Comput Syst Sci & Eng (2020) 4: 271–282
© 2020 CRL Publishing Ltd

International Journal of
Computer Systems
Science & Engineering

A Proposal for Addressing Security Issues Related to Dynamic Code Loading on Android Platform

Aleksandar Kelec*[*] and Zoran Djuric[†]

Faculty of Electrical Engineering, University of Banja Luka, Banja Luka, Bosnia and Herzegovina

One of the constant challenges faced by the Android community, when it comes to the safety of the end users, is the ability of applications to load code dynamically. This mechanism may be used for both legitimate and malicious purposes. A particular problem is the fact that remote code is not analyzed during the verification process because it doesn't have to be present in the application package at the publishing time. Previous research has shown that using this concept in an insecure way can cause serious consequences for the user and his device. Solving this problem has proved to be a big challenge that many have tried to address in different ways. This paper deals with the problem of dynamic code loading on Android platform. For the purpose of this paper, an application that demonstrates the abuse of the dynamic code loading concept has been developed and published in the Google Play Store. Also, a proposal of the modified Android ecosystem that should address this problem and improve the security of the whole platform is given.

Keywords: Android security, Dynamic code loading, Privacy issues

1. INTRODUCTION

Android platform was designed to provide applications the ability to load an additional code from an external location during the execution, such as various third-party servers that offer additional functionalities (like ad serving). Dynamic Code Loading (DCL) mechanism brings a lot of flexibility when it comes to code reuse, application extensibility, rapid updating, improving application startup performance, etc. On the other hand, this option is very critical in terms of security, considering that potential attackers can add a malicious activity to the application after the application has successfully passed the security check at the publishing time. Since the remote code is not contained in the APK (Android Package Kit) file during the vetting process, while publishing an application in the Google Play Store, it is very difficult to analyze such a code and determine whether it is malicious or not. In addition, attackers can intercept the communication of applications that legitimately seek to download and execute remote code and modify that code or replace it with the other (malicious) one (MITM i.e. Man-In-The-Middle attack).

However, the previous research [1–7], has shown practically that DCL mechanism may be abused by malicious applications in order to circumvent the mechanisms of static and dynamic code analysis and perform malicious activities. Furthermore, research [3] has shown that the implementation of a secure DCL functionality is really challenging and non-trivial.

This paper describes techniques and attacks based on loading and execution of the remote code within the Android application. It conducts an analysis of several techniques which provide the use of DCL mechanism on the Android platform and gives examples of the use of this mechanism for both legal and malicious purposes (Section 2). Related work is discussed in Section 3. An application that demonstrates the abuse of the DCL with the aim of compromising user's privacy has been developed. This application is described in details in Section 4. Additionally, a proposal of the modified Android

[*]aleksandar.kelec@etf.unibl.org
[†]zoran.djuric@etf.unibl.org

ecosystem that could address the problem of verification of code that applications load at runtime is given in Section 5. The proposed solution introduces two additional components to the Android ecosystem - DCLVerifier service and verification server, which are in charge of detection and verification of the code that applications load during runtime. Section 6 describes testing and evaluation of the proposed system. Limitations of the proposed system and segments which could be improved are discussed in Section 7. Before concluding the paper, the comparison of the proposed solution with the existing solutions is given in Section 8.

2. MOTIVATION FOR DCL

There are various situations where it is suitable to use DCL techniques [3, 8, 9]. Previous research [3] has shown that 9.25% of 1,632 popular applications (including Facebook) from the Google Play store use DCL. Aysan and Sen [10] showed that 19.60% of 25,000 applications from three different markets use DCL for the purpose of updating. The research conducted by Maier et al. [11] on a sample of 14,885 malicious and 22,032 benign applications, showed that 36.4% of malicious and 13.1% of benign applications use DCL. Spreitzenbarth et al. [9] have found that 24% of 36,000 applications from alternative sources use native calls. Thus, a growing trend of applications that use DCL for different purposes can be noticed. These purposes include, but are not limited to, the following: new features testing, using common frameworks to extend application's functionality, updating third-party libraries which provide additional functionalities, using external modules to improve application performance and using add-ons.

2.1 Dynamic Code Loading Techniques

The combination of Class loaders and Java reflection provides an efficient implementation of DCL. However, this feature can also be used for malicious purposes, as it allows the application to be extended at runtime with additional code. The existing state of the art static analyzers for mobile applications [12–14] assume that the code base does not change dynamically and the targets of reflection calls can be discovered in advance. AnserverBot Trojan [15] illustrates the abuse of DCL and reflection in order to circumvent the mechanisms of static code analysis and perform malicious activities. The name and location of the file are not known at compile time but are computed at runtime. Thus, a malicious code is allowed to be loaded during the application's execution.

The Context class allows the Android application to access its resources as well as the resources of the other applications. The example of abusing the context of the Swift keyboard application is demonstrated in [16]. This application provides the keyboard on the Samsung devices. The application is signed by a manufacturer's private key and runs with system privileges. The attackers used a MITM attack to modify the zip file that application was downloading during the update and added the malicious code into it. After the application unzips the file, the malicious code is executed with system privileges.

In order to load the code into the Java Runtime Environment, the programmer has to go through a well-defined API. On the other hand, loading and execution of a code from the native environment are significantly easier and are based on the execution of the arbitrary executables. While the previous research [3] has argued that attacks that originate from the native code are in the domain of theory, more recent studies [8, 17, 18] have demonstrated that they are practicable and may be various. Furthermore, they are resistant to most of the static application analyzers, because they occur below the layer of the Dalvik Virtual Machine (DVM). Rasthofer et al. [18] showed how to abuse the native code in order to circumvent the tools for automatic application analysis. They have concealed a malicious code within the native code, making it unreachable for tools that analyze Dalvik bytecode.

The PackageManagerService is responsible for managing the installation and uninstallation of applications on the Android OS. There are situations where an application requires a direct installation of the APK, thus circumventing legitimate online markets. PackageManagerService verifies the digital certificate of each APK file during its installation, but it can not verify the trust level of the certificate because any self-signed certificate is acceptable.

In addition to techniques for loading a code that is executed within the DVM or on the native level, there are mechanisms for loading the code which are interpreted by specific applications. A typical example is the web browser which executes an arbitrary script written in the JavaScript language. Moreover, Android provides a WebView component that is able to display the web page within the Android application.

3. RELATED WORK

Different studies have examined the influence of DCL on Android applications, during the past few years. Most of them successfully identified the problems that may arise in the case of various abuses of this approach.

