shortly). At first glance, it seems possible to implement the integrity checks at a higher level in the system, e.g., as a modification of the Java framework that Dalvik provides to applications. However, the reason for us to choose the lower level is *reflection*, a Java mechanism for introspection: Reflection allows Android applications to call any Java method irrespective of its protection attributes. This would allow applications to escape our protection system if we implemented it in Java. For example, imagine a Java method that checks code integrity and subsequently calls the (private) native method provided by Dalvik to load the code. Then a malicious application could just call the native method directly using reflection, thus circumventing the check. As a result, we have to implement any checks in Dalvik's native code.

Similarly, one might argue that rooting the protection scheme deeper in the Android system would constitute a more comprehensive way to address the issue. However, we refrained from doing so mainly for two reasons:

1) A separate branch of the Linux kernel is maintained for each device, so that modifications to the kernel are more difficult to adopt on all devices than modifications to higher levels of the Android system.
2) Our approach is based on a key property of Java, so there is no need to work at a lower level than the Java virtual machine. Doing so would only introduce unnecessary complexity.

Since our system relies on the necessity to make code-loading operations explicit in Java, native code imposes a challenge. Native machine-code instructions, as opposed to Java byte code, provide a variety of ways to execute additional code or modify existing one. We cannot prohibit the use of native code, because applications use it legitimately, e.g., to accelerate computationally expensive tasks [1]. Zhou et al. found in 2012 that 4.52% of applications from different markets use native code [39]. However, applications always start by executing Java code in the Dalvik VM. By hooking the interface in Dalvik that applications have to use in order to load native libraries, we can detect when an application tries to execute native code. We enforce an integrity check on any native code that the application tries to load.

It is the responsibility of the verification service that approves an application to make sure that the application's native-code part does not load any additional code. Note that this decision is much easier for the verification service than it would be for our protection system: Application developers can provide verification services with additional material to prove that their native code does not load any additional external code (e.g., source code under appropriate agreements). Our protection system, on the other hand, can only rely on the bare assembly code at runtime, which makes it much harder to assess the code's properties. Even without loading any external code, techniques such as return-oriented programming allow applications running native code to execute arbitrary programs, leveraging code that is already present in memory [8], [30]. While the obligation to provide proof for properties of native code is an obstacle for developers, we believe that the resulting large improvement of system security justifies it.

C. Signature scheme based on whitelists

The integrity check that we enforce on loaded code is a lookup in a signed whitelist. Note that keeping whitelists signed by verification providers is an equivalent approach to attaching signatures to applications, but it has additional advantages: A signature in the common formats [7] is just a hash of the file in question that is signed by the issuer's private key. Similarly, a (signed) whitelist provided by a verification provider is essentially a signature for multiple files. We do not attach the verifier's signature directly to the APK in order to keep store and verification service separate. This has the advantage that nothing changes in the way stores work, and existing stores can continue to operate as before. Conceptually, the verification systems are simply an addition to the ecosystem. They provide signatures, but do not have to offer applications

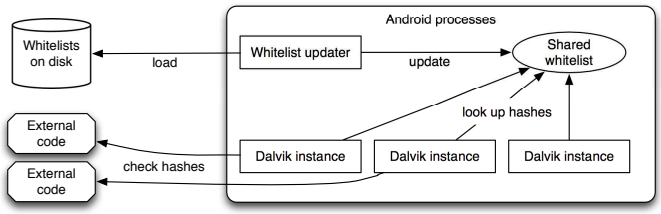

Fig. 4. The architecture of our protection system. Instances of the Dalvik VM share a system-wide whitelist, which is backed by the device's file system.

for download. So, the user's device downloads applications from stores – as before – and validates the executables with signatures that it downloads from verification services. Due to the relatively small size of the hashes, we combine all hashes of the same verification service into one file that can be cached on the device. The file contains a signed list of hashes – the whitelist.

