StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications

Yury Zhauniarovich
University of Trento
Trento, Italy
yury.zhauniarovich@unitn.it

Maqsood Ahmad
University of Trento
Trento, Italy
maqsood.ahmad@unitn.it

Olga Gadyatskaya
SnT, University of Luxembourg
Luxembourg
olga.gadyatskaya@uni.lu

Bruno Crispo
University of Trento, Italy
DistriNet, KU Leuven, Belgium
bruno.crispo@unitn.it

Fabio Massacci
University of Trento
Trento, Italy
fabio.massacci@unitn.it

Abstract

Static analysis of Android applications can be hindered by the presence of the popular dynamic code update techniques: dynamic class loading and reflection. Recent Android malware samples do actually use these mechanisms to conceal their malicious behavior from static analyzers. These techniques defuse even the most recent static analyzers (e.g., [12, 21, 31]) that usually operate under the "closed world" assumption (the targets of reflective calls can be resolved at analysis time; only classes reachable from the class path at analysis time are used at runtime). Our proposed solution allows existing static analyzers to remove this assumption. This is achieved by combining static and dynamic analysis of applications in order to reveal the hidden/updated behavior and extend static analysis results with this information. This paper presents design, implementation and preliminary evaluation results of our solution called StaDynA.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-throughs, Tracing*

Keywords

Android; Dynamic Code Updates; Security Analysis

1. INTRODUCTION

Mobile applications (apps for short) are complex programs that offer sophisticated user experiences by exploiting the whole spectrum of dynamic code update features provided by the Android platform.

Yet, these features (reflection and dynamic class loading) combined with the common practices adopted by mobile app developers make the static analysis of mobile apps a challenging task. This is particularly daunting when static analysis is used in order to check the security of mobile applications (e.g., to detect the presence of malicious behavior). Indeed, Rastogi et al. [40] mention reflection among the techniques that make most of the current static analysis tools unable to detect malicious code. Additionally, static analysis is hindered by the code that evolves dynamically, because some parts of the code are impossible to discover or to analyze at installation time as they appear only at runtime. As a matter of fact, existing state of the art static analyzers for mobile applications (e.g., [12, 21, 31]) assume that the code base does not change dynamically and the targets of reflection calls can be discovered in advance. This is a clear simplification of what happens in the real world, where many apps rely on code base updated at runtime.

Wang et al. [43] demonstrate the difficulty to certify apps written by a malicious developer. They developed a proof of concept malicious iOS app that passed successfully the review process on Apple's App Store. The code submitted for review was benign, yet the app was able to update itself on the device in order to introduce malicious control flows and to perform illicit tasks (such as attacking other apps and exploiting kernel vulnerabilities). Similar proof of concept apps, which were able to bypass the Google Bouncer[1] check using dynamic code update features, were also developed for the Android platform [38].

At the same time, previous approaches that enhanced static analyzers of Java code in the presence of dynamic code update techniques (e.g., [17]) cannot be directly applied to Android due to the differences in the platforms (in Android, load-time instrumentation of classes is not available). Moreover, offline instrumentation also cannot solve the problem because this approach breaks the application signature, while some apps check it at runtime. If the signature does not correspond to some hardcoded value they may refuse to work. In case of malicious apps this check may be used to conceal illicit behaviour.

In this paper we present StaDynA, a system supporting security app analysis in the presence of dynamic code update features. Our main contributions can be summarized as follows:

[1]A system that checks applications uploaded by developers to Google Play for malicious functionality.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CODASPY'15, March 2–4, 2015, San Antonio, Texas, USA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-3191-3/15/03 ...$15.00.
http://dx.doi.org/10.1145/2699026.2699105.

- We analyzed a large set of apps (downloaded from Google Play and third-party markets) and malware samples. Our findings show that extensive amount of Android apps relies on dynamic code update features.
- We designed and implemented StaDynA – a system that interleaves static and dynamic analysis in order to reveal the hidden/updated behavior. StaDynA downloads and makes available for analysis the code loaded dynamically, and is able to resolve the targets of reflective calls complementing app's method call graph with the obtained information. Thus, StaDynA can be used in conjunction with other static analyzers to make their analysis more precise.
- We release our tool as open-source[2] to drive the research in this direction.
- We evaluated StaDynA on a set of real applications. We report that StaDynA is useful in uncovering dangerous functionality not present (or not visible to static analyzers) in the initial distribution of the app.

The rest of the paper is organized as follows. §2 presents the results of our analysis of dynamic code update feature usage in Android apps. §3 provides a background on dynamic class loading and reflection in Android. §4 gives a high-level description of StaDynA, while §6 covers the implementation details. §5 presents our approach to build method call graphs and visualise them. §7 reports on the evaluation of StaDynA on real apps. §8 discusses the limitations of the current implementation, and envisages the future work. §9 overviews the related work, and §10 concludes.

2. ANALYSIS OF DYNAMIC CODE UPDATE FEATURES IN ANDROID APPS

To understand how significant is the use of reflection and dynamic class loading (DCL) in Android apps we performed a study of 13,863 packages from Google Play [10] (the official market maintained by Google), and 14,283 apps from several third-party markets gathered in July 2013, along with 1260 malware samples from [51]. Notice that for reflection cases we consider calls that influence the app method call graph (MCG), i.e., method invocation (`invoke`) and object creation (`newInstance`) functions, and do not study other reflection API capabilities like field modification (because they do not influence the MCG used for analysis in our system).

The aggregated results of the analysis with our modified version[3] of AndroGuard [1] are shown in Table 1. It is evident that dynamic code update features are widely used by application developers.

On Google Play we downloaded approximately 500 top free applications from each category. The analysis shows that on average 18.5% of dissected apps in Google Play contain DCL and 88% use reflection. On average, apps with DCL contain 1 DCL call and apps with reflection incorporate around 22 reflective calls. The categories *"BUSINESS"*, *"SHOPPING"* and *"TRAVEL_AND_LOCAL"* show minimal DCL rates (at most 10% of apps use DCL). The most "dynamic" category is *"GAME"*: 38.3% of applications in this category use DCL[4].

Table 1: Usage of DCL and Reflection in Applications

Markets	**Total**	**DCL used by**		**Refl. used by**	
	Apps	**Apps**	**%**	**Apps**	**%**
Google Play	13863	2573	18.5%	12233	88.2%
Androidbest	1655	35	2.1%	1088	65.7%
Androiddrawer	2677	379	14.1%	2596	96.9%
Androidlife	1677	117	6.9%	1368	81.5%
Anruan	4230	162	3.8%	2868	67.8%
Appsapk	2664	112	4.2%	1907	71.5%
F-droid	1380	11	0.07%	792	52.8%
Malware	1260	251	19.9%	1025	81.3%
Total	**29406**	**3640**	**12.3%**	**23877**	**81.1%**

We further downloaded apps from 6 third-party markets, namely, *androidbest* [4], *androiddrawer* [5], *androidlife* [6], *anruan* [7], *appsapk* [8] and *f-droid* [9]. The first 5 markets distribute only provided apk files, while the latter (*f-droid*) along with the final packages also provides links to the source code of the apps. The lowest fraction of applications with DCL calls were observed on the *f-droid* market that contains only open-source apps. In terms of individual usage, the average number of reflection calls is around 19 per app package across all third-party markets (with *f-droid* exhibiting again the lowest number of reflection calls at around 14).

