StaDART: Addressing the problem of dynamic code updates in the security analysis of android applications

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

Maqsood Ahmad, Valerio Costamagna, Bruno Crispo, *Senior Fellow, IEEE*, Francesco Bergadano and Yury Zhauniarovich

***Abstract*—Dynamic code update techniques [2], such as dynamic class loading and reflection, enable Android apps to extend their functionality at runtime. At the same time, these techniques are misused by malware developers to transform a seemingly benign app into a malware, once installed on a real device. Among the corpus of evasive techniques used in modern real-world malware, evasive usage of dynamic code updates plays a key role.**

First, we demonstrate the ineffectiveness of existing tools to analyze apps in the presence of dynamic code updates using our test apps, i.e., *Reflection-Bench* and InboxArchiver. Second, we present StaDART, combining static and dynamic analysis of Android apps to reveal the concealed behavior of malware. StaDART performs dynamic code interposition using a vtable tampering technique for API hooking to avoid modifications to the Android framework. Furthermore, we integrate it with a triggering solution, DroidBot, to make it more scalable and fully automated. We present our evaluation results with a dataset of 2,000 real world apps; containing 1,000 legitimate apps and 1,000 malware samples. The evaluation results with this dataset and Reflection-Bench show that StaDART reveals suspicious behavior that is otherwise hidden to static analysis tools.

***Index Terms*—Android, Dynamic Code Updates, Reflection, Dynamic Class Loading, Security Analysis.**

I. Introduction

DYNAMIC code update is a software engineering feature inherited by the Android framework from Java. Theoretically, it enables app developers to update the code base of their apps and completely transform them after being installed on user devices. This feature plays a vital role in maintaining the life cycle of highly feature-rich and evolving Android apps. However, it also possesses the risk of being misused by adversaries to update an app's code base with malicious functionality. Ensuring smartphone users' privacy and security is a major concern and requires adequate measures from app developers, framework providers, and app stores, etc. Google's open source operating system, Android, being the most popular platform for mobile devices, uses Google Bouncer as an app vetting process at its official Google Play store. Vetting processes generally use some form of static/dynamic analysis to scrutinize apps for malicious content and Google Bouncer is no different. In addition, starting from Android 7.0, Android introduced Verify Apps, a new security feature to analyze apps downloaded from sources other than the Google Play store.

However, a growing number of malware samples found in the Android ecosystem reveals that malware developers bypass such vetting processes using various evasion techniques. Previous research shows that the use of dynamic code update techniques along with various forms of obfuscation makes it extremely hard for the state-of-the-art analysis tools to understand the behavior of an app [10], [31], [37]. Thus, the use of these evasion techniques in newly found malware is not surprising [14], [32]. This paper provides an empirical demonstration of the lack of effectiveness of the state-of-the-art tools when it comes to analysing apps that hide suspicious behavior, that is code running with special privileges, using reflection and dynamic code loading. We develop a set of benchmark apps that use reflection in different ways to conceal information leakage. Our analysis of reflection-bench using some of the state-of-the-art static analysis tools shows their ineffectiveness to handle apps that use reflection. Furthermore, we develop InboxArchiver, a seemingly benign app that uses dynamic code loading to hide its suspicious functionality, and use it to test some of the most well known online analysis services. The analysis show that InboxArchiver easily bypasses these security analysis services.

Static analysis relies on the availability of all the information at analysis time, hence, it suffers from dynamic features and unavailability of information that are known only at execution time, e.g., the parameters used in the dynamic code update APIs. Therefore, reflection that is a programming technique widely used by mobile app developers can be only partially investigated by current static analysis tools. As a result, reflection is usually used by malware developers to hide malicious code. The inherent limitation of all static analyzers (e.g., [12], [26]) is the operational assumption 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 updates instantiated only at runtime.

There exist approaches that enhanced static analyzers of Java code to deal with the presence of dynamic code update techniques (e.g., [17]). However, they cannot be applied directly to Android due to the differences in the Java and Android platforms. The alternative of instrumenting the app offline has the major drawback of breaking the app signature, that some apps check at runtime. As the app starts, it checks

The authors are with University of Trento, Italy, KU Leuven, Belgium, University of Torino, Italy and Hamad Bin Khalifa University Doha, Qatar, respectively. (emails: maqsood.ahmad@unitn.it, bruno.crispo@kuleuven.be, valerio.costamagna@di.unito.it, fpb@di.unito.it, yzhauniarovich@qf.org.qa)

the integrity of the signature against a value hardcoded in the app and terminates if the check fails. In case of malicious apps this check may be used to conceal illicit behavior.

In this paper, we present StaDART, a mobile app security analysis tool that combines static and dynamic analysis to address the presence of dynamic code updates. Instead of relying on modifications to the Android framework, StaDART utilizes a vtable tampering technique for API hooking to perform dynamic instrumentation [20]. Furthermore, we integrate StaDART with DroidBot, a triggering tool for Android apps, to make the analysis fully automated. StaDART is evaluated using a dataset of 2,000 real world apps (both malicious and benign) and the results of our evaluation reveal that it is more common in malicious apps to use dynamic code updates to conceal malicious behavior which is only exhibited once the app is installed on a real device. Moreover, 33% of malware samples that use DCL introduce APIs guarded with new (not used in the initial code base) dangerous permissions in the newly loaded code, whereas the analysed benign apps do not exhibit such behavior.

Contributions:

- We present the design and implementation of StaDART, a system that interleaves static and dynamic analysis in order to reveal the hidden/updated behavior of Android apps. StaDART analyzes the code loaded dynamically, and is able to resolve the targets of reflective calls complementing app's method call graph with the obtained information. Therefore, StaDART can be used in conjunction with other static analyzers to make their analysis more accurate.
- We integrate StaDART with DroidBot to make it fully automated and to ease the evaluation. Our analysis results show the effectiveness of StaDART in revealing behavior which is otherwise hidden to static analysis tools.
- We release our tool as open-source to drive the research on app analysis in the presence of dynamic code updates.
- We design and develop reflection-bench, a set of benchmark apps that use reflection to conceal information leakage, and use it to test some of the state-of-the-art static analysis tools. We publish reflection-bench so that researchers can test the effectiveness of their analysis tools in the presence of dynamic features (i.e., reflection).