Upon receiving a request to load code, our modified version of the Dalvik VM computes a hash of the code that is to be loaded and checks it against a system-wide whitelist. We use the SHA-256 implementation of the OpenSSL library that is part of Android for the hash computation. This computation has to be done only once per code file, because Dalvik keeps a cached version of the file in a private location after the first load operation. If the computed hash is not contained in the whitelist, the VM cancels the load procedure immediately. The whitelist is shared among all instances of the VM on the device and managed via files in a system-protected directory. Thus, only system applications (i.e., applications that are signed with the same key as the system itself) can update the whitelist files, which are immediately reloaded into shared memory after a modification. Fig. 4 illustrates the design.

D. Whitelist management

We envision that each verification service that the user configures on the device provides a single whitelist file that is updated regularly. A system application periodically downloads the latest version of the whitelists and stores them in the dedicated system directory from which our modified Dalvik VM loads them into shared memory. Note that updating whitelists in face of newly emerging malware is not as critical as is the case for blacklists: Malware that is detected after the creation date of the whitelist still cannot execute on the protected system, because its hash is not part of the whitelist. Forging a malicious application with the same hash as a whitelisted benign one requires breaking the hash function (SHA-256 in our case), which is generally assumed to be a hard problem.

In our implementation, we just combine the whitelists from different providers by a set union (a Boolean OR of the approvals). It is sufficient for *one* of the trusted verification services to approve code in order for the system to execute it. It is conceivable to use more complicated expressions in future versions (e.g., "accept code if it is approved by verification service A or by verification services B and C"). Such behavior is easy to add to the current system.

Additional considerations are necessary for *DexClassLoader*. As described previously, *DexClassLoader* allows the user to specify the cache directory on which the system relies for subsequent load operations (see Section III-A). Thus, an attacker can override the files in the cache directory after our verification succeeded. The mitigation of such an attack can be implemented as follows: If the verification of the original file succeeds during the first load operation, the system can "extend" the trust to the corresponding optimized file in the cache directory by adding its hash to a temporary whitelist. Subsequently, any modification of the cached file can be detected.

E. Permission for Runtime.exec

The API *Runtime.exec* (previously mentioned in Section III-A) allows applications to execute arbitrary binaries on the system. This way, applications can use system binaries that are very generic in nature (such as a shell). It is difficult for verification services to classify such generic binaries into categories like "benign" and "malicious" – they can be used for legitimate purposes as well as for malicious ones.

A classic example is the system shell */system/bin/sh*: It is present on all Android systems and not malicious in itself, but it has inherent potential to be misused by malicious applications. Due to its ability to carry out almost arbitrary tasks on the system and to launch other binaries, malware can use it to conduct unwanted activities. Thus, a verification service cannot approve */system/bin/sh* per se.

Instead, we require applications that use *Runtime.exec* to ask for explicit permission using the Android permission system. The reason is that we expect developers to be able to prove the non-malicious character of their JARs, APKs or native code libraries in order to have them approved by a verification service, whereas *Runtime.exec* enables applications to use binaries that are too generic for such a classification.

Any application that uses *Runtime.exec* has to declare this intention in its manifest, so that the user is made aware of the potentially dangerous behavior at installation time. We implement the permission by modifying the Android framework. Like some of the already existing permissions (e.g., the permission to access the Internet), our permission is enforced by Linux groups: An application that has been granted permission for *Runtime.exec* executes in a process that contains a particular Linux group in its set of complementary groups. Our implementation of *Runtime.exec* verifies the group membership before taking any action. By doing this, we can make sure that applications have to explicitly declare to the user that they will use the dangerous API.

From a technical point of view, executing an arbitrary binary is no different from loading native code through JNI. However, native libraries are usually not as generic as some system binaries, because they are designed to support a single application, whereas system tools, such as the shell, are designed to accomplish a wide variety of tasks. Thus, we expect it to be much easier for developers to prove the non-maliciousness of a native library to application verification services. Therefore, we believe that a distinction between JNI native libraries and binaries executed through *Runtime.exec* is appropriate.

Note that the introduction of the new permission requires developers to change the manifest file of their applications if

they use *Runtime.exec*. However, we believe that this is the only way to mitigate the risk that uncontrolled use of this particular API imposes on the overall system's security.

F. Evaluation

After having presented our protection system's design, we now assess both its effectiveness and efficiency.