Besides the analysis of benign applications, we studied malware samples provided in [51]. The average percentage of DCL usage across all malware samples is 19.9%, whereas 81% of all samples use reflection. However, this dataset is old, and DCL usage rates in more recent malware applications are expected to be significantly higher [38] because this functionality is used to conceal malicious payloads [26] from static and dynamic analyzers like Google Bouncer.

Listing 1 is a code snippet of the `AnserverBot` Trojan [50], which illustrates how reflection and DCL are used to thwart static analyzers from detection of malicious functionality. Line 16 shows an example of a dynamic class loading call in Android using the `DexClassLoader` class. The name of the file from which the code is loaded is computed at runtime in Line 8. Line 26 exhibits how to create an object of the loaded class using reflective call of the default constructor. Line 28 demonstrates a method invocation through reflection; the name of the invoked method is passed as a parameter and, thus, may not be available for static analysis.

3. REFLECTION AND DYNAMIC CLASS LOADING IN ANDROID

In order to understand the design of StaDynA, we first provide some background information on dynamic class loading and reflection implementation in Android. Notice that while in this paper we consider the Dalvik Virtual Machine (the Dalvik VM or DVM), the same functionality, i.e., DCL and reflection, is also present in the new Android runtime called ART that replaces DVM in the recent platform versions.

[2]`https://github.com/zyrikby/StaDynA`

[3]We found out that AndroGuard does not discover all possible cases of reflection and DCL.

[4]Mobile games can be very sophisticated and include realistic physics and a lot of graphics. Thus, developers often develop the original app as an installer that dynamically fetches additional code during the first run.

```java
[com.sec.android.providers.drm.Doctype]
public static Object b(File pFile, String pStr1,
    String pStr2, Object[] pArrOfObj) {
  String s3;
  if (pFile == null) {
    String s1 = a.getFilesDir().getAbsolutePath();
    //get the name of the file to be loaded
    //9CkOrC32uI327WBD7n__ -> /anserverb.db
    String s2 = Xmlns.d("9CkOrC32uI327WBD7n__");
    s3 = s1.concat(s2);
  }
  for (File locFile = new File(s3); ;locFile =
    pFile) {
    String s4 = locFile.getAbsolutePath();
    String s5 = a.getFilesDir().getAbsolutePath();
    ClassLoader locClassLoader = a.getClassLoader
    ().getParent();
    //get the class specified by "pStr1" from
    anserverb.db
    Class locCls = new DexClassLoader(s4, s5, null
    , locClassLoader).loadClass(pStr1);
    Class[] arrOfCls = new Class[5];
    arrOfCls[0] = Context.class;
    arrOfCls[1] = Intent.class;
    arrOfCls[2] = BroadcastReceiver.class;
    arrOfCls[3] = FileDescriptor.class;
    arrOfCls[4] = String.class;
    //get the method specified by "pStr2"
    Method locMtd = locCls.getMethod(pStr2,
    arrOfCls);
    //create new instance of the class
    Object locObj = locCls.newInstance();
    //invoke the method through reflection
    return locMtd.invoke(locObj, pArrOfObj);
  }
}
```

Listing 1: DCL and Reflection Usage in `AnserverBot`

3.1 Reflection

The ability of a program to manipulate as data something representing the state of the program during its own execution is called *reflection* [16]. Although Android is based on the Dalvik VM, the reflection API is almost the same as that of Java (with only several subtle differences). This API is used to access class information at runtime, create objects, invoke class methods, change the modifiers and the values of data field members [44]. More precisely, in Android the reflection API is used for the following purposes:

Hidden API method invocation. The developers of the Android OS may mark some methods as hidden (using `@hide` tag). In this case, the declaration and description of these methods does not appear in the SDK library and, thus, is not available for application developers. At the same time, app developers may use the reflection API to invoke these methods at runtime.

Access to the private API methods and fields. During compilation, the compiler ensures that the rules of access to fields and methods according to the specified modifiers hold. Yet, using the reflection API it is possible to manipulate with modifiers and, therefore, gain access to private members of a class at runtime.

Conversion from JSON and XML representation to Java objects. The reflection API is heavily used to generate automatically JSON and XML representation from Java objects and vice versa.

Backward compatibility. It is advised to use reflection to make an app backward compatible with the previous versions of the Android SDK. In this case, reflection is exploited either to call the API methods, which have been marked as hidden in the previous versions of the Android SDK, or to detect if the required SDK classes and methods are present in the current framework version.

Plug-in and external library support. In order to extend the functionality of an application, the reflection API may be used to call plug-ins or external library methods provided at runtime using dynamic class loading functionality.

3.2 Dynamic Class Loading

The Dalvik VM allows a developer to load at runtime code obtained from alternative locations, such as the internal storage or over the network [19]. This functionality is usually used to:

Overcome the 64K method reference limit. Maximum number of method references in a dex file is 64K, but additional methods can be put in a separate dex file and loaded dynamically.

Extend app functionality at runtime. An app can provide stubs that process events using the pieces of code written by different developers. These pieces of code are called plug-ins, and DCL is widely used here to load the plug-in code into the memory.

Although Android allows developers to load and execute code dynamically, Google strongly recommends to avoid using this feature [3]. These recommendations are based on the fact that DVM does not provide a secure environment for the code supplied dynamically. Thus, this code has the same permissions as the app that loads this code. Moreover, DVM does not isolate code from the underlying operating systems capabilities and, thus, dynamically loaded code can operate with native libraries without any constraints [3]. These are crucial differences of the Android security architecture comparing with Java's one.

Class loaders are responsible for controlling the loading of classes into DVM. The process of loading classes in Android resembles the one implemented in Java [34, 41]. As in JVM, Dalvik VM also has the *bootstrap* class loader responsible for loading core API classes. The *system* class loader is liable for loading application classes.

Similarly to Java, in Android class loaders form a tree. To organize this structure, each class loader holds a reference to its parent. The *bootstrap* class loader is the root of this tree; it has a `null` reference to its parent. An app may also define additional class loaders. In Android all particular class loaders are derived from `java.lang.ClassLoader` (possibly indirectly). Android provides several concrete implementations of this class, `PathClassLoader` and `DexClassLoader` being the most widely used ones.

4. AN OVERVIEW OF STADYNA

The architecture of StaDynA presented in Figure 1 comprises two logical components: a server and a client.

The static analysis of an application is performed on the server. In this respect, StaDynA allows an analyst to easily plug in and use any static analyzer in its architecture. The static analyzer on the server builds the initial *method call graph* (MCG) of the app, integrates the results of the dynamic analysis coming from the client, and stores the results of the scrutiny. The client part of StaDynA is a modified Android operating system, hosted either on a real device or an emulator. The client runs the application whenever the dynamic analysis is required.