Paper Organization:

§II provides a background on dynamic class loading and reflection in Android. §III discusses the design and implementation details of reflection-bench and InboxArchiver. It also provides the analysis results highlighting the shortcomings of state-of-the-art Android app analysis tools. §IV gives a high-level description of StaDART, while §VI covers the implementation details. §V presents our approach to build method call graphs and visualise them. §VII reports the evaluation results of StaDART on real world apps. §VIII discusses the limitations of the current implementation, and envisages the future work. §IX overviews the related work, and §X concludes the paper.

II. Dynamic Code Updates in Android

Dynamic code updates techniques, such as reflection and dynamic class loading (DCL), are used to extend apps' functionality at runtime. Inherited from Java into the Dalivk Virtual Machine (DVM), these features are equally supported by Dalvik's successor Android Runtime (ART). Android uses ART to run apps and system services which uses ahead of time (AOT) compilation using a dex2oat tool to convert DEX files into `.oat` binaries. ART is backward compatible with Dalvik runtime and can execute apps compiled for the DVM.

A. Overview of Dynamic Class Loading

DCL provides flexibility to a developer to load classes at runtime. Similar to Dalvik, ART allows a developer to load additional code obtained from alternative locations. It allows apps to load `.zip`, `.jar` and `.apk` files containing a valid `classes.dex` file from outside the app code base, such as files stored on the internal storage or downloaded from the network. Android provides a hierarchy of class loaders which are used to load classes into app's memory.

DCL is usually used for the following purposes:

Extensibility: As shared libraries help developers in building modular software, DCL permits to easily extend the app's capabilities such that developers can programmatically get new code running by loading it via different sources (i.e., network, persistent storage, etc.) at runtime.

App updates: Instead of distributing updated versions of the same app, functionality provided by the current app is extended using updates downloaded through the network and loaded dynamically using class loaders.

Common Frameworks: Depending upon functionality, apps might use certain common frameworks, e.g., an advertisement framework, which shows advertisements to the user. Common frameworks are installed as separate apps whose code can be loaded dynamically by the reliant apps when needed. In the absence of DCL, the functionality provided by the framework must have been implemented in each of the reliant apps. Similarly, in the case of updating that common functionality provided by the framework, only the framework needs to updated rather than updating all the reliant apps.

B. Overview of Reflection

Reflection is the ability of a program to treat its own code as data and manipulate it during execution [16]. Using reflection, an app can reason about and modify its execution state at runtime. The dynamically loaded code is usually accessed using reflection. Android uses the same reflection APIs as used in Java. As a result, reflection APIs can be used to retrieve Class objects, access and modify Class members, create instances of Class and invoke its methods.

In the following, we provide an overview of what reflection offers to a developer, more details can be found in [36].

Conversion from JSON and XML representation to Java objects: Reflection is heavily used in Android to automatically generate 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.

Plugin and external library support: In order to extend the functionality of an app, reflection APIs may be used to call plug-ins or external library methods provided at runtime.

In general, we can conclude that dynamic code loading and reflection are both highly useful and essential for apps, specifically Android apps. Thus, the widespread use of these techniques in modern Android apps is not surprising.

III. Reflection-Bench and InboxArchiver

This section demonstrates how malware developers can evade static analysis tools and the available online analysis systems using dynamic code updates.

A. Reflection-Bench

The usefulness of reflection in Android apps development is undoubted. However, reflection's inherent property to hinder static analysis of apps makes it attractive for malware developers. Although, researchers have worked on app analysis in the presence of reflection in Android apps, literature and the research community still lacks a benchmark of apps which could be used as a test suite to determine the effectiveness of app analysis tools in the presence of reflection. We present *reflection-bench*, a set of Android apps, which use reflection to conceal information leakage to make detection harder for static anlyzers. Reflection-bench is designed so that it can be used to test tools which perform taint analysis as well as those that only generate call graphs for other forms of static analysis. [1]

Overview: Reflection-bench consists of 14 apps which use reflection in various forms to conceal code that exfiltrate sensitive information (i.e, geo-position of the phone) and/or make the flow of the program ambiguous so difficult to analise. The hardness of resolving the targets of reflection depends upon the nature of the arguments used in the reflection APIs. We divide them into two classes, i.e., statically availabe arguments (those string arguments which are provided as part of the app package, e.g., strings defined inside the program, read from a file which is part of the app, etc.) and statically unavailable arguments (those received over the network, read from files on disk, received from other apps, etc.).

Statically unavailable arguments can make it impossible for static analysis tools to resolve reflection. In reflection-bench, we only consider the case of statically available arguments. However, with each case the complexity is gradually increased. In the first few cases, the arguments of reflection APIs are constant strings assigned to program variables. In the latter cases, we consider reading the arguments from a properties file (part of the APK file) and from a hashtable defined inside the program. Moreover, we also consider the cases where the string arguments are formed from the concatenation of multiple strings or decrypted from encrypted strings using crypto APIs. In addition, we consider two levels of complexity where in level one, reflection is used to call only the methods defined inside the app and in level two, both the methods defined inside the program as well as the sensitive APIs, which are responsible for leaking sensitive information, are called through reflection.

Implementation: There are two major classes in each app, i.e, BaseClass and MainActivity. BaseClass has two methods, where `Method1` gets the device ID using the `getDeviceID` API and stores it in a local field `Str`. `Method2` gets a string and sends it out using the `sendTextMessage` API. MainActivity calls `Method1` of BaseClass, gets its field `Str` and sends it to the `Method2` of BaseClass which leaks it out. In the following, we describe how different combinations of reflection APIs are used in each case.