Poeplau et al. [3] have identified the various uses of the DCL mechanism as well as techniques that provide an implementation of the DCL on the Android platform. They have developed a tool that detects vulnerabilities in the applications that use this mechanism, as well as a mechanism for code verification. This involves a series of modifications to the Android framework that should detect any attempt to load remote code.

Zhauniarovich et al. [1] have examined the problems of the static analysis of applications that use DCL. They have shown that a significant number of applications from different sources use DCL and reflection. Specifically, on a sample of 29,406 applications, they showed that 12.3% of applications use a DCL and 81.1% uses reflection. Their tool, StaDynA, combines static and dynamic analysis to detect hidden method calls that can load malicious code. StaDynA allows the application to load the code. Then, StaDynA loads the same code and submits it to the static analyzer. Thus, it is possible to extend the method call graph (MCG) of the application and provide enhanced accuracy of the tools for static analysis. This tool can be used by teams who analyze malicious applications to enhance the ability of detection of the suspicious samples.

Vidas and Christin [6] proposed a mechanism that alleviates the specific problem of verifying the authenticity of an application, to protect the user from repackaged applications that contain malicious code. Their tool, AppIntegrity, is based on creating a simple public-key infrastructure and using reverse package names for fetching the certificates for signing. This system protects applications from unauthorized repackaging and allows applications to safely download and load remote code which has been proven as legitimate.

Falsina et al. [5] have developed Grab 'N Run, the tool which consists of a protocol for the code verification as well as a set of libraries and APIs that could provide an implementation of the secure DCL mechanism. This tool is an important step forward when it comes to the secure implementation of the DCL. An application must include the library in order to use its services. In addition, Grab 'n Run contains an application-rewriting tool, which allows existing applications to use the secure APIs they were developed. This solution is based on the *DexClassLoader* class as well as corresponding API for the code loading, while other techniques that provide the implementation of DCL are not supported.

Ali-Gombe et al. [8] have presented AspectDroid, a system for dynamic analysis of applications which, among other things, focuses on DCL and reflection. The system is based on a static instrumentation of the bytecode at the compile time and detection of potential leaks of sensitive information. AspectDroid also includes a containment policy that allows defining the way in which sensitive calls can be restricted to the applications. In this way, the system can block execution of the code for which it is determined that it intends to send sensitive information to a remote location.

Athanasopoulos et al. [7] have studied applications that use third-party libraries containing native code. They showed that native code can completely modify the runtime environment, given that the native code is mapped to the same address space as well as the runtime environment, i.e. DVM. Their framework, NaClDroid, separates the native code from the application during the runtime and places it into the specific sandbox. Thus, the framework allows an application to use native calls, while concurrently preventing it from arbitrarily reading process memory, tampering with the Dalvik runtime, or directly accessing operating system interfaces, like system calls.

Spreitzenbarth et al. [9] developed a Mobile-sandbox which combines static and dynamic analysis in order to find any suspicious calls and activities. The results of the static analysis are used to guide dynamic analysis and extend coverage of executed code. The main purpose of this system is to provide malware investigators with an efficient tool for malicious behavior detection.

The problem of improper use of the DCL is characteristic not only for Android but also for the other mobile operating systems. Wang et al. [19] demonstrated issues related to verification of malicious applications on the iOS platform. They developed a proof of concept malicious iOS application that successfully passed the review process on Apple's App Store. The code submitted for review was benign, however, the application was able to update itself in order to introduce malicious control flows and to perform various illicit tasks.

4. THE FAST NEWS APPLICATION

Fast News [20] is an Android application developed with the aim to show that DCL mechanism can be abused in order to compromise user's privacy. The main purpose of this application is to provide the user with the latest news. Besides its main purpose, the application also performs one malicious activity: downloads and executes a malicious code that violates the user's privacy and steals confidential information.

Although Google has lately devoted significant attention to protecting user's privacy and providing a safe and secure environment for its users, the Fast News application shows that this is not applied sufficiently in practice. Android's Developer Program Policies Update from June 2017 has paid a special attention to the malicious behavior originating from the download of executable code [21]. They claim that an app distributed via Google Play may not modify, replace, or update itself using any method other than Google Play's update mechanism. Likewise, an app may not download executable code (e.g. dex, JAR, .so files) from a source other than Google Play. However, this paper shows that it is possible to publish an application that loads the code from the third-party server. The Fast News application has successfully passed all the tests during vetting process and found its place in the Google Play Store. Thus, it has become available to the users worldwide. Furthermore, three more applications have been published in September 2017, showing that Google does not apply mentioned policy properly. More details about these applications are given in the next section.

4.1 The Attack

Fast News application provides users with the latest news. When the application is started, the first screen shows the news list, while in the background a remote code is downloaded and executed without the user's knowledge. This code tries to collect sensitive information from the device and send them to the remote server. Figure 1 shows the basic function of the Fast News application, as well as the attack that it performs. The working principle can be explained through the following six steps:

1. When started, the application sends a request to the server for the latest news. In order to create a response, the server contacts certain free online services, which return the latest news, in the RSS format. Thereafter, the server parses those responses and sends the final response to the application with the latest news in JSON format.

2. After receiving the response, the application parses it and displays to the user a list of news articles.

3. Then, the application sends another request, without the user's knowledge, requesting an APK file which contains a malicious code. Sending request, as well as downloading and storing APK file in the memory of the device, is performed within an AsyncTask. Thus, the main thread runs without interrupts, so the user will not find anything suspicious. In addition, the application checks whether the device is connected to a WiFi network before sending the