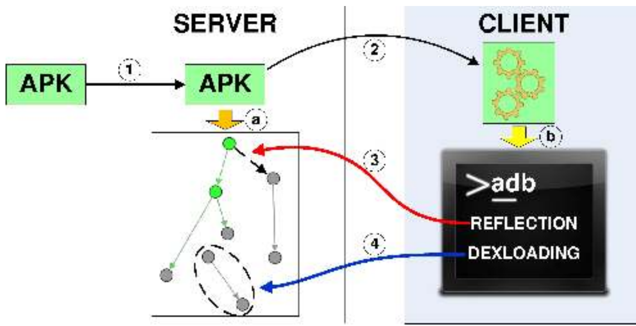

Figure 1: System Overview

In action, our system interleaves the execution of the static and dynamic analysis phases. However, to simplify the presentation, we describe them sequentially.

Preliminary analysis.

The server statically analyzes an app package and builds a MCG of that application (see Step *a* in Figure 1; solid arcs denote edges resolved statically). Dynamically loaded code cannot be analyzed during this phase and, thus, the corresponding nodes and edges are not present in the MCG. Further, the names of methods called through reflection may also not be inferred if they are represented as encrypted strings or generated dynamically. Still, a static analyzer can effectively detect the points in the MCG where the functionality of an application may be extended at runtime. Indeed, the usage of reflection and DCL requires to use specific API calls provided by the Android platform. The server detects these calls during the static analysis phase by searching for methods where DCL and reflection API calls are performed. We call these methods *methods of interest (MOI)*.

Dynamic execution.

If MOIs have been detected in the application, StaDynA installs the app on the client (Step *2*) and launches the dynamic analysis. The dynamic phase is exercised to complement the MCG of the app and to access the code loaded dynamically. The dynamic analysis is performed on a device (or an emulator) with a modified Android OS. The added modifications log all events when the app executes a call using reflection, or when additional code is loaded dynamically. Along with these events, the client also supplies some additional information, e.g., in case of a reflection call, the information about the called function and the stack trace (it contains the ordered list of method calls, starting from the most recent ones) is added. In case of DCL call, the path to the code file and the stack trace are supplied. The information collected by the client is passed back to the server side (Step *3*).

Analysis consolidation.

The server performs an analysis of the obtained information. In case a reflection call happens, the server complements the MCG of the app with a new edge (in Figure 1 it is represented by a dashed arc). This edge connects the node of the method that initiated the call through reflection (the node at the beginning) with the one corresponding to the called function (the node at the end).

When DCL is triggered the client infers which file was used to get the code. Using this evidence, the server downloads the file (Step *4*) containing the code, and performs the static analysis on it. The MCG of the app is then updated with the obtained information (see the part of the MCG in the dashed oval in Figure 1). Additionally, for each downloaded file the server analyzes whether it contains other MOIs. If it does, the list of the MOIs for the application is updated. This allows StaDynA to unroll nested MOIs. The stack trace data both for the reflection and DCL cases is used to detect which MOI initiated the call.

Marking suspicious behavior.

In Android, some API calls are guarded by permissions. Since APIs protected by the permissions could potentially harm the system or compromise user's data, the permissions must be requested in the `AndroidManifest.xml` file. However, there is no actual check which permissions are required to execute the written code and sometimes developers request more permissions than they actually use. In this case, those apps are called overprivileged. Many researchers, e.g., Bartel et al. [14], identified that malware, adware and spyware exploit additional permissions to get access to security sensitive resources at runtime.

Based on these considerations, we classify the following app behavior patterns as *suspicious*:

- An application dynamically loads the code that contains API functions protected with permissions. Indeed, malware may use this approach to evade detection by static analyzers, as the security-sensitive code is loaded dynamically.

- An application calls through reflection an API method protected with a *dangerous* permission[5]. This functionality can be used, for instance, to send malicious SMS, which cannot be detected by static analysis tools because the name of the SMS sending function is encrypted and decrypted only at runtime.

Detection of these suspicious patterns has been added to our tool. StaDynA raises a warning if such patterns occur during the analysis. Section 7 shows that indeed malware samples do expose such suspicious patterns.

5. METHOD CALL GRAPH

Method call graphs (or function call graphs) identify the caller-callee relationships for program methods. These structural representations of programs are widely used for different purposes. In the scope of Android, method call graphs are used, e.g., to detect malware [27, 29, 33], to identify potential privacy leaks in applications [23, 28, 49], to find vulnerabilities [42] and execution paths for automatic testing [48].

StaDynA extends the initial MCG generated with a traditional static analyzer with the information detected at runtime. Thus, if an application exposes dynamic behavior all mentioned approaches can benefit from the expanded MCG obtained with StaDynA.

[5]Google classifies as "dangerous" permissions with higher-risk level that guard access to private user data or device controls [2].

Example.

To visualize the capabilities of StaDynA and the process of method call graph expansion, we show the evolution on the example of a *demo_app*. Figure 2a shows the MCG of the app obtained with the AndroGuard static analyzer [1]. Figure 2b shows the one gained with StaDynA before dynamic execution phase, and Figure 2c presents it with dynamic execution phase. The *demo_app* dynamically loads some code from an external jar file at runtime and calls the loaded methods through reflection.

Figure 2a illustrates that AndroGuard identifies only the presence of ordinary methods and DCL calls (Ellipse 1) but no further analysis is done about those. Yet, Figure 2b shows that after preliminary analysis StaDynA selects 3 paths, which are surrounded by dashed ellipses. Ellipse 1 shows that a MOI (the dark grey node) invokes a constructor (the dark green node) through reflection. Similarly, Ellipse 2 displays a method invocation through reflection. Ellipse 3 depicts that a DCL call (the red node) is performed in a MOI (the dark grey node).

During the dynamic analysis StaDynA adds the edges that are outlined by Ellipses 4-7 (see Figure 2c). These ellipses show the cases when the MOIs are resolved and corresponding nodes and edges are added to the MCG. Ellipse 4 shows that as a result of a DCL call (the red node) a new code file has been loaded (the pink node). Ellipse 7 shows that a class constructor (the grey node) is called through reflection. Ellipse 5 shows a method invoked through reflection. This method contains an API call protected by the Android permission indicated by the blue node in Ellipse 6. There are also nodes and edges that appear as a result of the analysis of the code file (the pink node) loaded dynamically. These nodes and edges are connected with the rest of the graph through the reflection *new instance* call (see Ellipse 7).

Ellipses 2, 3, 8, 9 show other types of connections possible among nodes in a MCG obtained with our tool. Ellipse 2 shows the connection between the class and its constructor, Ellipse 3 shows an ordinary relation between two methods, Ellipse 9 connects the static initialization block and the class, and Ellipse 8 shows that the method is called from the static initialization block.

Each node type is assigned with a set of attributes, not shown in the figures. The analysis of values of these attributes can facilitate dissection of Android applications accompanied by the expanded method call graph. For instance, each method node is assigned with attributes, which correspond to a class name, a method name and a signature of this method. A permission node is assigned with a permission level along with the information about the API call that it protects.

