securityandtrust.lu

Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android

Alexandre Bartel, SnT, University of Luxembourg
Jacques Klein, SnT, University of Luxembourg
Martin Monperrus, University of Lille (France)
Yves Le Traon, SnT, University of Luxembourg

3 October 2011

978-2-87971-107-2

Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android

Alexandre Bartel[A], Jacques Klein[A], Martin Monperrus[B,1] and Yves Le Traon[A]
[A] University of Luxembourg, Interdisciplinary Center for Security, Reliability and Trust (SnT)
[B] University of Lille

***Abstract*—Android based devices are becoming widespread. As a result and since those devices contain personal and confidential data, the security model of the android software stack has been analyzed extensively. One key feature of the security model is that applications must declare a list of permissions they are using to access resources. Using static analysis, we first extracted a table from the Android API which maps methods to permissions. Then, we use this mapping within a tool we developed to check that applications effectively need all the permissions they declare. Using our tool on a set of android applications, we found out that a non negligible part of the applications do not use all the permissions they declare. Consequently, the attack surface of such applications can be reduced by removing the non-needed permissions.**

***Keywords*-permissions, call-graph, android, security, soot, java, static analysis**

I. Introduction

Android is one of the most widespread mobile operating system in the world accounting 48% market share [5]. More than 300 000 Android applications available on dozens of application markets can be installed by end users. The other side of the coin is that all kinds of malware are waiting to be installed on thousands of Android devices. For instance, Zeus [15] sends banking information to malicious servers. This motivates researchers and engineers to devise security models, architectures and tools that are able to mitigate the malware harmfulness.

The security architecture of Android, the Google Chrome browser extension system and the Blackberry base platform, all use a similar security model called the permission-based security model [2]. A permission-based security model can be loosely defined as a model in which 1) each application is associated with a set of permissions that allows accessing certain resources[2]; 2) permissions are explicitly accepted by users during the installation process and 3) permissions are checked at runtime when resources are requested.

This permission model entails intrinsic risks. For instance, not all users may be able to cleverly reject powerful permissions at installation time. Malwares may also use platform vulnerabilities to circumvent runtime permission checks. Finally, applications can be granted more permissions than they actually need, what we call a "permission gap". Malwares have many ways to exploit permission gaps, for instance using code injection or return-oriented programming [7], and can leverage the unused permissions for achieving their malicious goals. Identifying permission gaps means reducing the risks for an application to be compromised, also known as reducing the application attack surface [17].

Let us make an analogy with a firewall. In a correctly configured firewall only the ports that are used are open. All the other ports are closed. However if the firewall is misconfigured, some unused ports remain open and the attack surface of the infrastructure behind the firewall is larger. For instance, let us assume that an information system internally uses a remote shell service on port 544. If port 544 is open on the firewall, an attacker could perform attacks on the remote shell server located behind the firewall. In the same way, an application that requires too many permissions, i.e. that suffers from a permission gap, may allow an attacker who compromised the application to access more resources than he should have.

Permission gaps appear because the process of declaring application permissions is manual and error-prone: Android framework developers manually document which permissions are required for each system resource, and Android application developers manually declare the permissions they *think* are needed. This paper presents an approach to support those *manual* software engineering activities with an *automated* tool. This approach secures permission-based software in the sense that it reduces the attack risks (not in the sense that the resulting software is unattackable).

Our tool, called COPES (COrrect PErmissions Set), proceeds as follows. First, using static analysis, it extracts from the Android framework bytecode a table that maps every method of the API to a set of permissions the method needs to be executed properly. Second, COPES lists all framework methods used by an application, based on static analysis of the application bytecode. Third, COPES computes the set of permissions that are required for the application to run, which means that all permissions in this set are used at least

[1]This work was initiated while M. Monperrus was at the Technische Universität Darmstadt.