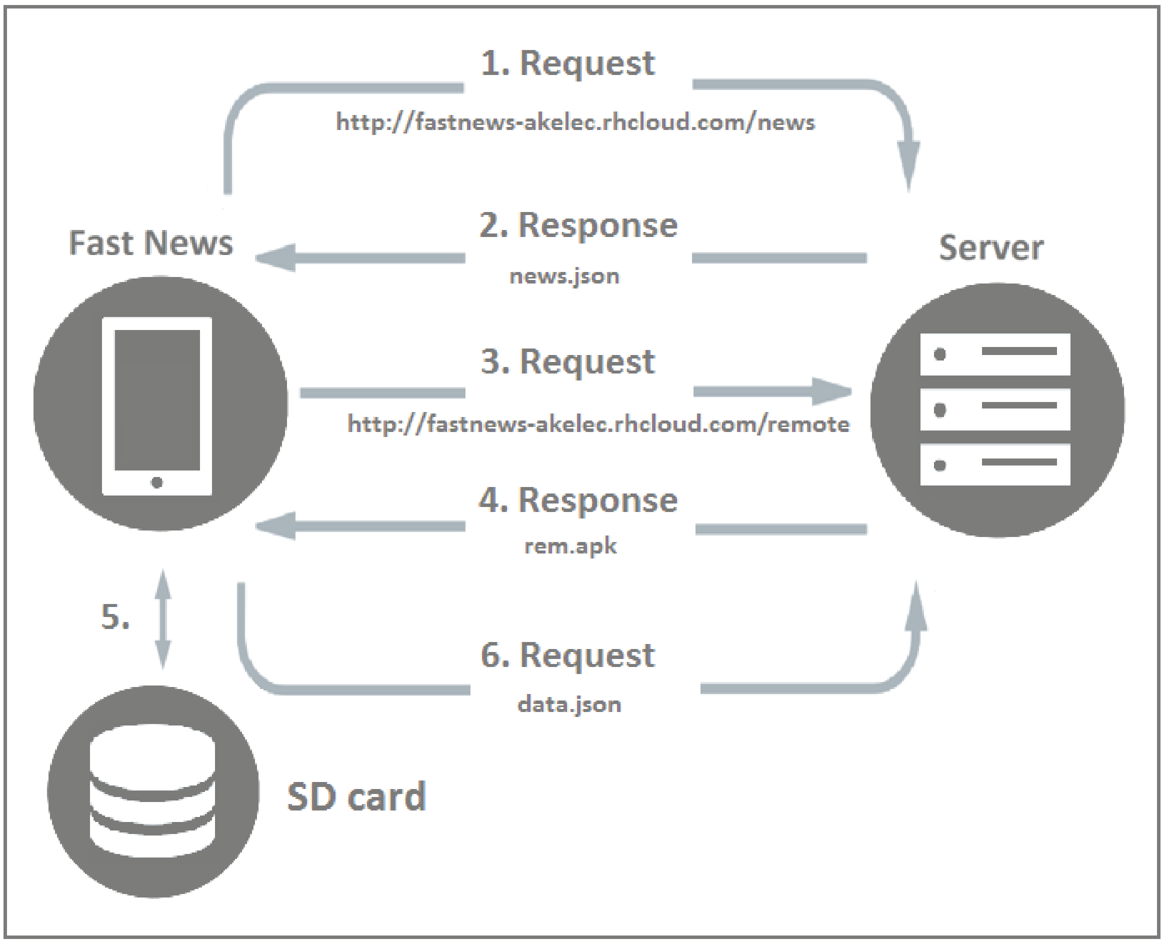

Figure 1 Fast News application - the basic mechanism of the attack

request because the cellular network usage and MMS might cause some extra fees.

4. The application downloads the APK file and stores it in the temporary directory of the external memory.

5. In order to read from the device's external memory and write to it, the application should request appropriate permissions. Requesting these permissions could be suspicious to the user. However, this could be justified by the additional explanation that application periodically caches the news and images while running, so the user could read the news even when he is offline. Actually, the application really does that. In addition, it would be possible to implement a functionality which allows the user to save favorite articles and images. However, it is not hard to find justification for a request of some, at first glance not logical, permission. Especially if one has in mind that requesting such permission has passed Google's vetting process.

6. When the file download is finished, the application loads it using the class DexClassLoader and makes a call to the certain method. This method searches the external memory and collects potentially sensitive information and packs them into a single file. Subsequently, the file is sent to the attacker's server. After the attack was successfully executed, the application removes the directory containing the malicious APK file and continues to perform its primary function.

4.2 An Attacker's Possibilities

After all the steps have been successfully completed, the appropriate file on the server is updated with the new data collected from the end users. A part of this file is shown in Figure 2. In this case, the data related to the specific device and the version of the operating system are sent to the server.

However, in this way, the attacker could also gather other data, such as user's credentials, banking data, contact information, SMS messages, search history, etc. Therefore, the application will misuse permissions allowed by the user, in order to gather personal information and send them to the server.

An attacker could use collected data to monitor the user and analyze his behavior. Thus, he could try to perform some activities on the target group of users, including identity theft and compromising user's privacy.

Limitations of the applications that misuse a DCL largely depend on users and their decisions. Such applications usually have as much freedom as the users give to them, which means that their limitations are related to the permissions assigned by the users. This especially refers to devices with the latest versions of the Android OS (Oreo, Nougat, and Marshmallow), where users can selectively grant permissions to the applications. However,

```
2018-02-17 (08:21:01)
Debug info:
    OS Version: 3.4.0-perf-gd2f961e(D80210e)
    OS API Level: 17
    Device: g2
    Hardware: g2
    Board: MSM8974
    Model: LG-D802
    Product: g2_open_com
-------------------------------------------
2018-02-26 (10:48:18)
Debug info:
    OS Version: 3.18.14-11940524(A520FXXU2BQH4)
    OS API Level: 24
    Device: a5y17lte
    Hardware: samsungexynos7880
    Board: universal7880
    Bootloader: A520FXXU2BQH4
    Model: SM-A520F
    Product: a5y17ltexx
```

Figure 2 A part of a file with the confidential information collected by the Fast News application

considering studies [22, 23] which have shown that a very small subset of users even reads information about permissions, the Fast News application has no significant limitations from that point of view.

5. EXTENDED ANDROID ECOSYSTEM

In order to install an application on the device, Android OS contacts the appropriate server that provides applications, such as the Google Play Store. This procedure, which begins when the application is published in the Google Play Store and ends with application's installation on a user's device, is followed by a number of security mechanisms, such as Google Play Review [24], SafetyNet [25], Google Play Protect [26], etc. Some of these mechanisms are placed on the server and some on the Android device. They are responsible for protecting the entire Android ecosystem from malicious applications and a variety of fraud originating from the external environment. Although they are introduced as strong and secure mechanisms, in the previous section it is practically demonstrated that they can be circumvented, putting a significant number of users at risk. Google Play Review has allowed the Fast News application to be found in the Google Play Store, while Google Play Protect has allowed the application to be installed and launched on the user's device. Even after a subsequent scan, this mechanism has not detected the Fast News application as malicious.

In order to address problems related to the execution of the remote code which is malicious, a proposal of the extended Android ecosystem is presented in this section. The extended system relies on DCLVerifier, an Android service which should be responsible for detecting every download of a remote code and for sending that code to the verification server before the application is allowed to load and execute the code in question. In this way, it is not allowed to execute any remote code that has not been verified and marked as legitimate.