6. IMPLEMENTATION

This section provides the implementation details of some key aspects of StaDynA. The workflow of our system operation is shown in Figure 3. App analysis starts at the server side. All occurrences of reflection and DCL methods are identified in the code of the application under analysis. In case neither of them is found, StaDynA builds a MCG of the app and exits. Otherwise, it starts the dynamic analysis on a device with the modified Android OS, which constitutes the client part of StaDynA.

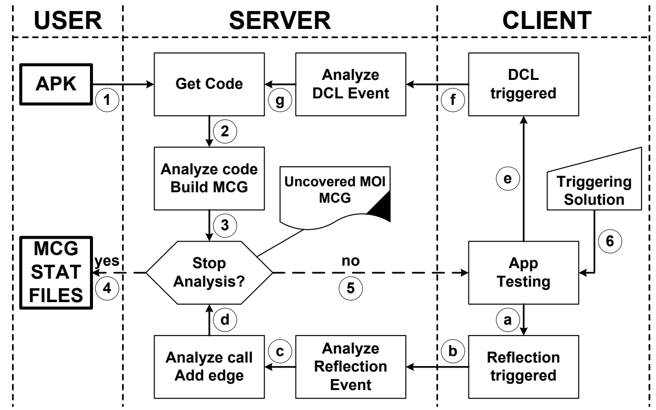

Figure 3: The StaDynA Workflow

6.1 The server

The server side of StaDynA is a Python program that interacts with a static analysis tool. Currently, StaDynA uses AndroGuard [1] as a static analyzer. AndroGuard represents compiled Android code as a set of Python objects that can be manipulated and analyzed. However, StaDynA can work with any static analysis tool that is able to analyze `apk` and `dex` files. To improve suspicious behavior detection we substituted the permission map embedded in AndroGuard (built for Android 2.2 in [25]) with the one generated by PScout [13] for Android 4.1.2.

Algorithm 1 App Analysis Main Function Algorithm

```
1:  function PERFORM_ANALYSIS(inputApkPath, resultsDirPath)
2:      makeAnalysis(inputApkPath)
3:      // Check if there are MOI
4:      if !containsMethodsToAnalyze() then
5:          performInfoSave(resultsDirPath)
6:          return
7:      end if
8:      dev ← getDeviceForAnalysis()
9:      package_name ← get_package_name(inputApkPath)
10:     dev.install_package(inputApkPath)
11:     uid ← dev.get_package_uid(package_name)
12:     messages ← dev.getLogcatMessages(uid)
13:     loop
14:         msg ← dequeue(messages)
15:         // analyzeStadynaMsg contains a switch statement
16:         // that selects a corresponding processing routine
17:         // shown in Algorithms 2 and 3 based on the msg type
18:         analyzeStadynaMsg(msg)
19:
20:         // Quit if a user finishes analysis
21:         if finishAnalysis then
22:             performInfoSave(resultsDirPath)
23:             return
24:         end if
25:     end loop
26: end function
```

The pseudo-code of the main server function is presented in Algorithm 1. The server starts the analysis of the provided app by extracting the `classes.dex` file (see Step *1*, *2* and *3* in Figure 3; Line 2 in Algorithm 1), and then dissects the extracted code. During this step StaDynA searches in the code all occurrences of reflection and DCL calls. The list of searched patterns for these API calls is presented in Table 2.

If MOIs are found, StaDynA selects a device (a real phone or an emulator) to perform the dynamic analysis on

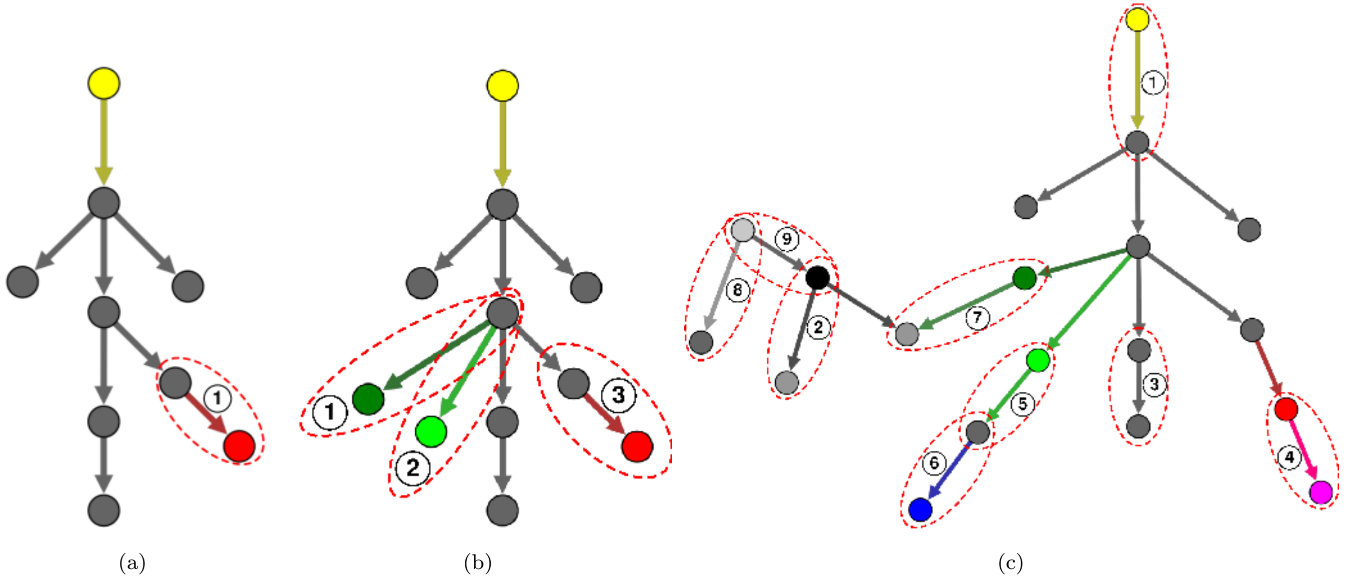

Figure 2: MCG of *demo_app* Obtained with a) AndroGuard b) StaDynA after Preliminary Analysis c) StaDynA after Dynamic Analysis Phase

Table 2: The List of Searched Patterns

Class	Method	Prot.
Dynamic class loading		
Ldalvik/system/PathClassLoader;	*< init >*	.
Ldalvik/system/DexClassLoader;	*< init >*	.
Ldalvik/system/DexFile;	*< init >*	.
Ldalvik/system/DexFile;	*loadDex*	.
Class instance creation through reflection		
Ljava/lang/Class;	*newInstance*	.
Ljava/lang/reflect/Constructor;	*newInstance*	.
Method invocation through reflection		
Ljava/lang/reflect/Method;	*invoke*	.

(Line 8) and installs the app under analysis (Line 10) onto the client (Step *5* in Fig. 3). After that the server obtains the UID of the installed package (Line 11) and starts a loop (Lines 13-25) that analyzes one by one messages (Line 12) obtained using the *logcat* utility from the *main* log file of the Android system. Basically, each obtained message is represented in the JSON format and contains values for the following fields: *UID* (required), *operation* (required), *stack* (required), *class* (optional), *method* (optional), *proto* (optional), *source* (optional), *output* (optional). The value of the `UID` field is used to select the messages produced by the analyzed app. If the user stops the analysis, StaDynA saves the results and finishes its execution.