[2]Contrary to the traditional Unix permission system where permissions are at the level of users, not applications.

once in the application, and consequently no permission gap remains. Eventually, COPES computes the permission gap as the difference between the declared permissions and the required permission. COPES can also help Android framework experts to comprehensively document the framework and novice application developers to automatically infer an initial set of permissions to declare.

To sum up, the contribution of this paper is an approach to identify and fix permission gaps in permission based software. More specifically:

- We show that the permission-based security model can be expressed within a boolean matrix algebra. This algebra is not specific to Android.
- We present a novel methodology to compute a close approximation of the required permission set and the permission gap based on static analysis, as opposed to concurrent work that uses testing [13].
- We discuss the design and the implementation of the approach for the Android platform.
- We evaluate our approach on 742 android applications and we show that 94 applications suffer from a permission gap.

The reminder of this paper is organized as follows. In Section II we explain the reason why reducing the attack surface is important and present a short study supporting our intuition. In Section III we propose a formalization for permission-based software and a generic method for deriving correct application permission sets. In Section IV we describe the android system and focus on the access control mechanisms. Then, in Section V we apply the generic method on the Android system. Experiments we conducted and results are presented and discussed in Section VI. We present the related work in Section VII. Finally we conclude the paper and discuss open research challenges in Section VIII.

II. The Permission Gap Problem

This section further details the permission gap problem introduced in the introduction, and presents short empirical facts showing that this problem actually happens in practice.

A. Possible Consequence of a Permission Gap

Let us consider an Android application, app_{wrong}, that is able to communicate with external servers since it is granted the INTERNET permission. Moreover, app_{wrong} has declared permission CAMERA while not using it. The CAMERA permission allows the application to take picture without user intervention. In this example, the permission gap consists of one permission: CAMERA.

Unfortunately, app_{wrong} uses a native library on which an buffer-overflow exploit has recently been discovered. As a result, app_{wrong} becomes subject to remote attacks through specific payloads. Consequently, attackers are able to attack devices that are running app_{wrong} in order to take pictures using the camera's device and send them to a remote location on the Internet.

On the contrary, if app_{wrong} did not declare CAMERA, this attack would not have been possible, and the consequences of the buffer-overflow exploit would have been mitigated. As noted by Manadhata [17], reducing the attack surface does not mean no risks, but less risks.

In order to show that this example of misconfigured applications is not artificial, we now discuss a short empirical study on the declaration of two permissions on 1000+ Android applications.

B. Declaration and Usage of Permissions CAMERA and RECORD_AUDIO

We conducted a short empirical study on a 1000+ Android applications downloaded from an application market (http://www.freewarelovers.com/android/).

For two permissions CAMERA and RECORD_AUDIO, we grepped the source code of the Android framework to find the methods requiring one of them. These two sets of methods are noted M_{CAMERA} and M_{RECORD_AUDIO}. Then, we computed the list A of all the applications which declare CAMERA or RECORD_AUDIO. Next, we took each application $app \in A$ individually and we checked the application uses at least one method of M_{CAMERA} and M_{RECORD_AUDIO} by analyzing the application's bytecode. If it is not the case, it meant that app is not using the corresponding permission. When this happened, we modified the application manifest that declares the permission and run the application again to make sure that our grepping approximation did not yield false positives.

The results are shown in table I. More than 8% of the applications declaring permissions CAMERA or RECORD_AUDIO do not use the permission. Those results confirm our intuition: declared permission lists are not always required, hence permission gaps exist. Developers would benefit from a tool that automatically computes the set of required permissions.

Permission P	Declare P	Do not use P
CAMERA	82	7 (8.5%)
RECORD_AUDIO	35	3 (8.6%)

Table I
Applications that declare a permission but do not use it

III. Manifest Inference

In this Section we formalize the concept of Permission-Based Software and propose a generic methodology to compute a mapping from code to permissions that are required for the application to run.