5.1 DCLVerifier Service

For security reasons, Android requires that all the content which applications download from a remote location must be stored in the external memory of the device before the content is used. Exactly this restriction has served as the basis for the implementation of the DCLVerifier. DCLVerifier is an Android service whose assignment is to observe the changes occurring in the external memory and thus detect any attempt to download any remote code. For observing the file system, it uses a modified version of the class FileObserver, called RecursiveFileObserver, which allows detection of changes in the entire tree of directories and files. In order to detect changes in the entire file system, DCLVerifier is configured to observe the root directory of the external memory.

DCLVerifier runs in the background, thus, the user of the device is not aware of its presence. Furthermore, it is very lightweight when it is about resources consumption, so the impact on the device performance is negligible (this will be discussed in more detail in Sections 6 and 7). The service starts automatically after the device powers on. The BroadcastReceiver launches the DCLVerifier service after receiving the action BOOT_COMPLETED from the OS. This service is running in the background as long as the device is turned on so that it could not be circumvented.

When detects the presence of a new file in the file system, DCLVerifier first replaces the original file name and extension, with some randomly generated array of characters. Thus, an

```
1 10:47:47.066 fastnews    REQUESTING: http://.../rem.apk
2 10:47:47.068 dclverifier DETECTED: /storage/.../rem.apk
3 10:47:47.068 dclverifier RENAMED: /storage/.../q1$.14
4 10:47:47.068 dclverifier VERIFYING: /storage/.../q1$.14
5 10:47:47.172 dclverifier APPROVED: /storage/.../rem.apk
6 10:47:48.173 fastnews    LOADING: /storage/.../rem.apk
```

Figure 3 An excerpt from the log files of the DCLVerifier service and the Fast News application at the moment of downloading the file

appropriate application cannot find the expected file by its name. Although this step does not seem to be quite safe given that renaming the file could produce a race condition or would not hide the file sufficiently well and it could still be found by a malicious application, this approach has proven to be convincing and reliable. For instance, a malicious application could search an external memory in order to find a file by its hash, but this is an incomparably longer process than the verification process performed by the DCLVerifier. Alternatively, the DCLVerifier could use the class FileLock to lock the file after its detection. In this way, other processes would be prevented from accessing the file until the DCLVerifier releases it. However, such an approach has not proved to be necessary.

After the file is downloaded, the service checks if the file contains any piece of the executable code or it is the arbitrary file, such as image, audio or video file. This check is performed inside the method containsExecutiveCode which returns true if one of the following conditions is met:

- APK file, i.e. an archive containing the files such as classes.dex, AndroidManifest.xml, etc.,
- JAR file, i.e. an archive containing files with the Java code,
- other archives that contain at least one file with the extension .class or .java,
- an executable file, i.e. a file containing some of the famous magic numbers related to the executable file types [27].
- a binary file with the unknown content, i.e. a file which mime type could not be determined by calling a method probeContentType of the class Files or methods of the class MimeTypeMap, or a file that has the mime type application/octet-stream, representing an arbitrary binary content.

In the case of some of these files, the service computes the hash of the file, signs it digitally and sends to the verification server. The original name will be returned to the file, only after it is determined that the code that is contained within the file is not malicious. Otherwise, the service will remove the file and add its hash in the register of the malicious files. If the method containsExecutiveCode returns false, the service will automatically restore the original file name, so the appropriate application will have the access to the downloaded file.

If the application tries to dynamically load multiple files, the service will detect any attempt and all files will be processed separately. This is possible due to the fact that the service is executed within the AsyncTask. In this way, when the new file is detected on the file system, a new instance of the service, which is responsible for the processing of that file, is created.

DCLVerifier has successfully detected any attempt to load a malicious code by the Fast News application. During the first launch of the application, the hash of malicious code was added in the register of malicious applications, thus permanently preventing applications to load this file. An attacker could try to circumvent this hash-based approach by creating multiple different files and injecting the same malicious code in them. Although each file would have a different hash, the service would detect each file as malicious and add all these different hashes to the database. In addition, an attacker could generate several different applications that will load the same malicious file, but such an attempt will also be blocked by the DCLVerifier. In order to demonstrate this, three more applications, Fast News Sport, Fast News Cars and Fast News Business, have been created. All three were published in the Google Play Store successfully and were installed on the user's device, but all three were stopped in an attempt to load malicious code.

DCLVerifier is implemented to verify the presence of a new file on the file system and react before the application that initiated the file download comes into possession of it. This is possible due to the fact that the class FileObserver detects the presence of the new file as soon as the first byte of that file is saved on the file system. Unlike to this class, an application that tries to download the file has to wait until the entire file is downloaded. As a proof of this claim could be used an excerpt from the log files of the DCLVerifier and the Fast News application. The conjoint excerpt of those two logs is presented in Figure 3. The excerpt refers the period from the beginning of the (benign) file download to the completion of its verification and loading by the application. Line 1 shows the moment when the application initiates the file download. Line 2 shows the moment when DCLVerifier detects the presence of a new file in the file system. After that, the service renames the file with some randomly generated string of characters, as shown in Line 3. Line 4 refers to the process of verification of the file to determine whether a file contains malicious executable code. Lines 5 and 6 show the moments when the service returns the original name to the file and the application loads the file, respectively.

The figure shows that the DCLVerifier service detected the presence of the file as soon as download started. On the other hand, the Fast News application tries to load a file from the external memory after the file is verified. Thus, the application could not load the file before the file is approved.