(1) MainActivity retrieves the field `Str` of BaseClass using `getField` reflection API.
(2) MainActivity retrieves an instance of BaseClass using the reflection API `forName`, creates its object using the `newInstance` API and gets its field `Str` using the `getField` reflection API.
(3) MainActivity retrieves an instance of BaseClass using the reflection API `forName`, gets its Constructor using the `getConstructor` API, creates its object using the `newInstance` API and gets its field `Str` using the `getField` reflection API.
(4) MainActivity retrieves an instance of BaseClass using the reflection API `forName`, creates its object using the `newInstance` API and gets its field `Str` using the `getField` reflection API. It also retrieves the methods of BaseClass using the `getMethod` reflection API and calls them using the `invoke` reflection API.
(5) MainActivity retrieves an instance of BaseClass using the reflection API `forName`, gets its Constructor using the `getConstructor` API, creates its object using the `newInstance` API and gets its field `Str` using the `getField` reflection API. It also retrieves the methods of BaseClass using the `getMethod` reflection API and call them using the `invoke` reflection API.
In the above cases, the names of the class "BaseClass", its methods and its field are provided as static strings in the MainActivity class. In the following, starting with Case (4) as a base, we try to acquire/generate these names at runtime.
(6) Reads the names of BaseClass, its methods and its field from a file.
(7) Reads the names of BaseClass, its methods and its field from a Hashtable.
(8) Constructs the names of BaseClass, its methods and its field from multiple strings in the program.
(9) Decrypts the encrypted names of BaseClass, its methods and its field using Crypto APIs.
In all of the above cases, reflection APIs are only used in MainActivity and the sensitive APIs, i.e., `getDeviceId` and `sendTextMessage`, are called directly in BaseClass. In the following cases, we introduce reflection in BaseClass too in addition to Case (4).

[1] Reflection-bench is available to researchers at the following link https://github.com/maqsoodahmadjan/reflection-bench

TABLE I: Analysis with State-of-the-art tools

Apps	Taint Analysis				Call Graphs		
	Flowdroid	IccTa	Amandroid	SCandroid	Androguard	SAAF	StaDART
DataFlow1	✗	✗	✗	-	NA	NA	NA
PlainStringsL1-1	✗	✗	✗	-	✗	✗	✓
PlainStringsL1-2	✗	✗	✗	-	✗	✓	✓
PlainStringsL1-3	✗	✗	✗	-	✗	✓	✓
PlainStringsL1-4	✗	✗	✗	-	✗	✓	✓
FileStringsL1-1	✗	✗	✗	-	✗	✗	✓
HashtableStringsL1-1	✗	✗	✗	-	✗	✗	✓
MultipleStringsL1-1	✗	✗	✗	-	✗	✗	✓
EncryptedStringsL1-1	✗	✗	✗	-	✗	✗	✓
PlainStringsL2-1	✗	✗	✗	-	✗	✗	✓
FileStringsL2-1	✗	✗	✗	-	✗	✗	✓
HashtableStringsL2-1	✗	✗	✗	-	✗	✗	✓
MultipleStringsL2-1	✗	✗	✗	-	✗	✗	✓
EncryptedStringsL2-1	✗	✗	✗	-	✗	✗	✓

(10) BaseClass retrieves an instance of the TelephonyManager class using the reflection API `forName`, creates its object using the `newInstanc` API, gets the sensitive APIs using the `getMethod` reflection API and calls them using the `invoke` reflection API.
In the above case, we use static strings for the names of the class TelephonyManager and the methods `getDeviceId` and `sendTextMessage`. In the following we acquire/generate these names at runtime in addition to Case (10).

(11) Reads the names of TelephonyManager class, methods `getDeviceId` and `sendTextMessage` from a file.

(12) Reads the names of TelephonyManager class, methods `getDeviceId` and `sendTextMessage` from a Hashtable.

(13) Constructs the names of TelephonyManager class, methods `getDeviceId` and `sendTextMessage` from multiple strings inside the app.

(14) Decrypts the encrypted names of TelephonyManager class, methods `getDeviceId` and `sendTextMessage` using Crypto APIs.

Tools analysis results: We report the results of analysis on recent state-of-the-art tools, e.g., Flowdroid [12], Androguard [1], Amandroid [35], SAAF [26], SCandroid [23] and IccTa [28]. A summary of the results is provided in Table I. Those tools which perform taint analysis, such as Amandroid, etc., are analyzed by performing taint analysis of the apps in reflection-bench. However, for those tools which do not perform taint analysis, such as Androguard, etc., we analyze them by generating call graphs of the apps using these tools. In Table I, a ✓ in column X, indicates that the app is successfully analyzed by tool **X**, whereas, a ✗ indicates otherwise.

Amandroid, Flowdroid, IccTa and SCandroid To analyze reflection-bench with Amandroid, Flowdroid, IccTa and SCandroid, we performed taint analysis of the apps using these tools. They analyze APK files and report the presence of sources/sinks of information as well as the tainted paths between these sources and sinks, if any. As shown in Table I, Flowdroid did not report any information leakage in any of the apps. Although, it did report the presence of sources and sinks in some of the apps. Similar results are obtained with Amandroid and IccTa. None of these tools could detect the information flows obfuscated using reflection in reflection-Bench. With IccTa, it is understandably so, because it relies on Flowdroid for information flow analysis. SCanDroid terminated with an error without any meaningful results. This tool is not maintained anymore and no help was available to fix it.

Androguard and SAAF Since Androguard and SAAF generate method call graphs (MCGs) that represent the invoking relationships among methods of apps, we analyze reflection-bench with these tools by generating the MCGs of the apps. In each of the generated MCGs, we look for the app's methods and APIs called through reflection. The first app of reflection-bench is only for those tools which perform taint analysis. It only uses reflection to make the data-flow ambiguous and does not effect the MCG. This is reflected by 'NA' (Not Applicable) in the first row of Table I for the tools that construct call graphs only. The rest of the apps can be used to test both kinds of tools. As shown in Table I, Androguard does not correctly identify any method called through reflection in any of the apps. SAAF's results are relatively better than Androguard's results. SAAF is able to correctly identify the targets of reflection calls in three of the apps in reflection-bench. In these four apps, the arguments provided to the reflection APIs are plain strings. SAAF does not resolve the targets in other cases where the arguments are either read from a file or hashtable, encrypted strings and formed from multiple strings inside the apps. It is important to remember here that none of the apps get any arguments from outside the app.

StaDART StaDART introduces the dynamic element in resolving the targets of reflection (further details are provided in §IV) and, therefore, it performs better as shown in Table I. It is based on Androguard and generates MCGs similar to those generated by Androguard. So, we do not analyze the first app in reflection-bench. Results of the rest of the apps, as shown by the ✓ in column 'StaDART', indicate that all the methods called through reflection are correctly identified by StaDART.