The function **`analyzeStadynaMsg`** (Line 18) analyzes the selected StaDynA messages obtained from the client. It extracts the value of the `operation` field and based on this value selects the appropriate routine to analyze the message.

The routines for the reflection messages analysis are similar, so we consider them on the example when operation corresponds to *reflection invoke*. The algorithm for analysis of the *reflection invoke* messages is shown in Algorithm 2[6]. Lines 2 - 4 extracts the method name along with its class name and the prototype, which has been called through reflection. Line 5 gets the stack from the message. Line 7 searches for the first *reflection invoke* occurrence in the stack. The next stack entry corresponds to the method that has performed the reflection call **`invSrcFrStack`** (Line 9). Then in the loop StaDynA compares this method with the list of MOIs extracted from the application executables (Lines 10 - 20). If the method is found StaDynA complements the MCG with the obtained information (Line 15), and deletes it from the list of uncovered invoke MOIs (Line 17). Otherwise, it adds this method to the list of vague methods (Line 21). This information is later analyzed to see why the method calling reflection was not found in the application executable during the static analysis phase.

The processing function for the DCL messages is slightly different (see Algorithm 3). From the message received from the client the server extracts the source path of the file containing the code loaded dynamically (Line 2). Using this information, StaDynA downloads the file locally (Line 4), and processes it (Line 5). This process includes computation of the file hash and copying the file into the results folder with a new filename, which includes the computed hash. The file hash allows us to check whether the file has been already loaded and avoid analysis of already checked code. Otherwise, the code analysis for MOIs is performed for the loaded code (Line 15). Function **`getDLPathFrStack`** (Line 6) searches for a pair of a DCL call and a MOI in the stack corresponding to the one extracted from the app executable. If this pair is found, then it is removed from the list of uncovered DCL calls (Line 11). Otherwise, StaDynA adds the information about the dynamic class loading call into the list of vague calls (Line 19).

[6]The algorithm for analysis of *reflection newInstance* messages is very similar so we do not show it.

Algorithm 2 Analysis of the Reflection Invoke Message

```
1: function processReflInvokeMsg(message)
2:     cls ← message.get(JSON_CLASS)
3:     method ← message.get(JSON_METHOD)
4:     prototype ← message.get(JSON_PROTO)
5:     stack ← message.get(JSON_STACK)
6:     invDstFrCl ← (class, method, prototype)
7:     invPosInStack ← findFirstInvokePos(stack)
8:     thrMtd ← stack[invPosInStack]
9:     invSrcFrStack ← stack[invPosInStack + 1]
10:    for all invPathFrSrcs ∈ sources_invoke do
11:        invSrcFrSrcs ← invPathFrSrcs[0]
12:        if invSrcFrSrcs ≠ invSrcFrStack then
13:            continue
14:        end if
15:        addInvPathToMCG(invSrcFrSrcs, thrMtd, invDstFrCl)
16:        if invPathFrSrcs ∈ uncovered_invoke then
17:            uncovered_invoke.remove(invPathFrSrcs)
18:        end if
19:        return
20:    end for
21:    addVagueInvoke(thrMtd, invDstFrCl, stack)
22: end function
```

Algorithm 3 Analysis of the DCL Message

```
1: function processDexLoadMsg(message)
2:     source ← message.get(JSON_DEX_SOURCE)
3:     stack ← message.get(JSON_STACK)
4:     newFile ← dev.get_file(source)
5:     newFilePath ← processNewFile(newFile)
6:     dlPathFrStack = getDLPathFrStack(stack)
7:     if dlPathFrStack then
8:         srcFrStack ← dlPathFrStack[0]
9:         thrMtd ← dlPathFrStack[1]
10:        if dlPathFrStack ∈ uncovered_dexload then
11:            uncovered_dexload.remove(dlPathFrStack)
12:        end if
13:        addDLPathToMCG(srcFrStack, thrMtd, newFilePath)
14:        if !fileAnalyzed(newFilePath) then
15:            makeAnalysis(newFilePath)
16:        end if
17:        return
18:    end if
19:    addVagueDL(newFilePath, stack)
20: end function
```

Notice that the presented algorithms are simplified versions of the ones actually implemented in the server part. For instance, in a real application it is possible that the same MOI acts like a proxy used to call different targets (e.g., the same method could be used to load different code files). The real algorithms implemented in StaDynA are able to process these cases.

6.2 The client

The client side can run either on a real device or on an emulator. Using the emulator is more convenient because one can run the client and server on the same machine. The main drawback is that currently the Android emulator is quite slow. Moreover, mobile applications may suppress some functionality if they detect they are running in an emulated environment. With these limitations in mind, we implemented and tested our client on a real device. However, the code is not device-dependent so it can be easily ported to an emulator or another device.

To obtain the information required for analysis of reflection and DCL usage, we have modified the `DVM` and `libcore` components. To obtain the information related to DCL we added a hook to the method `openDexFile` of the `DexFile` class. This method is called when a new file with the code is opened. It gets three parameters as an input, where `sourceName` is of our interest. The added code forms a *JSON* message that contains the path to the file, from which the code is loaded (`sourceName`). Along with this information, the stack trace data and the *UID* of the process are also added into the message, which is then printed out to the *main* log file of Android.

To get the information about method invocation through reflection, a hook was placed into the `invoke` function of the `Method` class. Each `Method` object has `declaringClass`, `name` and `parameterTypes` member fields, which represent class name, method name and prototype of the invoked method respectively. This information along with the stack trace is put into the StaDynA message. Similarly, to log the information about new class creation through reflection, we put our hooks into the `newInstance` method of the `Class` and `Constructor` classes.

Each StaDynA message contains the stack trace information. Stack trace is a sequence of method calls performed in the current thread starting from the most recent ones. The information from a stack trace is usually used to find the origin of an exception in a program. In our case, the stack trace information is used to detect the MOI, which calls the reflection or DCL methods. In essence, a stack trace is an array of stack trace elements. Each stack trace element contains information about the class name, the method name and the line number of the method call in the source code. Unfortunately, using only this information it is not possible to uniquely identify the MOI, because we do not have access to the source code of the application. Moreover, due to function overloading it is possible to have several methods in a class with the same name. To overcome this limitation we modified the`StackTraceElement` class so that it can store the information about the method prototype. Method name and its prototype allow us to uniquely identify a method in a class.

A StaDynA message has a header and a body. To distinguish StaDynA messages from other log messages we add a special marker to the header. The second part of the message header is the part number. Currently, there is a limit on the length of the Android log entries specified by the constant `LOGGER_ENTRY_MAX_PAYLOAD`. To overcome this problem, we added the functionality to the client that allows it to split a message into several parts. The server takes care of assembling the original message.

7. EVALUATION

This section describes our application test suite and reports on the results of our experiments. In order to evaluate StaDynA we tested it on real applications, both benign and malicious. The server runs on a machine with 2.5 GHz Intel Core i5 processor and 4 GB DDR3 memory. The client is a Google Nexus S smartphone with the modified Android OS version 4.1.2_r2 connected to the server using a standard USB cable.