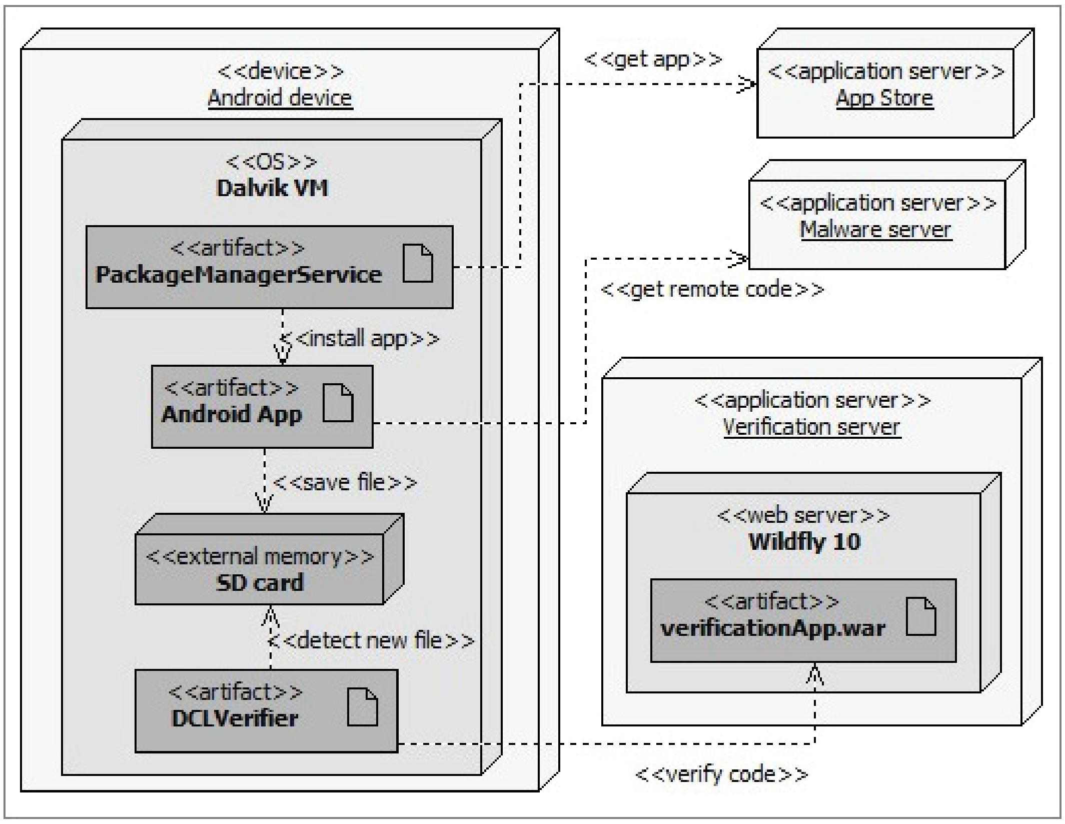

Figure 4 Deployment diagram for the modified Android ecosystem

5.2 Extended Ecosystem Overview

Figure 4 shows a deployment diagram for the extended Android ecosystem. The system implies the existence of the verification server which communicates with the DCLVerifier.

Since the remote code has an equivalent structure as the application code, the existing mechanisms that verify applications while publishing in the Google Play Store can also be used for the verification of downloaded code. Google Play Review is a security mechanism responsible for the detection of potentially dangerous applications and preventing such applications to be in the Google Play Store. Due to the comprehensiveness of this mechanism, this paper does not focus on the creation of a new protocol for code verification, but it proposes the introduction of the verification server in the Android ecosystem.

Verification server hosts an application that has the assignment to compare the hash values and to verify the remote code. The server maintains two registers which contain digital signatures of the files that are verified. The first register contains signatures of the legitimate files, while the second register contains the signatures of the malicious files. The verification server would be introduced as a separate module, so the existing system would not have to be subject to any modifications.

5.3 The Working Principle

Integrity checks that this system imposes, include searching the register which contains the digital signatures of the legitimate files. The digital signature of the code implies the hash computed by the SHA-256 algorithm.

The working principle of the proposed system consists of the following three steps:

1. When an application attempts to download and execute a remote code, DCLVerifier service detects the attempt and computes the hash of the code. Then the service sends the hash to the verification server. The server then searches the register with the signatures of legitimate files to find a match with the received hash. If the match is found, the server sends a response to DCLVerifier with the permission that allows the application to load the file. Otherwise, the server will search the register of the malicious files. If the match is found in this register, the server will send a response to not allow the code loading. If there is no match in any register, then the server will require the downloaded file for its verification.

2. After receiving the response, DCLVerifier decides the next step. If the server requires a remote code, the service adds it to the new request and sends to the server. The server verifies the code in order to determine whether it is malicious. The tests could be similar to the those performed within the Google Play Review mechanism. If the tests show that the code is legitimate then its hash will be added to the register with signatures of legitimate files. Otherwise, the register with signatures of malicious files will be updated. After that, the server returns a response to the DCLVerifier service.