These analysis results show that reflection makes static analysis of apps harder. Specially, when the parameters of reflection APIs are not readily available in the code, static analysis tools find it extremely hard to properly analyze apps.

B. InboxArchiver: Test Malware using DCL

App developers use dynamic code loading for various legitimate purposes, mainly extending the functionality of the app. However, this feature can be used by malware developers to bypass analysis tools deployed at the app markets. A malware developer can submit an apparently benign app with hidden malicious functionality, i.e., obfuscated functionality to load additional code provided once the app is installed on a user's device. Since Reflection-bench only relies on reflective calls, We developed an InboxArchiver app to demonstrate how a malware developer can bypass analysis tools using DCL.

Overview: InboxArchiver is a simple app that reads the SMS inbox and sends some statistics to a number provided by the user. These statistics include the number of SMS messages sent to and received from certain numbers. A user can

configure InboxArchiver to receive a daily, weekly or monthly SMS message containing these statistics. The malicious part of the app, however, downloads some additional code from the Internet which contains other numbers potentially owned by an adversary, loads this code using the DCL APIs and leaks these SMS inbox statistics.

Implementation: The main features of InboxArchiver are the use of DCL and reflection having encrypted strings representing the code paths, class names and method names. This helps InboxArchiver to evade static analysis tools. In order to evade dynamic analysis, it makes use of a simple delay technique where again the APIs are called using reflection with encrypted parameters. It waits for 10 minutes before downloading the malicious code from the Internet and loading it using DCL. Although there are other more sophisticated anti-analysis techniques available, such as emulator detection, root detection, etc., the use of just a delay technique in InboxArchiver highlights the role of DCL/reflection in evading analysis tools.

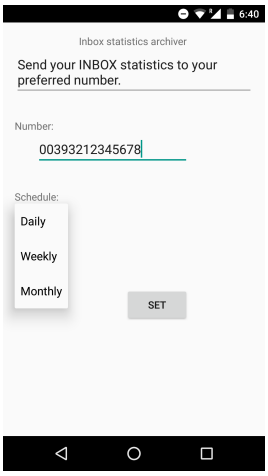


Fig. 1: InboxArchiver

InboxArchiver consists of three main classes, i.e., a `MainActivity` class, a `MessageSender` class and a `Loader` class. The `MainActivity` class presents an interface to the user as shown in Figure 1. The `MessageSender` class, which is a Service and runs in the background, is responsible for retrieving the inbox statistics and sending it periodically to a pre-configured number. After a certain delay, the `MessageSender` class instantiates an object of the `Loader` class which handles the downloading of additional code from the Internet and dynamically loading it using DCL APIs. It makes use of encrypted parameters and encryption/decryption functionality provided by other auxiliary classes.

Analysis results: We uploaded InboxArchiver to a number of online Android app analysis systems. Table II shows a summary of the obtained results[2]. Column *Analyzed* shows whether the app is properly analyzed or not. The next two columns, *Obfuscation* and *DCL*, show if the analysis systems detect obfuscation and the use of dynamic code loading, respectively. The last column in the table represents the final remarks about the app.

Among the online analysis tools shown in Table II, we did not receive any results from CopperDroid and the app is still in the queue for more than a year now. All other tools were unable to detect that the submitted app is malicious. VirusTotal scanned the app with 54 antivirus tools, including BitDefender, GData, AVG, Avast and Kaspersky, etc., and none of them labeled it suspicious. UnDroid and SandDroid termed the app as obfuscated, while SandDroid could also detect dynamic code loading in the app. However, it could not detect the loaded file and analyze it.

[2]Similar proof of concept apps, which were able to bypass the Google Bouncer check using dynamic code update features, can also be found in previous research [19], [31].

TABLE II: InboxArchiver: Analysis Results

Analysis System	Analyzed	Obfuscation	DCL	Malware
VirusTotal [9]	✓	✗	✗	✗
UnDroid [5]	✓	✓	✗	✗
AndroTotal [3]	✓	✗	✗	✗
ds-andrototal [7]	✓	✗	✗	✗
MobiSec Lab [6]	✓	✗	✗	✗
CopperDroid [34]	Queued	-	-	-
SandDroid [8]	✓	✓	✓	✗

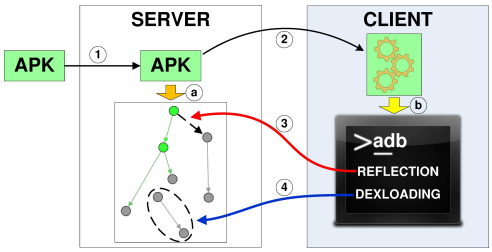


Fig. 2: System Overview

IV. An Overview of StaDART

The lack of effectiveness of state-of-the-art static analysis tools and online analysis systems to capture behavior exhibited at runtime by means of reflection/DCL is clearly evident from the analysis results of reflection-bench and InboxArchiver as discussed in the previous section. Consequently, these tools cannot successfully detect malware that use these dynamic code update features to execute malicious functionality. To address this issue, we design StaDART[3], a tool that interleaves static and dynamic anlaysis, to analyze apps that use reflection and DCL.

The architecture of StaDART presented in Figure 2 comprises two logical components: a server and a client. The static analysis of an app is performed on the server. StaDART 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 that analysis. The client part of StaDART is based on an API hooking technique that intercepts calls to dynamic code update APIs and captures the dynamic behavior. The client part can be hosted either on a real device or an emulator. The client runs the app whenever dynamic analysis is required. StaDART interleaves the execution of the static and dynamic analysis phases and an app can have several of these phases. However, for simplicity of the presentation without loss of generality, we describe them sequentially.

a) Preliminary analysis: The server statically analyzes an app package and builds a MCG of the application (see

[3]Stadart is available to researchers at the following link https://github.com/maqsoodahmadjan/stadart

Step *a* in Figure 2; 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 nodes in the MCG where the functionality of an app 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)*.

b) Dynamic execution: If any MOI is detected in the app, StaDART 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 at run time. In our implementation the dynamic analysis is performed on a device which uses a vtable tampering technique for API call interception and adding StaDART client side functionality. The added functionality logs 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, its parameters and the stack trace (that contains the ordered list of method calls, starting from the most recent ones) is added. In case of a 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*).

c) Analysis consolidation: The server performs an analysis of the obtained information. In case of a reflection call, the server complements the MCG of the app with a new edge (in Figure 2, 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 captures the location of the code file. Using this evidence, the server downloads the file (Step *4*) containing the code, and analyze it statically. The MCG of the app is then updated with the obtained information (see part of the MCG in dashed oval in Figure 2). Additionally, for each downloaded file the server analyzes whether it contains other MOIs. If it does, the list of the MOIs for the app is updated. This allows StaDART to unroll nested MOIs. The stack trace data for both the reflection and DCL cases is used to detect which MOI initiated the call.

d) Marking suspicious behavior: In Android, some API calls are guarded by permissions. Since APIs protected by permissions could potentially harm the system or compromise a user's data, the permissions must be requested in the `AndroidManifest.xml` file. However, there is no actual check on the permissions 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. [15], 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 app dynamically loads 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 app uses reflection APIs to call an API method protected with a *dangerous* permission. 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.