The evaluation test suite consists of a set of 5 benign and 5 malicious applications. The benign applications were selected based on their popularity and the presence of MOIs in the code. The malware samples were selected based on the study presented in Section 2 from the families exhibiting DCL as a part of malicious behavior. We also added two malware samples (`FakeNotify.B` and `SMSSend`) to our test suite based on the reports of antivirus companies [24, 37].

Table 4: Evaluation: MCG Expansion

Apps	Nodes		Edges		Perm. Nodes	
	Init.	Final	Init.	Final	Init.	Final
Benign Applications						
FlappyBird	8592	8614	11014	11031	9	9
Norton AV	42886	55372	65960	85665	63	81
Avast AV	31317	32363	43554	44956	22	25
Viber	42536	46312	60078	65627	67	71
ImageView	5708	5713	6488	6496	7	7
Malicious Applications						
FakeNotify.B	148	171	137	191	1	2
AnserverBot	1006	1614	1138	2093	12	23
BaseBridge	1172	1780	1364	2333	14	25
DroidKungFu4	1550	21168	1779	23589	26	250
SMSSend	431	537	826	951	0	3

To evaluate StaDynA, the selected apps were manually inspected in order to trigger execution of MOIs. We also experimented with automatic triggering using the monkey tool [11]. This tool generates pseudo-random streams of user events and executes them on a device. Unfortunately, due to its random nature this tool was not useful for our experiments because StaDynA requires triggering of precise methods which contain reflection and DCL calls. To facilitate manual analysis, we extend our tool with the functionality that reports which MOIs have not been yet triggered. Observing this list an analyst may predict what actions will cause the execution of the uncovered MOIs. Here we report the results obtained using manual triggering.

Table 3 shows the numbers of detected MOIs for each operation ("Refl. Invoke", "Refl. NewInstance" and "DCL"). Each operation column has 3 subcolumns that present the number of MOIs in the initial application executable ("Init."), the number of detected MOIs after the analysis ("Final"), and the number of calls we managed to trigger during the analysis ("Triggered"). As StaDynA also analyzes the dynamically loaded code for MOIs, the numbers in the "Final" columns are usually higher than in the "Init." ones. The ratio between the numbers in the "Triggered" and "Final" columns can be considered as a coverage metric for StaDynA for every operation. Indeed, achieving 100% for this metric would mean that all MOIs were triggered at least once.

As the result of MOIs triggering, the MCG of the applications grows. Table 4 characterizes the effect of MCG expansion after StaDynA's analysis. MCG expansion is determined by two factors: a) StaDynA is able to analyze the code loaded dynamically and includes this information into the final MCG b) StaDynA can resolve the targets (unavailable in the initial graph) of reflective calls. The first subcolumn ("Init.") shows the number of nodes, edges and permission nodes in the initial MCG, while the second ("Final") presents the parameters of the MCGs obtained after analysis with StaDynA.

The column "Perm. Nodes" in Table 4 shows the number of detected API methods protected with dangerous permissions. Table 5 presents the analysis of the dangerous permission nodes discovered with the help of StaDynA. The column "Permissions" lists the names of the dangerous permissions required to run the code added by dynamic code updates features. A cross (**X**) in the column "New" shows the fact that the API calls protected with this permission were not discovered in the initial application executable. At the same time this permission is required to run the code

Table 5: Evaluation: Dangerous Permissions

App	Permissions	New
Benign Applications		
Norton AV	WRITE_SETTINGS	
	READ_PHONE_STATE	
	INTERNET	
	WRITE_SYNC_SETTINGS	X
	GET_TASKS	
Avast AV	INTERNET	
Viber	READ_PHONE_STATE BLUETOOTH	
	INTERNET	
Malware		
FakeNotify.B	SEND_SMS	X
AnserverBot	INTERNET	
	READ_PHONE_STATE	
BaseBridge	INTERNET	
	READ_PHONE_STATE	
DroidKungFu4	CHANGE_NETWORK_STATE	X
	ACCESS_COARSE_LOCATION	
	BLUETOOTH	X
	INTERNET	
	BLUETOOTH_ADMIN	X
	WRITE_SETTINGS	X
	SET_TIME_ZONE	X
	WRITE_SYNC_SETTINGS	X
	READ_PHONE_STATE	
	CHANGE_WIFI_STATE	X
	MODIFY_AUDIO_SETTINGS	X
	MOUNT_UNMOUNT_FILESYSTEMS	X
SMSSend	READ_PHONE_STATE	X
	SEND_SMS	X

added by dynamic code updates features. These applications (with the cross in the column "New") will be considered as overprivileged by the tools [13, 14, 25], although in general, the apps do not belong to this category (because they use these permissions to run the dynamic code).

Results on benign apps.

ImageView does not contain the dynamic class loading functionality, thus its MCG was not expanded significantly by StaDynA. A popular game FlappyBird contains 1 *DCL* call, which was successfully uncovered during the analysis, and several instances of *Reflection Invoke* and *Reflection NewInstance*. However, the expansion of MCG produced by StaDynA was also relatively small (22 new nodes and 17 new edges). More complex applications like the mobile antiviruses Norton and Avast and the popular messenger Viber demonstrated significant expansion of their MCGs: more than 1000 of new nodes and edges were discovered by StaDynA for each app.

Norton AV, Avast AV and Viber also demonstrated suspicious behavior: they dynamically added code that invokes dangerous Android APIs protected by permissions. Notice that one of new API calls added by Norton AV (protected by the WRITE_SYNC_SETTINGS permission) was not even present in the original MCG. Thus, Norton AV would have been flagged as an overprivileged app (the one that requests more permissions than it actually uses in the code) by the tools [13, 14, 25].

Results on malware samples.

FakeNotify.B and SMSSend do not contain DCL calls, and new elements of their MCGs discovered by StaDynA appeared only as a result of reflection calls. Uncovered parts of MCGs of these apps are relatively small (while still revealing

Table 3: Evaluation: Number of MOIs for Each Operation

Apps	Refl. Invoke			Refl. NewInstance			DCL		
	Init.	Final	Triggered	Init.	Final	Triggered	Init.	Final	Triggered
Benign Applications									
FlappyBird	10	11	6	6	6	0	1	1	1
Norton AV	18	137	5	8	12	2	4	4	2
Avast AV	42	42	6	19	19	5	1	1	1
Viber	101	107	26	21	47	14	2	2	1
ImageView	6	6	5	2	2	2	0	0	0
Malicious Applications									
FakeNotify.B	68	68	68	9	9	9	0	0	0
AnserverBot	4	4	1	4	5	2	5	6	3
BaseBridge	5	5	1	2	3	2	2	3	3
DroidKungFu4	9	13	1	4	6	0	1	1	1
SMSSend	193	193	128	1	1	1	0	0	0

hidden suspicious functionality). More interesting results were demonstrated by StaDynA on AnserverBot, Basebridge4 and DroidKungFu43, where uncovered new parts of MCGs are comparable in size with the original statically produced graphs. In fact, the DroidKungFu43 code size exploded after dynamic class loading (an order of magnitude increase of the MCG size). This sample loaded the file settings.apk that contained approximately 13 times more nodes and edges than the original application.