3. DCLVerifier service decides whether the application will be allowed to execute the downloaded code.

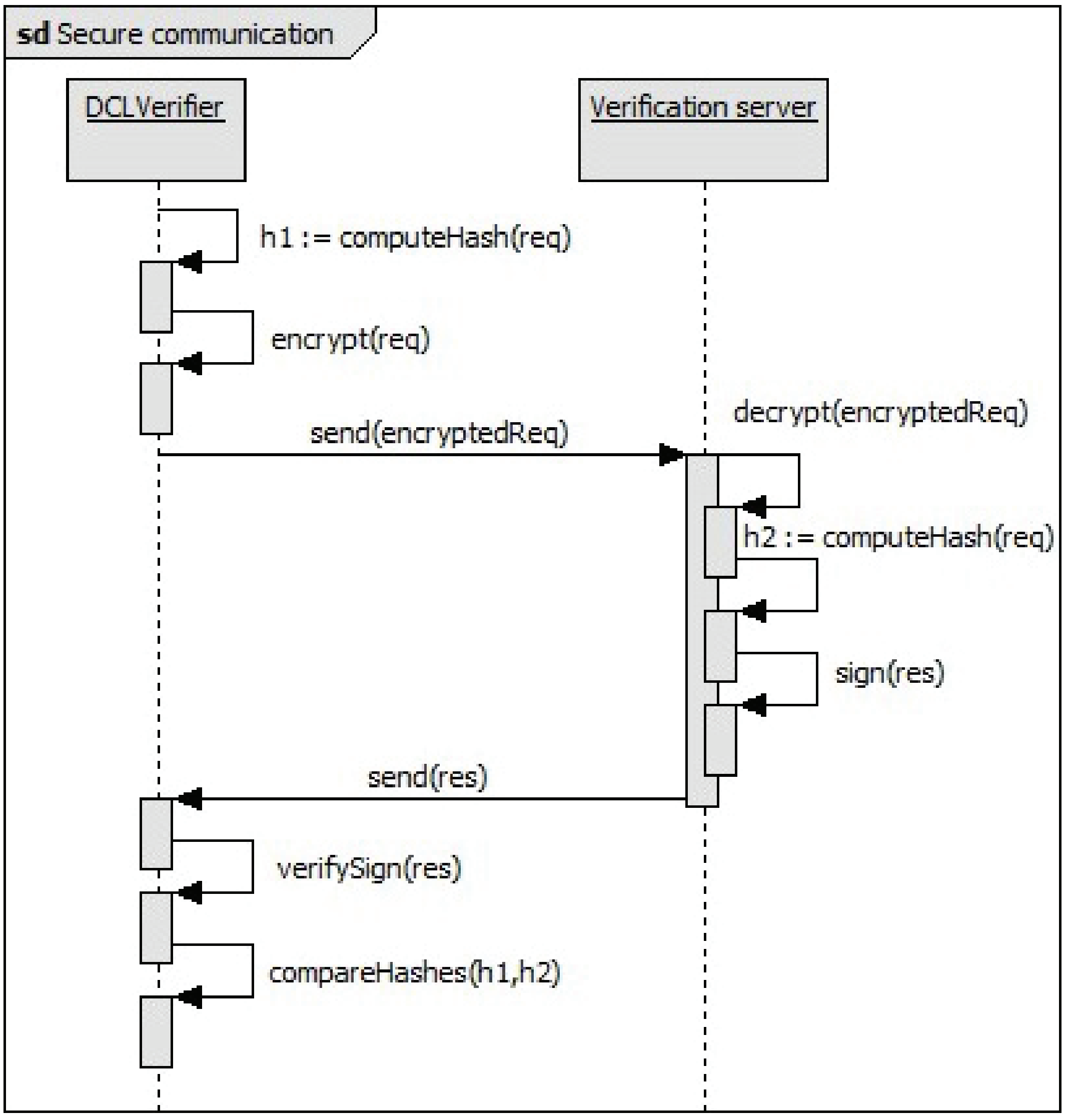

Figure 5 Protected communication between the DCLVerifier service and the verification server

5.4 The Secure Communication

In order to protect a communication between DCLVerifier service and the verification server, it is necessary to use appropriate algorithms and protocols. In the case of the insecure communication, an attacker could perform MITM attack and modify the data contained in the request and response. In this way, for instance, the user's device could allow execution of a malicious code.

It is well known that all the applications that are published in the Google Play Store have been signed by a programmer's self-signed certificate. This means that there is no certificate authority to issue certificates to the participants and verify them during communication. Creating a PKI (public key infrastructure) and storing certificates of all Android devices on the server side would significantly facilitate the implementation of the secure protocol for the communication between the DCLVerifier and verification server. However, it would require too much effort, so that idea was not considered in this paper.

Instead, the proposed system implies that the user devices keep the server's certificate for the response verification. This certificate could be forwarded to the user devices through the secure channel using a simple update from the Google Play Store. The certificate would be stored in an appropriate system directory, in order to protect it from an unauthorized use. Figure 5 shows a Sequence diagram that models the protected communication between the DCLVerifier and the verification server.

The protected communication can be described as follows:

1. When a request is prepared, the DCLVerifier computes a hash (h1) of the entire request and saves it for the later verification. Then, DCLVerifier encrypts the request using the public key from server's certificate and sends encrypted data to the server.

2. After the request is received, the verification server can decrypt it using its private key and extract the hash of the code from it. When the search of registers is finished, the server creates a response for the client. In addition, the server computes a hash (h2) of the request in the same way as the DCLVerifier and includes it in the response. Then, the server digitally signs the response and sends it to the client.

Table 1 Results of testing a DCLVerifier service with various file types.

File type	**Detected**	**Proc. time (ms)**	**Detection rate (%)**
APK	10/10	120	100
JAR	10/10	114	100
class	10/10	88	100
dex	10/10	94	100
exe	10/10	93	100
com	10/10	92	100
bin	19/20	84	95
binary file	20/20	106	100
unknown file	18/20	102	90
jpg	0/10	0	0
pdf	0/10	0	0
avi	0/10	0	0
Summary	**117/120**	**99.22**	**98.33**

3. After receiving the response, DCLVerifier first verifies the digital signature. Then, it checks whether the received hash (h2) matches to the computed one (h1). If hashes are matched, it means that the server received an original and unmodified request and the service should consider the response content. If hashes do not match, the service will reject the response.

The proposed communication model can be considered a safe and resistant to attacks. The only way to break this secure communication is by performing hash collision attack. Since it is proven that the SHA-1 algorithm is not completely resistant to collision [28], the proposed system uses the SHA-256.

Additionally, this communication could be based on a more complex protocol, which would include a Session ID, in order to protect the server against invalid requests. Such a protocol implies establishing a secure tunnel between the client and the server and generating a Session ID on the server. The Session ID must be included in the first message that is sent from the server to the client. Furthermore, the client should include a Session ID obtained from the server at every request. However, such communication protocol would introduce an additional delay and would slow down the whole process of the code verification.

6. TESTING AND EVALUATION

Any new software should go through a detailed process of testing and evaluation in order to determine its relevance, efficiency, and effectiveness. When it comes to the proposed system, DCLVerifier and its capability of detection files with executable code, as well as the communication to the verification server, are the main components to be tested. Furthermore, the impact of this service on the performance of the device as well as the resources needed to perform such a processing is very important.

For testing DCLVerifier a test set of 150 files of various types has been created. The testing application loaded these files dynamically using DexClassLoader class. In addition, a simple prototype of the verification server is developed for testing purposes. On the server, the two registers with 100,000 different hashes have been created.

Testing results are shown in Table 1. The results show the success of detection of files that could potentially contain executable code. In addition, the average time required to process these files is shown. When it comes to the standard Java-based file types, such as APK, JAR, class, dex, etc., the detection rate is very high. DCLVerifier has detected all of 40 files of this type and sent them to the verification server. The same success rate has been achieved when it comes to executable files, i.e. files with the extension exe, com, etc. When it comes to the files with unknown (binary) content, service has detected 57 out of 60 files that potentially contain executable code, while 3 files were omitted. Other file types, such as pdf, jpg, avi, etc. are not detected because they have famous magic numbers, that are not related to the executable files. Thus, the application has been able to load them.

Table 1 shows that the average detection rate of the files that potentially contain executable code is about 98%. However, when calculating this percentage, files which are recognized as files with the standard types, have not been considered.

Furthermore, Table 1 shows the average time required to process a file. Processing time includes time needed for detection of the file, time spent for hash computation and sending to the server, a time needed for searching the registers and time spent for returning the results back to the device. The table shows that the average time required to process any type of file is about 99 milliseconds. In the tests devices with the following specifications have been used:

- Samsung Galaxy A5 with octa-core 1.9 GHz CPU,
- Red Hat server, i.e. a virtual machine powered by two vCPUs with memory ratio up to 512 MB and 1GB of RAM.

The results show that such a system could operate without the need for allocation of the significant amount of resources on both sides (device and server). When installed on a device, DCLVerifier occupies only 5.04 MB of the storage. Furthermore, the time required to process files that applications dynamically load should not affect the performance of the devices and applications.