StaDART automatically detects such suspicious patterns and raises a warning if such patterns occur during the analysis. Section VII shows that indeed malware samples do expose such suspicious patterns.

In addition, we further analyze the parameters passed to methods called using reflection APIs. Indeed, a suspicious pattern, i.e., a reflective call to an API guarded with dangerous permission, in conjunction with suspicious parameters, e.g., a premium number in case of the `sendTextMessage()` API, helps in identifying malicious behavior concealed using reflection.

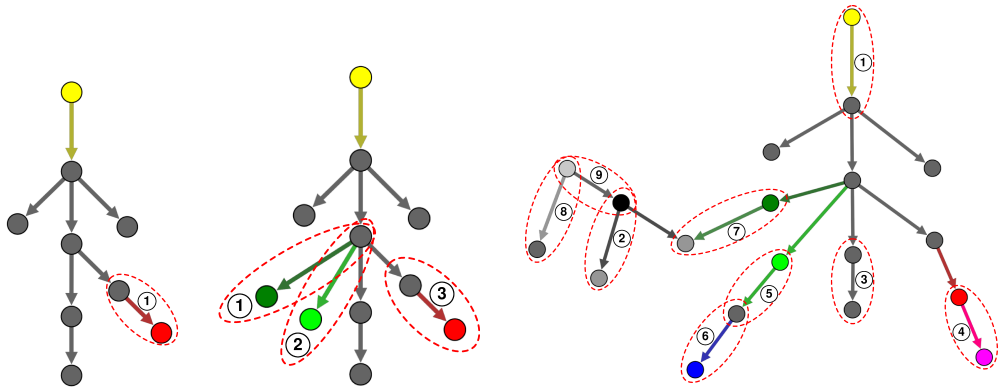
V. Method Call Graph

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 to detect malware, identify potential privacy leaks in apps, find vulnerabilities and execution paths for automatic testing, etc. StaDART extends the initial MCG generated with a traditional static analyzer with the information detected at runtime. Thus, if an app exposes dynamic behavior, all mentioned approaches can benefit from the expanded MCG obtained with StaDART.

a) Example: To visualize the capabilities of StaDART and the process of method call graph expansion, we show the evolution using an example of a *demo_app*. Figure 3a shows the MCG of the app obtained with the AndroGuard static analyzer [1]. Figure 3b shows the one gained with StaDART before dynamic execution phase, and Figure 3c 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 3a 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 3b shows that after preliminary analysis StaDART 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, StaDART adds the edges that are outlined by Ellipses 4-7 (see Figure 3c). 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) (b) (c)

Fig. 3: MCG of *demo_app* Obtained with a) AndroGuard b) StaDART after Preliminary Analysis c) StaDART after Dynamic Analysis Phase

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 apps accompanied by the expanded MCG. 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.

VI. Implementation

The workflow of StaDART's operation is shown in Figure 4. 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, StaDART builds a MCG of the app and exits. Otherwise, the app is analyzed using StaDART client on a device.

A. The server

The server side of StaDART is a Python program that interacts with a static analysis tool. Currently, StaDART uses AndroGuard [1] as a static analyzer. AndroGuard represents compiled Android code as a set of Python objects that can be

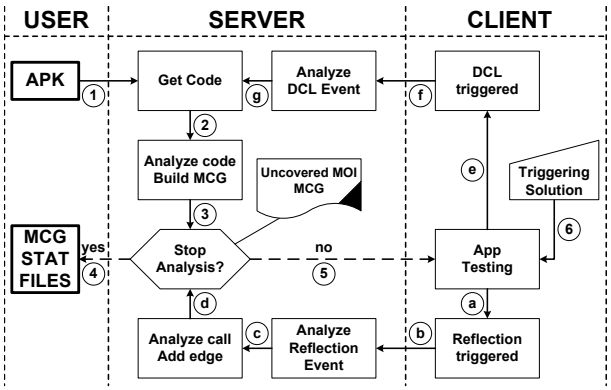


Fig. 4: StaDART Workflow

manipulated and analyzed. However, StaDART 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 [22]) with the one generated by PScout [13] for Android 5.1.1, which is the latest API-permission mapping available in the research community.

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 4; Line 2 in Algorithm 1), and then dissects the extracted code. During this step StaDART searches for all the occurrences of reflection and DCL calls in the code. The list of searched patterns for these API calls is presented in Table III.

If MOIs are found, StaDART selects a device (a real phone or an emulator) to perform the dynamic analysis on (Line 8) and installs the app under analysis (Line 10) onto the client (Step *5* in Fig. 4). 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, the 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),

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:        // analyzeStadartMsg contains a switch statement
16:        // that selects a corresponding processing routine
17:        // shown in Algorithms 2 and 3 based on the msg type
18:        analyzeStadartMsg(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
```

TABLE III: 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		
$Ljava/lang/reflect/Method;$	$invoke$	.

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, StaDART saves the results and finishes its execution.

The function `analyzeStadartMsg` (Line 18) analyzes the selected StaDART 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 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 (algorithm for analysis of *reflection newInstance* messages is very similar so we do not show it). 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 StaDART compares this method with the list of MOIs extracted from the app executable (Lines 10 - 20). If the method is found StaDART 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

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
```

reflection was not found in the app 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, StaDART 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, StaDART adds the information about the dynamic class loading call into the list of vague calls (Line 19).