1) Effectiveness: For an assessment of the protection system's effectiveness, we created a simple application that exercises all the code loading techniques described in Section III-A. We verified that, in an unmodified Android system, they all led to the execution of external code. We tried to use the same techniques in our protected version of Android, without whitelisting the code that the application tries to load and without giving the application permission for *Runtime.exec*.

The protection system successfully blocked all attempts to load external code. The techniques using class loaders, native code and package contexts, respectively, were detected and blocked immediately. The attempt to execute code using *Runtime.exec* was prohibited by the framework due to the missing permission. APKs can be installed, but the system does not allow launching the installed applications.

Note that it would be possible to intercept APKs during installation already, for a more user-friendly experience. This would require hooking the Package Manager Service. However, we refrained from doing so in order to modify the operating system and its services as little as possible while still achieving complete protection against all presented code-loading techniques.

In a further test, we tried to execute the two attacks presented in Section V-B against a device running our protection system. As expected, the protection system blocked both exploits because the injected code files were not trusted.

2) Efficiency: For the analysis of the system's efficiency, we examine performance and memory overhead.

In order to assess the performance overhead, we measured the time that the modified Dalvik VM needs to check code during the first load operation. The check consists of a SHA-256 computation over the file that is to be loaded and a lookup in the in-memory whitelist. Running on a 2.8 GHz Intel Core i7 CPU, the release build of our system needs 0.25 milliseconds on average in the Android emulator to look up a hash in a whitelist with 1,000,000 entries, and by varying the number of entries we find that the lookup time increases logarithmically. The SHA-256 computation uses the OpenSSL implementation of the hash function and takes 123 milliseconds on average for an APK file of 20 MB in size.

The second important aspect of the protection system's efficiency is memory consumption. The only factor that influences its memory consumption is the number of whitelist entries that are used on the device. Since we use 32-byte long SHA-256 hashes, the space required in memory can be specified as $32 \cdot w + c$ bytes, where w is the number of entries in the whitelist and c a constant smaller than 100. For example, a whitelist containing the approximately 1,000,000 applications from Google Play would consume roughly 30.5 MB of memory. Our current implementation reserves a memory block of fixed size for simplicity, but it is easy to replace this behavior with a strategy that adjusts the amount of reserved memory based on the number of entries in the system whitelist.

We believe that the whitelisting mechanism will not lead to scalability issues. Should the number of applications grow so fast that the size of the whitelist turns problematic, the following modification will fix the issue: In our current system, Android devices download complete whitelists from verification services, containing the hashes of all applications that a respective service approves. A future version of the system could just download hashes for installed applications during the installation process. So, whenever the user would choose to install an application, the system would ask all trusted verification services for corresponding signatures. This would eliminate any problems with storage space at the cost of requiring the user to be online while installing applications. While this would constitute a limitation, we believe that it would not normally affect users. Applications are usually installed from online stores, so that connectivity is provided during installation.

Note that the specified amount of memory is the total amount allocated on a device. Since all Dalvik instances share a global whitelist, space needs to be reserved only once for all of them.

VII. Limitations

While we believe that our system addresses the major issues with runtime code loading, it has certain limitations. In this section, we discuss such limitations and suggest future improvements to address them.

A. Automatic detection tool

Our detection tool for dynamic code-loading behavior uses static analysis techniques. Therefore, it does not have any information per se whether a given code fragment is executed in the default configuration of the application. This issue could be addressed by more sophisticated data-flow analysis, which would yield the conditions that are necessary for the application to execute the vulnerable functionality.

Furthermore, the detection tool does not detect if an application secures the code-loading operation by implementing custom integrity checks. While we did not find such custom checks in the applications reported as vulnerable that we analyzed manually, we might have to add detection capabilities for them in the future, as developers become aware of the security risk and start to implement custom protection. Good starting points for the detection of integrity checks in applications are hash computations, because common mechanisms to verify the integrity of data usually involve hash functions.

Finally, there are several Android-specific challenges that need to be addressed in order to perform precise static analysis of Android applications. For example, one would need to model the life-cycle of each application's *Activities*, as well as the implicit method calls that the Android framework performs as a reaction to user input (such as the click on a GUI button). One approach would be to manually add support for these Android-specific features, but as with all manual modeling, the results are likely to be incomplete.