The other two malware samples where DCL is present are from the AnserverBot and BaseBridge families. Both samples contain more than one instance of DCL. These samples both load two files with the names moduleconfig.jar and bootablemodule.jar. The former one contains no MOIs, whereas the latter contains *reflection invoke* and *DCL* calls. bootablemodule.jar then loads another file mainmodule.jar. This example shows how StaDynA unrolls nested calls.

In contrast to the benign apps, all evaluated malware samples exhibit suspicious functionality. This is an interesting result, as it shows that advanced malware indeed conceals its logic and reveals it only at runtime. E.g., SMSSend did not have any node labeled with a dangerous permission prior to the analysis. StaDynA has uncovered 4 such nodes (new nodes are protected with permissions READ_PHONE_STATE and SEND_SMS).

Our results show evidence that malware samples are more overprivileged (they contain more permission types required for the code loaded dynamically), so it is valid to identify the apps as suspicious if they are overprivileged. Yet, as benign apps can be overprivileged too, more research is required to understand if an application is benign or malicious, and StaDynA can be handy in exploration of this topic.

8. DISCUSSION

Our tool has space for future improvements. For StaDynA the coverage of MOIs (the ratio between the number of executed MOIs at least once and total number of discovered MOIs) is especially important. Currently, our system uses a manual approach to trigger MOI. Since we triggered the methods manually, StaDynA was not able to cover all MOIs in the apps because manual triggering is mostly GUI-based (it is challenging for a human analyst to produce a sufficient range of system events that might trigger all MOIs). As a way to improve StaDynA we plan to implement an automatic approach for triggering. As a first step in this direction we explored if the tools like *monkey* [11] can be handy. However, in our experiments we found out that pseudo-random events generated by the tool do not produce tolerable coverage values for MOIs. A possible approach to achieve satisfying values is to use systems like SmartDroid [48]. SmartDroid allows an expert to specify sensitive API methods required to be triggered. In case of StaDynA the sensitive API methods correspond to reflection and DCL calls. Other possible tools, which may be useful in developing fully automatic approach, are [15,39,45].

Another possible direction to reduce the amount of manual work is to resolve the targets of reflection calls statically at least those that are represented by constant strings [31]. The analysis performed in [25] has shown that it was possible to resolve automatically the targets of reflection calls in 59% of applications that used reflection. At the same time, the analysis was performed for the "closed world" scenario, which is not realistic, given that dynamic class loading is a popular technique for modern apps. Additionally, we can see that reflection is used more heavily today than in 2011 (88% of apps in our study versus 61% reported in [25]).

Usually, dynamic analysis allows an expert to explore only one execution path at a time. However, dynamic traces may differ depending on the context of the execution, e.g., some methods may contain calls invoked with parameters affecting the reflection call target. Therefore, another direction for improving StaDynA is to incorporate information obtained during different runs of analysis.

StaDynA has also other limitations. Its analysis is based on the UID of an application. However, it is possible in Android that several apps have the same UID. In this case, StaDynA will also collect the information produced by other apps with the same UID. At the same time, this information will not be used to complement MCG, but will be added to the category of vague calls that need to be later analyzed manually.

9. RELATED WORK

Being the most popular mobile OS, Android has won this position due to the openness of its ecosystem and the ease with which developers can publish apps on Google Play and third-party markets. Yet the openness comes at the price of large volumes of malware apps polluting the ecosystem. One approach to tackle security and privacy of mobile apps is to extend the security controls of the platform to detect misbehaving apps or to enforce the desired security policy [20,47]. Solutions following this approach, often require to modify the system image.

Another approach, more relevant to StaDynA, consists in the analysis of the mobile application code. Many static and dynamic analysis techniques have been proposed for Android. The ded system [23] re-targets Dalvik bytecode into Java class files that can be analyzed by the variety of tools developed for Java. In the original paper [23] the FortifySCA static analysis toolset was used for detecting vulnerabilities and dangerous functionality, like leaking the device IMEI. DroidAlarm [49] performs static detection of privilege-escalation vulnerabilities in apps by constructing paths in inter-procedural call graphs from a sensitive permission to a public interface accessible to other apps. StaDynA complements these static analysis techniques by completing inter-procedural call graphs.

Hu et al. proposed to explore functional call graphs (FCG) and rely on graph similarity metrics to detect malware based on known malware graph patterns [33]. Gascon et al. continue this research direction for Android with a technique to detect malware apps based on comparing FCGs that are mined with AndroGuard [27]. StaDynA can complement these techniques by providing more precise graphs required for analysis.

TaintDroid was among the first dynamic analysis tools for Android apps [22]; it allows to track propagation of information via the TaintDroid infrastructure-equipped smartphone software stack. Sources of sensitive information are typically the device sensors or private user information, and sinks are network interfaces; thus the main scope of TaintDroid is detection of privacy leaks. This approach is followed by DroidScope [45]. DroidScope allows to emulate app execution and trace the context at different levels of the Android software stack: at the native code level, at the Dalvik bytecode level, at the system API level, and at the combination of both native and Dalvik levels. While executing an app in DroidScope a security analyst can track events at different levels and instrument parameters of invoked methods to discover a malicious activity.

Dynamic analysis techniques are especially difficult to automate due to the need of emulating a comprehensive interactions of applications with the system and a user (UI interactions). Several approaches are proposed to automate the triggering of UI events, from random event generation [32] to more advanced approaches like AppsPlayground [39] and SmartDroid [48]. However, all of them still have many limitations on the type of events they can handle and the coverage.

Recently, Poeplau et al. [38] have identified the problem of dynamic code loading in Android apps. The authors selected possible vulnerable patterns of dynamic code loading and built a tool that can analyze Android apps for the found patterns. Moreover, they propose to use whitelists to prevent dynamic code loading that can potentially expose dangerous behavior. Whitelisting prevents unauthorized code from running. To get authorization the code must either signed [46] and its signature has to be included into a special list distributed by trusted authorities. However, as mentioned in the article [38], extraction of the dangerous behavior is a difficult problem by itself, especially when the protected API is called through reflection. In contrast, StaDynA aims not at preventing this loading (because a lot of legitimate apps use it and extra complications will not be welcomed by the developers) but at its analysis.

Reflection and Dynamic Class Loading in Java.

Gaps in the static analysis techniques in the presence of dynamic class loading, reflection and native code were previously studied for Java. For example, similarly to our approach, in [30] a pointer analysis (based on program call graphs) technique for the full Java language is extended by addressing dynamic class loading and reflection via an "on-line" analysis, when a call graph is built dynamically based on the program execution, and dynamic class loading, reflection and native code are treated in real time by modifying the pointer analysis constraints accordingly.

A run-time shape analysis for Java is investigated in [18]. Traditionally a shape analysis operates based on the call graph of a program, and it allows to conclude how the heap objects are linked to each other (e.g., if a variable can be accessed from several threads). Yet in Java the call graph produced from a program can be incomplete; and [18] suggests how to execute an incremental shape analysis when the call graph evolves dynamically. Our proposal does not involve a shape analysis, yet the ideas behind our proposal and [18] are similar.