Notice that the presented algorithms are simplified versions of the ones actually implemented in the server part. For instance, in a real app 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 StaDART are able to process these cases.

B. 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 apps 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 device-independent and easily portable to any other device/emulator.

To capture the dynamic behavior offered by reflection and DCL, we intercept a number of Android API methods that provide an interface to DCL and reflection capabilities. A brief overview of these APIs is provided earlier in §VI. Some of them have been modified across different Android versions moving their implementation to the native side (e.g., `java.lang.Class.newInstance` has only a native implementation in Android 6). To achieve dynamic instrumentation of Java-level APIs we used the approach proposed in ArtDroid [20] to intercept Java virtual methods. It intercepts all calls to monitored Java virtual methods including calls via Java reflection, native code or dynamically loaded code without any modification to both Android OS and the target app. In addition, we integrated native function hooking capabilities in StaDART by means of *inline hooking* technique. The client side employed by StaDART is completely Android version-agnostic and it is able to interpose custom code on both Java methods and native functions. Therefore, it can be used to analyze Android apps on any Android version intercepting DCL and reflection calls irrespective of the actual code representation (i.e., Java or native). To support all available Android versions, we included in StaDART the capability of intercepting DCL and reflections calls according to the running Android version. In the following we describe methods intercepted by StaDART on both Dalvik and ART runtime. The code added by StaDART to perform requested analysis is not influenced by the underlying Android version.

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. Moreover, we added a hook to the constructor of `DexClassLoader` class that is used to create a class loader that loads classes from JAR and DEX files. It gets four parameters as an input, where `dexPath` and `optimizedDirectory` are of our interest. The former specifies the complete path of the DEX file that is being loaded while the latter is the directory where the optimized version will be written to as a result of the compilation step. 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` method of the `Method` class. As of the release of Android version 6, this method is defined as `public native`, thus the client will hook the appropriate function by means of the proper hooking engine, according to the running Android version. Each `Method` object has `declaringClass`, `name` and `parameterTypes` member fields, which represent class name, method name and prototype of the invoked method, respectively. Moreover, `invoke` gets an array of Object type as input which represents the arguments intended for the target method. This information along with the stack trace is put into the StaDART 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. As for the `invoke`, different hooks were added targeting `newInstace` code representation for both DVM and ART runtime.

Each StaDART 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 eventually 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 app. Moreover, due to function overloading it is possible to have several methods in a class with the same name. In the previous version of StaDART (i.e., StaDyna), we had modified the `StackTraceElement` class so that it can store the information about the method prototype, but this approach is not feasible when it comes to dynamic instrumentation. To overcome this limitation and detect MOIs from stack trace data even when they appear multiple times with same name but different prototype, we employed a hybrid approach. First, we statically detect potentially ambiguous methods (i.e., methods in a class with the same name) declared in the target app and for each method found we store its prototype information. Then, we dynamically instrument the app to insert a shadow method that is basically an empty wrapper in order to distinguish calls to the wrapped ambiguous method. The dynamically added wrapper is named as the concatenation of ambiguous method's name and its prototype that has been stored in the previous step. As result of an intercepted call, the wrapper makes a direct call to the wrapped method. In this way, we are able to distinguish target MOIs by looking for them into the stack trace data as it is normally returned by the Android OS. In fact, method name and its prototype allow us to uniquely identify a method in a class.

A StaDART message has a header and a body. To distinguish StaDART 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.

VII. Evaluation

Experiment Setup and Test Suite: This section describes our app test suite and reports on the results of our experiments. We evaluated StaDART with a dataset of real world benign and malicious apps. The server runs on a machine with 3.2 GHz Intel Core i7 processor and 8 GB DDR3 memory. The client is a Google Nexus 6 smartphone running stock Android OS version 7.1.1 connected to the server using a standard USB cable. The evaluation test suite consists of a set of 1,000 benign and 1,000 malicious apps. The benign apps were selected based on their popularity. We selected the 1000 most downloaded apps according to AppFigures [4], an app tracking platform that monitor the downloads and sales of apps from Google and Apple app stores. The malware samples were selected from Drebin [11] dataset populated by 5,560 apps from 179 different malware families. We selected the samples only from families exhibiting DCL as part of malicious behaviour.

Evaluation Goal: In line with the aim of StaDART, i.e., uncovering dynamic behavior, we set certain research questions that this evaluation should answer as our evaluation goal.

- **RQ1:** How widespread is the use of these dynamic code update features in the analyzed dataset and does StaDART reveal dynamic behavior in each of the analyzed app?
- **RQ2:** How effective is StaDART in expanding the MCGs? How expansion of MCGs due to dynamic behavior differ in the malicious and benign dataset?
- **RQ3:** Does StaDART reveal potentially dangerous behavior, i.e., reveal nodes guarded with permissions? How do they differ in benign and malicious apps?
- **RQ4:** Does malware exploit the APIs used for dynamic code updates, e.g., DCL or reflection, to upload dynamically additional code?
- **RQ5:** Do the analyzed apps show suspicious behavior, i.e., use additional new permissions which are not used in the initial MCG? How does this behavior differ in malicious and benign apps?

Analysis Results: Figure 5 illustrates the prevalence of dynamic code update APIs in the analyzed dataset and the effectiveness of StaDART in expanding the MCGs. It shows the percentage of apps with invoke, newInstance and DCL among both benign and malicious app dataset. The right most bar represents the percentage of apps where StaDART expanded the MCG. In the dataset, close to 90% of the apps use `invoke` and/or `newInstance` APIs. Similarly, 48% of the apps use DCL feature which is considerably higher to previous analysis results [40] (first part of RQ1). Increase in the number of apps using DCL could largely be related to the increasing complexity of the Android apps. StaDART was able to expand the MCG by at least one node in 80% of the analyzed apps (second part of RQ1).

Figure 6 shows MCG expansion using StaDART for the apps in the analyzed dataset using reflection only, both benign and malicious. It shows the average percentage increase in the number of nodes, edges, nodes with normal permission and nodes with dangerous permissions. Clearly, the lower percentage increase is attributed to apps that use only reflection as dynamic code update feature. The MCG expansion in these apps, which do not use DCL, is minimal and more or less similar in benign and malicious apps (RQ2).