An analysis of power consumption is also an important aspect to consider when it comes to introducing additional processing on mobile devices. It is well known that smartphones have limited power resources, so any additional consumption could have negative consequences. Research [29] has shown that the majority of power consumption can be attributed to the GSM module and the display when it comes to smartphones. Furthermore, the paper [30] has shown that the CPU-intensive calculations are minor power consumers compared to some hardware components, such as GPS receiver. DCLVerifier runs in the background and most of the time it resides in a sleeping mode. It wakes up only when a new file appears on the device. The Android operating system runs many services, so the addition of DCLVerifier service increased the power consumption insignificantly. Additionally, the "App Info" from the device's settings has shown no battery draining.

7. LIMITATIONS

Although the proposed system shows the potential to seriously and fully address the problem that exists since the very beginnings of the Android, there are certain limitations of the proposed system and segments which could be improved.

7.1 Static Analysis

Static analysis typically involves inspection of the application bytecode and monitoring of instructions and calls that can be potentially dangerous and lead to leaks of confidential information. Dynamic analyzers monitor the application during its execution and focus on the actions that application performs dynamically. The system described in this paper is based solely on the dynamic analysis and does not consider the APK file, but only the content that application downloads at the runtime. Theoretically, it is possible to pack a malicious code within the APK, which the application would load and execute at the runtime. However, detection of such code is subject to mechanisms for verification of the application before its publication.

7.2 Detection of Files With Executable Code

Although the DCLVerifier is very effective when it comes to detecting new files in the external memory, it is hard to determine the success of detection in the real system. Section 5 describes an algorithm that, among other things, uses a method of the magic number for detection of executable code in files. Theoretically, there is a chance to circumvent this checking mechanism if the magic number is removed from the file or changed to an unknown value. However, this situation should be also addressed by considering that, in this case, it comes to the binary file with unknown content, which would be detected. In addition, this algorithm also uses other approaches to detect files with executable code.

7.3 Executing Scripts

If they want to avoid executing code directly on the Android platform, attackers can take advantage of some of the scripting languages for the injection of malicious code. In that case, the application does not have to load executable code, but a simple text file that will be interpreted by the appropriate interpreter. Examples of misuse of the WebView related to this approach are given in [31]. Unfortunately, this paper does not address such attacks, given that it is a code that is not executed directly on the Android platform but in the context of the web browser. Obviously, this is more subject to web security issues.

7.4 Performance Impact and Costs

The proposed system introduces additional components in the Android ecosystem, which implies certain costs. A number of verification servers could grow significantly, depending on the amount of the work that should be performed. The introduction of the DCLVerifier service entails additional processing on the device. However, the vetting process that the DCLVerifier performs is very efficient and measured in milliseconds, so the applications should not experience any degradation of performance. Unlike sending a hash of the file to the server, sending the entire file could have a greater impact on the performance. This could be especially important if the application repeatedly obtains remote files during execution. However, the assumption is that these situations are rare and applications that load code dynamically, do not perform it too often because it would significantly affect the application's performance. Sending the file to the server, also should not be time-critical, if we take into account that the size of these files is usually in the order of kilobytes. Tests have shown that this time is usually expressed in milliseconds.

8. COMPARISON WITH THE EXISTING SOLUTIONS

The approach described in this paper has certain similarities with the approach used in [3]. However, there are several key advantages of the approach presented in this paper that make it feasible as opposed to [3]. The main difference relates to the method of detecting DCL as well as the layer of architecture which implements this mechanism. DCLVerifier service can be quickly and easily installed on all Android devices with just one update from the Google Play Store. On the other hand, modifications within the Android OS that have been proposed in [3], require updating the OS on all devices, which is often a very slow and lengthy process, when it comes to Android. Furthermore, when one considers the high level of fragmentation of the Android OS versions [32], it is very likely that a large number of devices would never receive the necessary update. On the other hand, any intervention and modification of the system are very easy to apply by simply publishing an updated version of DCLVerifier service, as opposed to complicated changes within the OS. In addition, in the approach that is described in this paper, the complete processing related to verification of the remote code is performed on the verification server, unlike to [3], wherein the verification performs mainly on the device, making the verification mechanism transparent and vulnerable.

StaDynA [1] is the system that has successfully addressed the problem of hidden calls related to the DCL mechanism.

The system combines static and dynamic analysis to enhance an MCG of the application. However, in order to detect malicious DCL calls by the system, an application must initiate them and load malicious code. In other words, this system cannot prevent the application to dynamically load malicious code, but only to find that the code loaded, is malicious.

AspectDroid [8] uses a similar approach, performing the static instrumentation of the bytecode at the compile time and detecting potential leaks of sensitive information. Furthermore, this system includes a containment policy that allows defining the way in which sensitive calls can be restricted to the applications. However, this component is not described in detail in [8], and it is not known on what principle it works and whether it can detect all of the techniques for the implementation of DCL mechanism, described in this paper.

The solution offered by Vidas and Christin [6] is based on the PKI system and using reverse package names for fetching the certificate for signing. Although this work represents a significant step forward in ensuring secure communication and protecting remote code against unauthorized modifications, yet it does not solve all the problems that are related to the DCL concept. Specifically, this system protects applications from unauthorized repackaging and allows applications to safely download and load remote code which has been proven as legitimate. However, this system generally cannot detect attempts to dynamic load malicious code and thus cannot prevent misuse of the DCL.

Grab 'N Run [5] is the library that allows the implementation of secure DCL concept, wherein it must be included in the application in order to provide its services. In this way a dependency between the application and the external library is created, whereby a loss of the flexibility, portability, and reliability of the entire system occurs. In addition, this solution is acceptable only for applications that use *DexClassLoader* class and associated API to load code, while other methods for DCL, which are described in this paper, are not covered. Furthermore, this solution, as well as one described in [6], provides the implementation of the secure DCL only when it is used in the legitimate purposes, while illegal attempts to download and execute the code are not addressed. In other words, the Grab 'N Run is statically oriented system and cannot detect attempts of dynamic code loading by the malicious applications.

NaClDroid [7], as well as other systems based on the concept of the sandbox, has addressed the problems related to the injection of the malware through native code. However, the techniques for dynamically code loading at the layer of DVM are not covered by these solutions.

9. CONCLUSION AND FUTURE WORK

This paper describes the extended Android ecosystem that addresses the problems related to the abuse of the DCL. DCLVerifier service, as the main component responsible for preventing the applications to load a malicious code, is developed. In addition, the proposal of an organization of the verification server is given as well as the working principle of the whole system. The proposed system has shown promising results in detecting files downloaded from third-party servers that could potentially contain executable code. The paper also describes the application whose publication in the Google Play Store has shown that the existing security mechanisms still do not properly address the abuses of the DCL mechanism.