B. Protection system

A central question is the practicality of our proposed protection mechanism. The system requires every piece of code that an application likes to load to be submitted to a verification service. However, we consider this a feature because it prevents Android devices from loading unknown and potentially dangerous code. Google already conducts an extensive analysis on every application submitted to Google Play, so it seems completely feasible for them to also run checks on additional code that applications load dynamically. Furthermore, it is entirely possible for existing stores, such as Google Play, to act as verification providers. Thus, no new entities are necessary in the ecosystem. Our system simply provides the option to add verification providers independently from application stores in order to grant the users additional freedom.

Note that our protection system requires changes in the Android source code, meaning that only reinstalling or updating the operating system can deploy it. We acknowledge that this constitutes an obstacle for fast and widespread adoption. Nevertheless, we believe that the severe security threat imposed by dynamic code-loading techniques (as documented in this paper) makes changes at the operating-system level in Android unavoidable in order to establish a reliable security model for the handling of external code.

Another concern is the need to prove the benign character of native code to verification providers. A reliable proof technique likely poses considerable overhead on application developers. The question how to alleviate this burden is subject to ongoing research. One possible approach could be to encapsulate native code in a sandbox environment similar to Google Native Client [29], [36].

In theory, attackers can write an interpreter for a scripting language in Java and download and execute scripts once the application is installed on the victim's device. Such an application would not have to load *code* in order to behave maliciously – the loaded script would just be data that the application can interpret. Percoco and Schulte used a similar approach in 2012 to circumvent detection by the Google Bouncer [27]. Their approach is based on WebViews – Luo et al. examined the security of WebViews in detail [23]. However, note that the application's Java code has to contain all the functionality that the scripting language offers to the attacker. The ability to read the user's contacts, for instance, must be present in the Java code if the attacker wants to use it from the scripting language. Thus, verification services might be able to recognize that the application provides a lot of seemingly unused functionality.

A more general concern is that an attacker could use reflection for a similar purpose. In this scenario, an application would just receive class and method names as strings and invoke the corresponding methods via reflection. For such an application, the detection approach outlined above would not work, because it would not contain any suspicious API calls. Furthermore, it is difficult to detect in an automated analysis that an application implements such a "reflection VM." A possible remediation is for verification services to reject all applications that make extensive use of reflection.

VIII. Related work

A. System protection

Google introduced a security mechanism in Android 4.2 that contacts the Google servers whenever an application is installed via sideloading [2]. The system computes a SHA-256 hash of the APK and sends it to Google along with other information about the application, asking if it is known to be malicious. If so, the system warns the user instead of installing the application. Google effectively implements a blacklist approach with this protection mechanism. Such a system has the considerable disadvantage that it is restricted to known malware. Note that the protection scheme is only active if Google Play is installed on the device and a network connection is available. If the device is not currently connected, the system silently allows the user to install the application. Furthermore, the protection mechanism is only invoked during application installations, so most of the techniques described in Section III-A can be used to circumvent it.

Apple uses a technique called *Mandatory Code Signing* in iOS [11]. It enforces at a memory-page level that all executed code is signed. Developers sign their applications with certificates issued by Apple. Additionally, from the moment the system is powered on, it ensures that only signed code is executed, including verification of the boot loader and the operating system itself (the *Secure Boot Chain*) [4]. The chain starts from a key that is hard-coded into read-only memory on the chip of any device running iOS. While this approach provides a high level of security against code injection and execution of unapproved code, it requires modifications deep within the operating system as well as changes to the hardware. We feel that Android cannot adopt a similar mechanism in the near future due to the extensive changes to hardware and software that would be necessary. Furthermore, Apple's approach ties the user to a single application verification entity: If Apple does not approve an application, then there is no way to run it on an iOS device (as long as the device is not "jailbroken"). Our approach addresses this issue by giving users the possibility to choose which verification services to trust.

Smalley and Craig ported the well-known security kernel extension SELinux [28], [33] to Android [32]. SELinux is able to enforce policies on the behavior of running applications, thus limiting the interaction of a surveyed process with its environment to those activities that it legitimately requires. However, without tailoring policies specifically to the application in question, only general rules can be established. Thus, SELinux on Android does not restrain an application that loads and executes additional code but only conducts activities within the limits of its permissions. Alternatively, application developers would have to write policies for their applications. The risk in such a scenario is that developers write policies that are not strict enough to avoid malicious activity. Also, the need to define a policy in SELinux's extensive policy language [31] would slow down application development and would thus be difficult to enforce.