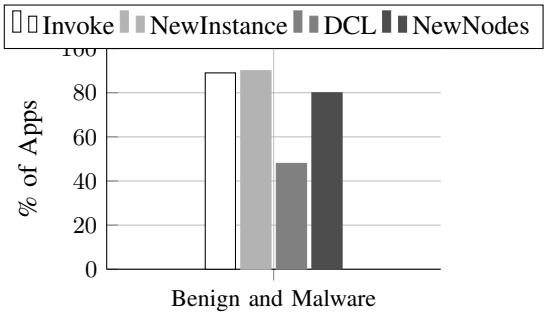


Fig. 5: Prevalence of Reflection/DCL and StaDART effectiveness in expanding MCG

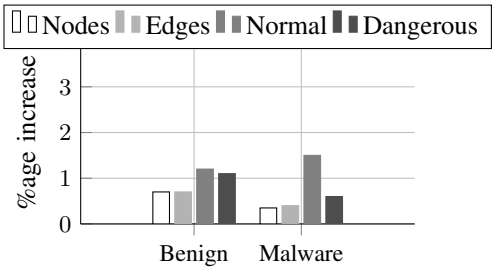


Fig. 6: MCG Expansion

To clarify the role of DCL in MCG expansion and dynamic behavior, we extracted the results from apps that use DCL. Figure 7 shows the effectiveness of StaDART when the apps use DCL. It shows the average percentage increase in the number of nodes, edges, nodes with normal permissions and nodes with dangerous permissions. It clearly shows a considerably higher increase in the number of nodes, edges and nodes guarded with permissions (both normal and dangerous). In addition, it can be seen that the malicious apps hugely increase their code base when they use DCL (RQ4). Similarly, the number of nodes guarded with permissions for malicious apps doubled or in some cases quadrupled (RQ3). This clearly indicate that malicious apps make use of sensitive APIs in the loaded code. We also check the added nodes for Signature level permission and SignatureOrSystem level permission. However, we did not observe a noticeable increase in the number of nodes guarded with these permissions.

Although, the high increase in the number of nodes guarded with dangerous permissions is indeed suspicious, we investigate the analysis results further for a more suspicious malware behavior. In practice, malicious payloads are packaged inside legitimate apps and their manifest files are modified to cover for the extra permissions needed by the payload. In this scenario, the final MCG of the app contains nodes guarded with new permissions, i.e., those not found in the initial MCG.

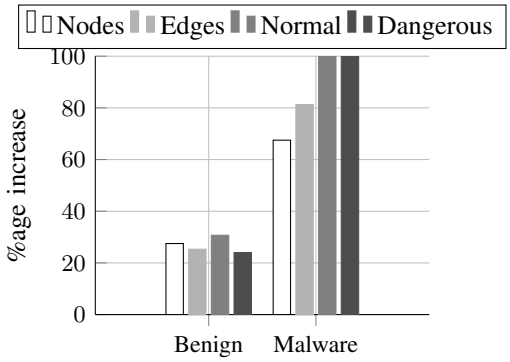


Fig. 7: MCG Expansion when apps use DCL

Figure 8 shows the distribution of apps based on increase in the number of nodes guarded with permissions in the form of pie-charts, in benign apps and malware. Here we discuss only those apps which use DCL. The white part shows the percentage of apps with no increase in the number of nodes guarded with permissions, whereas the grey part represents the percentage of apps with increase in the number of nodes guarded with permissions. The darker grey part shows the percentage of apps where new permissions are used in the dynamically added part using StaDART.

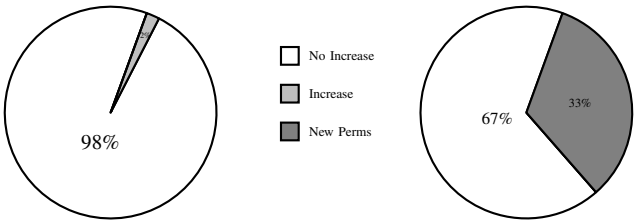


Fig. 8: Increase in permission nodes. (L) Benign (R) Malicious apps

The pie-chart for the benign apps shows that a very small fraction of the apps observe an increase in the number of nodes guarded with dangerous permissions. In contrast, a considerably higher number of malicious apps reveal such behavior. Also, in none of the benign apps in the dataset, the loaded code contained nodes guarded with new dangerous permission. However, all the malicious apps in the dataset that loaded code dynamically contained nodes guarded with at least one new dangerous permission (RQ5). Moreover, a further analysis of the loaded code in malicious apps reveals a pattern of dangerous permissions, e.g., `READ_PHONE_STATE` and `INTERNET`, that could be associated with malicious functionality, such as privacy leakage, etc.

Also, noteworthy here is the fact that the revealed behavior is only due to triggering of a small fraction of the total MOIs. Albeit the most advance automated triggering tool in the research community, DroidBot does not serve well for app exploration from a security point of view. Taking into account the low exploration that DroidBot achieved in most of the apps and the suspicious results that we observed, the actual hidden suspicious/malicious behavior could be alarming.

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 StaDART can be handy in exploration of this topic.

VIII. Discussion

For any dynamic (or hybrid for that matter too) analysis tool, coverage is the main limiting factor and StaDART is no different in that regard. For StaDART the coverage of MOIs (the ratio between the number of executed MOIs at least once and total number of discovered MOIs) is especially important. In order to achive higher MOI coverage, we explored if the tools like *monkey* 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. Therefore, we opted for a more advance automated triggering tool, DroidBot, to trigger MOIs. However, as discussed in the previous section, even DroidBot did not achieve reasonable coverage of MOIs. Possible enhancement can be achieved using techniques such as the one used in SmartDroid [41]. SmartDroid allows an expert to specify sensitive API methods required to be triggered. In case of StaDART the sensitive API methods correspond to reflection and DCL calls.

Another possible direction to reduce the dependence on the triggering tool is to resolve as many targets of reflection calls as possible statically, at least those which are represented by constant strings [26]. The analysis performed in [22] 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. Consequently, we can minimize the more expensive dynamic part of the analysis.

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 StaDART is to incorporate information obtained during different runs of analysis.

StaDART 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, StaDART 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 manually analyzed later.