As a future work, the plan is to develop a prototype of the verification server as well as a complete system with associated components and to optimize it in order to minimize the impact on device's performance. This particularly refers to the development of the new version of DCLVerifier which should increase the speed of processing by caching the hashes of the files that are most frequently downloaded. It also refers to a new version of the algorithm which DCLVerifier uses to decide whether the detected file potentially contains executable code. In addition, a detailed testing of the system and determining its efficiency against all DCL-related techniques is imposed. As a part of the future work, the plan is to integrate the appropriate static analyzer in the proposed system, to achieve complete protection against malicious code. Finally, the plan is to evaluate the system against known malware databases.

This paper deals with Android as the most widespread operating system. Although other operating systems are based on different technologies, there are many common features that would facilitate the implementation of the proposed system. For instance, the iOS ecosystem has many similarities to the Android ecosystem, such as application publishing, permissions model, etc. Wang et al. have shown that iOS applications perform dynamic code loading similarly to Android applications [19]. This encourages the authors to argue that the proposed solution can be ported to other operating systems while retaining core functionality and algorithm logic. Future work will include plans to adapt and apply the proposed solution to other mobile operating systems.

REFERENCES

1. Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications," in *Proc. 5th ACM Conference on Data and Application Security and Privacy*, ACM, 2015, pp. 37–48.
2. V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks," in *Proc. 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 2013, pp. 329–334.
3. S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," in *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.
4. L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna, and F. Maggi, "Grab 'N Run: Secure and Practical Dynamic Code Loading for Android Applications," in *Proc. 31st Annual Computer Security Applications Conference (ACSAC)*, 2015, pp. 201–210.
5. Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proc. of the 2012 IEEE Symposium on Security and Privacy (S&P)*, 2012, pp. 95–109.
6. T. Vidas and N. Christin, "Sweetening android lemon markets: Measuring and combating malware in application marketplaces," in *Proc. of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013, pp. 197–208.
7. E. Athanasopoulos, P. V. Kemerlis, G. Portokalidis, and D. A. Keromytis, "NaClDroid: Native Code Isolation for Android

Applications," *Computer Security – ESORICS 2016. Lecture Notes in Computer Science*, vol. 9878, pp. 422–439, 2016.

8. A. Ali-Gombe, I. Ahmed, G. G. Richard, III, and V. Roussev, "AspectDroid: Android App Analysis System," in *Proc. 6th ACM Conference on Data and Application Security and Privacy*, 2016, pp. 145–147.
9. M. Spreitzenbarth , F. Freiling , F. Echtler , T. Schreck, and J. Hoffmann, "Mobile-sandbox: having a deeper look into android applications," in *Proc. 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 1808–1815.
10. A. I. Aysan and S. Sen, "Do you want to install an update of this application? A rigorous analysis of updated android applications," in *IEEE 2nd International Conference on Cyber Security and Cloud Computing (CSCloud 2015),* New York, USA, 2015, pp. 181–186.
11. D. Maier, M. Protsenko, and T. Muller, "A game of droid and mouse: The threat of split-personality malware on android," *Computers & Security*, vol. 54, pp. 2–15, 2015.
12. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 259–269.
13. J. Hoffmann, M. Ussath, T. Holz and M. Spreitzenbarth, "Slicing Droids: Program Slicing for Smali Code," in *Proc. 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 1844–1851.
14. W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no.2, pp. 1–29, 2014.
15. Y. Zhou and X. Jiang, "An Analysis of the AnserverBot Trojan", http://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_ Analysis.pdf, last accessed April 2018.
16. R. Welton, "Remotely Abusing Android", https://www.blackhat .com/docs/ldn-15/materials/london-15-Welton-Abusing-Android -Apps-And-Gaining-Remote-Code-Execution.pdf, last accessed April 2018.
17. R. Fedler , M. Kulicke, and J. Schütte, "Native code execution control for attack mitigation on android," in *Proc. 3rd ACM workshop on Security and privacy in smartphones & mobile devices*, 2013, pp. 15–20.
18. S. Rasthofer, I. Asrar, S. Huber, and E. Bodden, "How Current Android Malware Seeks to Evade Automated Code Analysis," *Information Security Theory and Practice. Lecture Notes in Computer Science*, vol. 9311 pp. 187–202, 2015.
19. T T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, "Jekyll on iOS: When Benign Apps Become Evil," in *Proc. 22nd USENIX Conference on Security*, 2013, pp. 559–572.
20. A. Kelec, "Fast News, Google Play Store", https://play.google .com/store/apps/details?id=com.akelec.fastnews, last accessed April 2018.
21. Google, "Developer Policy Center", https://play.google.com/ about/privacy-security/malicious-behavior/, last accessed April 2018.
22. A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proc. of the Eighth Symposium on Usable Privacy and Security*, 2012.
23. A. K. Jha and W. J. Lee, "Analysis of Permission-based Security in Android through Policy Expert, Developer, and End User Perspectives," in *Journal of Universal Computer Science*, vol. 22, no. 4, pp. 459–474, 2016.
24. Google, "How we keep harmful apps out of Google Play and keep your Android device safe", https://static. googleuser-content.com/media/source.android.com/en//security/reports/ Android_WhitePaper_Final_02092016.pdf, last accessed April 2018.
25. J. Kozyrakis, "SafetyNet: Google's tamper detection", https://koz.io/inside-safetynet/, last accessed April 2018.
26. Google, "Protect against harmful apps", https://support.google .com/accounts/answer/2812853?hl=en, last accessed April 2018.
27. G. Kessler, "File signatures table", http://www.garykessler.net/ library/file_sigs.html, last accessed April 2018.
28. M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full SHA-1," *accepted at* CRYPTO 2017.
29. A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone", in *Proc. of the 2010 USENIX conference on USENIX annual technical conference*, Boston, MA, 2010, pp. 21–21.
30. C. Thompson, D. Schmidt, H. Turner and J. White, "Analyzing Mobile Application Software Power Consumption via Model-driven Engineering", *Advances and Applications in Model-Driven Engineering*, Chapter 16, IGI GLOBAL, 2014, pp. 342–366.
31. B. Hassanshahi, Y. Jia, R. H. C. Yap, P. Saxena and Z. Liang, "Web-to-Application Injection Attacks on Android: Characterization and Detection," *Computer Security - ESORICS 2015. Lecture Notes in Computer Science*, vol. 9327, pp. 577–598, 2015.
32. X. Zhou, Y. Lee, N. Zhang, M. Naveed and X. Wang, "The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations," in *Proc. of the 2014 IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 409–423.