Another widely researched technique to protect Android systems is Dalvik bytecode rewriting [20]. The basic idea is to detect the portions of an application that call security-sensitive APIs and to redirect the calls to a monitor service

that implements fine-grained access control. This modification of the bytecode can take place offline if static analysis is used. However, external code that an application might load at runtime poses a problem [12]. If rewriting takes place before the application is installed, then code loaded at runtime evades the rewriting process and is thus not restricted by the added protection mechanisms. To counter this attack, bytecode rewriters would have to run on the Android device and constantly watch for attempts to access sensitive APIs that have not been rewritten. This is computationally expensive and not desirable on mobile devices, where system resources are limited and battery life is a crucial factor.

B. Vulnerability analysis

In a study on the use of SSL in Android applications, Fahl et al. show that a large portion of applications does not make appropriate use of the security benefits of SSL [15]. In fact, they demonstrate that many popular applications from Google Play are vulnerable to various attacks, because the developers did not implement security mechanisms correctly. They conclude that one part of the problem is the lack of easy-to-use security features for application developers. Their results emphasize the necessity of security mechanisms that the operating system enforces for all applications, independent from the individual application developers, as implemented in our approach.

In 2010, Jon Oberheide demonstrated how to download arbitrary additional code in Android applications at runtime [26]. He used the technique to distribute root exploits to devices running his application.

Bellissimo et al. demonstrated in 2006 that many update mechanisms of applications and operating systems are insecure [6]. Their attacks are similar to those that we apply against Android applications, but targeted at desktop software.

We are not the first to use static analysis in order to detect vulnerabilities in Android applications. Au et al. use it to find a mapping between Android APIs and required permissions for different versions of Android [5]. Lu et al. employ static analysis techniques to detect applications that expose components to other applications in an insecure way, leading to the risk of what they call *component hijacking* [22]. Zhou and Jiang find similar vulnerabilities – they detect applications that expose access to content providers in a way that allows other applications to read or modify protected content [38]. Chin et al. analyze the general communication interface that an application provides to other applications in order to detect possibilities of information leakage [9]. Grace et al. detect *capability leaks* in preinstalled Android applications, i.e., permissions that are requested by system applications, but not properly protected against unauthorized use by third-party applications [17].

The previously presented approaches focus on detecting specific vulnerabilities in applications. They could be used by the verification services that we propose in order to detect vulnerabilities in benign applications.

In 2012, Grace et al. used static analysis to detect some of the code-loading techniques discussed in this paper [18]. Specifically, they searched for uses of native code and *DexClassLoader* in Android applications in an effort to identify malicious applications in stores. However, their work does not cover the other categories of code-loading techniques discussed in Section III-A, nor do the authors propose a protection scheme to systematically prevent the risks associated with dynamic code loading. Similarly, another publication by Grace et al. leverages static analysis to find uses of *DexClassLoader* during the analysis of advertisement frameworks [19] but also falls short of detecting the other code-loading techniques or offering a protection mechanism in solution.

IX. Conclusion

Our analysis shows that the ability of Android applications to load code at runtime causes significant security issues. We were able to demonstrate that a surprisingly large portion of existing applications is vulnerable to code injection due to improper use of different loading techniques. This is made worse by the fact that the vulnerabilities are often found in frameworks that are used by large numbers of applications. Additionally, we showed that attackers could use dynamic code-loading to avoid detection by offline application analysis engines, in particular the Google Bouncer. In order to automatically detect such vulnerable or malicious functionalities, we implemented a static analysis tool and we showed its effectiveness in detecting interesting samples.

Furthermore, we presented a modification to the Android system that prevents exploits resulting from vulnerable loading techniques by ensuring that all loaded code is approved by an application verification service. Based on this mechanism, we proposed a general architecture of different verification services that users can choose from. We showed that our protection system is able to prevent all attacks presented in this paper. It is our hope that the proposed modification will find its way into a future release of Android in order to be distributed to as many devices as possible.