IX. Related Work

Apps are analyzed for malicious contents before being published to the app markets. Many static and dynamic analysis techniques have been proposed for Android. The ded system re-targets Dalvik bytecode into Java class files

that can be analyzed by the variety of tools developed for Java. DroidAlarm [42] 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. Gascon et al. [24] employ comparsion of functional call graphs (FCG) mined using AndroGuard to detetct malicious Android apps. StaDART can complement these techniques by providing more precise graphs required for analysis.

TaintDroid was among the first dynamic analysis tools for Android apps [21]; it tracks propagation of information via the TaintDroid infrastructure-equipped smartphone software stack. It detects leakage of user private information to network interfaces. This approach is followed by DroidScope [38]. 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 apps with the system and a user (UI interactions). Several approaches are proposed to automate the triggering of UI events, from random event generation [27] to more advanced approaches like AppsPlayground [33] and SmartDroid [41]. However, all of them still have many limitations on the type of events they can handle and the coverage.

Poeplau et al. [31] 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 be signed and its signature has to be included into a special list distributed by trusted authorities. However, as mentioned in the article [31], extraction of the dangerous behavior is a difficult problem by itself, especially when the protected API is called through reflection. In contrast, StaDART 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.

Comparing to Stadyna [40], StaDART differs in various aspects. The client side of StaDART is based on API hooking using a vTable tampering technique used in ArtDroid, rather than modification to the Android framework, and therefore, can be easily ported to different versions of Android. Also, StaDART analyzes the arguments passed to the APIs/methods called using reflection API `invoke`. On the client side, unlike Stadyna which requires a human user to interact with the app during analysis, StaDART relies on a triggering tool, DroidBot, to make the analysis fully automated. StaDART is evaluated on a much larger set of applications, 2,000 apps (1,000 benign and 1,000 malicious) in comparison to Stadynas 10 apps (5 benign and 5 malicious).

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 [25] 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 "online" 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 on the call graph of a program and determines how heap objects are linked to each other (e.g., if a variable can be accessed from several threads). As call graph produced from java program can be incomplete, [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 et. al. [30] proposed a refinement of 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 StaDART 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 StaDART 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 app signature in its code that is changed during the patching. Thus, for these apps 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 StaDART is able to process correctly this case due to dynamic API hooking.

More recently, reflection aware analysis of Android apps has been the focus of some research publications. For example, DroidRA uses string inference analysis to resolve reflective calls and replaces them with regular Java calls by instrumenting apps for further analysis [29]. However, DroidRA cannot resolve the targets of reflection when the arguments to reflection APIs are not readily availabe in the app. StaDART's dynamic element could prove fruitful in this regard. Ripple uses a combination of formal analysis and pointer analysis to ensure reflection aware static analysis in incomplete information environment (IIE) [39]. Although Ripple resolves most targets of reflection, various cases of IEE lead to high false positives. In fact, Ripple in conjunction with StaDART could prove beneficial for both where Ripple reducing the dynamic analysis part of StaDART and StaDART reducing the false positive rate of Ripple.

X. Conclusion

Today mobile apps 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 presented StaDART, a technique that interleaves static and dynamic analysis in order to scrutinize Android apps in the presence of reflection and dynamic class loading. Our approach makes it possible to expand the MCG of an app by capturing additional modules loaded at runtime and additional paths of execution concealed by reflection calls. In order to produce the expanded call graph, StaDART relies on code interposition based on a dynamic API hooking technique. It does not require any modification to the Android framework or the app itself. As observed from the evaluation results malware apps were more inclined to exhibit a suspicious increase in dangerous permissions after dynamic loading of new code, proving that StaDART is an effective hybrid approach able to detect and capture apps' dynamic capabilities used at runtime.

The results produced by StaDART 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, StaDART may help malware analysts by increasing their ability to detect suspicious samples.

References

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

[2] Android Studio – Support for Dynamic Delivery. https://developer.android.com/studio/projects/dynamic-delivery.

[3] Andrototal - free service to scan suspicious apks against multiple mobile antivirus. http://andrototal.org/.

[4] AppFigures. https://appfigures.com/.

[5] Avc undroid. http://undroid.av-comparatives.info/.

[6] Mobisec lab. http://www.mobiseclab.org/.

[7] Previously ds-andrototal - now droydseuss. http://droydseuss.necst.it/.

[8] Sanddroid - android app analysis tool. http://sanddroid.xjtu.edu.cn/.

[9] Virustotal - free online virus, malware and url scanner. https://www.virustotal.com.

[10] M. Ahmad, B. Crispo, and T. Gebremichael. Empirical analysis on the use of dynamic code updates in android and its security implications. In *Nordic Conference on Secure IT Systems*, pages 119–134. Springer, 2016.

[11] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[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. I. Aysan, F. Sakiz, and S. Sen. Analysis of dynamic code updating in Android with security perspective. *IET Information Security*, 13(3):269–277, 2018.

[15] 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.

[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] G. Canfora, F. Mercaldo, G. Moriano, and C. A. Visaggio. Composition-malware: building android malware at run time. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 318–326. IEEE, 2015.

[20] V. Costamagna and C. Zheng. Artdroid: A virtual-method hooking framework on android art runtime. *Proceedings of the 2016 Innovations in Mobile Privacy and Security (IMPS)*, pages 24–32, 2016.

[21] 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.

[22] 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.

[23] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android. Technical report, 2009.

[24] 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.

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

[26] 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.

[27] 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.

[28] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.

[29] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 318–329. ACM, 2016.

[30] 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.

[31] 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.

[32] A. Polkovnichenko and A. Boxiner. Braintest - a new level of sophistication in mobile malware. Technical report, Check Point Technologies Ltd.

[33] 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.

[34] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[35] F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

[36] E. R. Wognsen, H. S. Karlsen, M. C. Olesen, and R. R. Hansen. Formalisation and analysis of dalvik bytecode. *Science of Computer Programming*, 92:25–55, 2014.

[37] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang. Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique. *IEEE Transactions on Information Forensics and Security*, 12(7):1529–1544, 2017.

[38] 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.

[39] Y. Zhang, T. Tan, Y. Li, and J. Xue. Ripple: Reflection analysis for android apps in incomplete information environments. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 281–288. ACM, 2017.

[40] 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 *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 37–48. ACM, 2015.

[41] 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.

[42] 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.