Livshits, Whaley and Lam have studied the reflection analysis for Java [36]. They propose refinement for the static algorithms to infer more precise information on approximate targets of reflective calls, as well as to discover program points where user needs to provide a specification in order to resolve reflective targets.

Relevant to StaDynA is TamiFlex [17] that complements static analysis of Java programs in the presence of reflection and custom class loaders. Using the load-time Java instrumentation API TamiFlex modifies the original program to perform logging of class loading and reflection call events. This information is used to seed a tool that performs static analysis of the program having the information obtained during the dynamic analysis phase. This work differs from StaDynA in several aspects. First, TamiFlex uses a special Java API that is not available in Android. Second, although in Android it is possible to instrument an app before loading it on a device (offline instrumentation), some Android apps check the application signature in its code that is changed during the patching. Thus, for these applications the TamiFlex approach will not work in Android. Third, TamiFlex requires some debug information (the line number of the function call) to be present. In Android during the obfuscation phase this kind of information may be deleted from the final package. Therefore, the TamiFlex approach will not work, while StaDynA is able to process correctly this case due to the modifications we added to the Dalvik VM.

10. CONCLUSION

Today mobile applications make an extensive use of dynamic capabilities, namely reflection and dynamic class loading, available in the Android OS. Being adopted from Java, these techniques in Android incur an additional threat because the loaded code receives the same privileges as the loading one. Malicious apps can leverage these facilities to conceal their malicious behavior from analyzers.

In this paper we present StaDynA, a technique that interleaves static and dynamic analysis in order to scrutinize Android applications in the presence of reflection and dynamic class loading. Our approach makes it possible to expand the method call graph of an application by capturing additional modules loaded at runtime and additional paths

of execution concealed by reflection calls. In order to produce the expanded call graph StaDynA does not require modification of the application itself.

The results produced by StaDynA can then be fed to the state of the art analyzers in order to improve their precision (for instance, a reachability analysis will be more precise over the expanded MCG than over the original one). Thus, StaDynA may help malware analysts by increasing their ability to detect suspicious samples.

11. ACKNOWLEDGEMENTS

This work has been partially supported by the EU project CAPITAL. We would like to thank Martina Lindorfer from the Andrubis project [35] for the provided dataset of malicious applications.

12. REFERENCES

[1] AndroGuard: Reverse engineering, malware and goodware analysis of Android applications. Available Online. https://code.google.com/p/androguard/.

[2] Android - App Manifest - Permission http://developer.android.com/guide/topics/manifest/permission-element.html.

[3] Android Security Tips. Available Online. http://developer.android.com/training/articles/security-tips.html.

[4] AndroidBest – Android market. http://androidbest.ru/.

[5] AndroidDrawer – Android market. http://www.androiddrawer.com/.

[6] AndroidLife – Android market. http://androidlife.ru/.

[7] Anruan – Android market. http://www.anruan.com/.

[8] AppsApk – Android market. http://www.appsapk.com/.

[9] F-Droid – Android market. https://f-droid.org/.

[10] Google Play – Android official market. https://play.google.com/store/apps.

[11] UI/Application Exerciser Monkey. Available Online. http://developer.android.com/tools/help/monkey.html.

[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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 2014.

[13] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 217–228, 2012.

[14] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically Securing Permission-based Software by Reducing the Attack Surface: An Application to Android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 274–277, 2012.

[15] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1021–1036, August 2014.

[16] D. G. Bobrow, R. P. Gabriel, and J. L. White. Object-oriented programming. chapter CLOS in Context: The Shape of the Design Space, pages 29–61. MIT Press, 1993.

[17] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250, 2011.

[18] J. Bogda and A. Singh. Can a Shape Analysis Work at Run-time? In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, pages 2–2, 2001.

[19] F. Chung. Custom Class Loading in Dalvik. Available Online. http://android-developers.blogspot.it/2011/07/custom-class-loading-in-dalvik.html.

[20] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich. CRêPE: A System for Enforcing Fine-Grained Context-Related Policies on Android. *IEEE Transactions on Information Forensics and Security*, 7(5):1426–1438, 2012.

[21] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 73–84, 2013.

[22] 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. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–6, 2010.

[23] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Conference on Security*, pages 21–21, 2011.

[24] F-Secure. Trojan:Android/FakeNotify Gets Updated. Available Online, Dec. 2011. http://www.f-secure.com/weblog/archives/00002291.html?tduid=f57e2769518f081721ffca586e797b2a.

[25] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 627–638, 2011.

[26] E. Fernandes, B. Crispo, and M. Conti. FM 99.9, Radio virus: Exploiting FM radio broadcasts for malware deployment. *Information Forensics and Security, IEEE Transactions on*, 8(6):1027–1037, 2013.

[27] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, pages 45–54, 2013.

[28] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proceedings of the 5th International*

Conference on Trust and Trustworthy Computing, pages 291–307, 2012.

[29] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pages 281–294, 2012.

[30] M. Hirzel, D. von Dinklage, A. Diwan, and M. Hind. Fast Online Pointer Analysis. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007.

[31] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth. Slicing Droids: Program Slicing for Smali Code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1844–1851, 2013.

[32] C. Hu and I. Neamtiu. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 77–83, 2011.

[33] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale Malware Indexing Using Function-call Graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 611–620, 2009.

[34] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 36–44, 1998.

[35] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[36] B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, pages 139–160, 2005.

[37] Pandalabs. New Malware Attack through Google Play. Available Online, Feb. 2014. `http://pandalabs.pandasecurity.com/new-malware-attack-through-google-play/`.

[38] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the 21st Annual Network & Distributed System Security Symposium*, 2014.

[39] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, pages 209–220, 2013.

[40] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 329–334, 2013.

[41] D. Sosnoski. Java programming dynamics, Part 1: Java classes and class loading. Available Online. `http://www.ibm.com/developerworks/library/j-dyn0429/`.

[42] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2014.

[43] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on iOS: When Benign Apps Become Evil. In *Proceedings of the 22nd USENIX Conference on Security*, pages 559–572, 2013.

[44] E. R. Wognsen and H. S. Karlsen. Static Analysis of Dalvik Bytecode and Reflection in Android. Master's thesis, Aalborg University, 2012.

[45] L. K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, pages 29–29, 2012.

[46] Y. Zhauniarovich, O. Gadyatskaya, and B. Crispo. DEMO: Enabling Trusted Stores for Android. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 1345–1348, 2013.

[47] Y. Zhauniarovich, G. Russello, M. Conti, B. Crispo, and E. Fernandes. MOSES: Supporting and Enforcing Security Profiles on Smartphones. *IEEE Transactions on Dependable and Secure Computing*, 11(3):211–223, May 2014.

[48] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 93–104, 2012.

[49] Y. Zhongyang, Z. Xin, B. Mao, and L. Xie. DroidAlarm: An All-sided Static Analysis Tool for Android Privilege-escalation Malware. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 353–358, 2013.

[50] Y. Zhou and X. Jiang. An Analysis of the AnserverBot Trojan. Available Online, September 2011. `http://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_Analysis.pdf`.

[51] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 95–109, 2012.