Acknowledgements

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0101. 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, and by Secure Business Austria.

References

[1] "Android NDK." [Online]. Available: http://developer.android.com/tools/sdk/ndk/index.html

[2] "Security Enhancements in Android 4.2," accessed July 2013. [Online]. Available: https://source.android.com/devices/tech/security/enhancements.html

[3] "Android library statistics," AppBrain, URL shortened to protect users of the vulnerable framework. [Online]. Available: http://www.appbrain.com/stats/libraries

[4] *iOS Security*, Apple, October 2012. [Online]. Available: http://images.apple.com/iphone/business/docs/iOS_Security_Oct12.pdf

[5] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 217–228.

[6] A. Bellissimo, J. Burgess, and K. Fu, "Secure software updates: disappointments and new challenges," *USENIX Hot Topics in Security*, 2006.

[7] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format," RFC 4880. [Online]. Available: http://tools.ietf.org/html/rfc4880#section-5.2

[8] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 2010, pp. 559–572.

[9] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services*. ACM, 2011, pp. 239–252.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[11] D. A. Dai Zovi, "Apple iOS 4 security evaluation," *Black Hat USA*, 2011.

[12] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-arm-droid: A rewriting framework for in-app reference monitors for android applications," *Mobile Security Technologies*, vol. 2012, 2012.

[13] A. Desnos, "androguard – Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !)." [Online]. Available: http://code.google.com/p/androguard/

[14] 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*. ACM, 2013, pp. 73–84.

[15] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," in *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 50–61.

[16] "Google Play Developer Program Policies," Google, Inc., accessed July 2013. [Online]. Available: https://play.google.com/about/developer-content-policy.html

[17] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proceedings of the Network and Distributed System Security Symposium*, 2012.

[18] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services*. ACM, 2012, pp. 281–294.

[19] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2012, pp. 101–112.

[20] H. Hao, V. Singh, and W. Du, "On the effectiveness of API-level access control using bytecode rewriting in Android," in *Proceedings of the ACM SIGSAC Symposium on Information, Computer and Communications Security*. ACM, 2013, pp. 25–36.

[21] H. Lockheimer, "Android and security." [Online]. Available: http://googlemobile.blogspot.com/2012/02/android-and-security.html

[22] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 229–240.

[23] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *Proceedings of the Annual Computer Security Applications Conference*. ACM, 2011, pp. 343–352.

[24] *McAfee Threats Report: First Quarter 2013*, McAfee, 2013. [Online]. Available: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2013.pdf

[25] J. Oberheide and C. Miller, "Dissecting the android bouncer," *SummerCon New York*, 2012.

[26] J. Oberheide, "Android hax," *SummerCon New York*, 2010.

[27] N. J. Percoco and S. Schulte, "Adventures in bouncerland," *Black Hat USA*, 2012.

[28] N. Peter Loscocco, "Integrating flexible support for security policies into the Linux operating system," in *Proceedings of the FREENIX Track, USENIX Annual Technical Conference*. The Association, 2001, p. 29.

[29] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures." in *Proceedings of the USENIX Security Symposium*, 2010, pp. 1–12.

[30] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 2007, pp. 552–561.

[31] S. Smalley, "Configuring the SELinux policy," *NAI Labs Rep*, pp. 02–007, 2002.

[32] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *Proceedings of the Network and Distributed System Security Symposium*, 2013.

[33] S. Smalley, C. Vance, and W. Salamon, "Implementing SELinux as a Linux security module," *NAI Labs Report*, vol. 1, p. 43, 2001.

[34] T. Vidas, D. Votipka, and N. Christin, "All Your Droid Are Belong to Us: A Survey of Current Android Attacks." in *Proceedings of the USENIX Workshop on Offensive Technologies*, 2011, pp. 81–90.

[35] M. Weiser, "Program slicing," in *Proceedings of the International Conference on Software Engineering*. IEEE Press, 1981, pp. 439–449.

[36] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 79–93.

[37] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 95–109.

[38] ——, "Detecting passive content leaks and pollution in android applications," in *Proceedings of the Network and Distributed System Security Symposium*, 2013.

[39] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Proceedings of the Network and Distributed System Security Symposium*, 2012.