Permission-based software is conceptually divided in three layers: 1) the core platform which is able to access all system resources (e.g. change the network policy), it is generally the operating system; 2) a middleware responsible for providing a clean application programming interface

(API) to the OS resources and for checking that applications have the right permissions when they want accessing them; 3) applications built on top of the middleware. They have to explicitly declare the permissions they require. Layers #2 and #3 motivate the generic label "permission-based software". Since the middleware also hides the OS complexity and provides an API, it is sometimes called, as in the case of Android, a "framework".

We automatically infer the list of permissions required by an application by executing the following steps:

1) perform a static analysis of the framework to determine the list of permissions required for every method of every class of the framework. Let us call this matrix M. This step has to be done only once for every framework.
2) perform a static analysis of the application to obtain a list of framework methods called by the application, which we normalized in respect to M to obtain the vector AV (for Access Vector).
3) compute the boolean vectorial product of AV by M to to obtain the permissions that are actually required by the application.

We start by defining the different elements of our approach and then we detail the three aforementioned steps.

A. Definitions

Definition 1 (Framework)**.** A framework $\mathcal{F}$ is a layer that enables applications to access resources available on the platform. We model it as a bi-partite graph[1] between framework API methods and resources.

Example: In the case of Android, $\mathcal{F}$ is the Android Java Framework composed of 4000 classes and 142000 methods. To access a resource, an android application has to make a method call that goes through $\mathcal{F}$.

Definition 2 (Permission-based system)**.** A permission-based system is composed of at least one framework, a list of permissions and a list of protected resources. Each protected resource is associated with a fixed list of permissions

Definition 3 (Entry point)**.** An entry point of a framework is a method that an application can use. Constructors are also considered as entry points. We denote $Entry_{\mathcal{F}}$ the set of all entry points of $\mathcal{F}$.

Example: One of the entry points of the Android framework is the method *getAccounts()* from class *android.accounts.AccountManager*.

An application can call any method of the framework. Some methods accessing some system resources (like an account) are protected by one or more permissions. Let us suppose that the method *getAccounts()* allows access to a set of accounts and is protected by one permission `GET_ACCOUNTS`. An application can successfully call method *getAccounts()* if and only if it declared `GET_ACCOUNTS` in the application-specific list (this list is contained in a "manifest", we shall use this term later in the paper).

Definition 4 (Permission)**.** A permission is a token that an application needs to access a specific resource. We make no assumptions on permissions, and we consider them as independent (neither grouped, nor hierarchical) .

Example: Developers of an Android application define a list of permissions in a file called the Manifest. To read contact information, the manifest of the application must declare the `READ_CONTACT` permission.

Permissions can be checked at different levels in the system. We call high-level permissions the set $P = \{p_1, p_2, ..., p_n\}$ of permissions that are checked at the framework level. Low-level permissions are permissions that are checked at the operating system level.

Definition 5 (High-level permission)**.** A high-level permission, is a permission that is only checked at the framework level.

Example: In the case of Android, `READ_CONTACT` is a high-level permission.

Definition 6 (Low-level permission)**.** A low-level permission is a permission associated with a high-level permission and is checked at a lower level than the framework level.

Example: In this paper we are only interested in high-level permissions. They are 135 such permissions in the Android system, while 8 permissions are checked at a low-level. This shows that the framework is responsible for most of the work related to permissions. Note that if a permission is checked at the operating system level, it is not possible to detect that an application uses it by only analyzing the framework. In the case of Android, `INTERNET` is a low-level permission. Indeed this permission is checked at the filesystem level by verifying the presence of the *inet* group ID when accessing a socket.

Definition 7 (Declared permission)**.** A declared permission for an application app is a permission which is in the permission list of app. The set of all declared permission for an application app is noted $P_d(app)$.

Definition 8 (Required permission)**.** A *required permission* for an application app is a permission associated with a resource that app uses at least once. The set of all required permissions for an application app is noted $P_{req}(app)$.

Example: For an application app, if the set of required

[1] a bi-partite graph "is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V" [26]

permissions $P_{req}(app)$ is equal to the set of declared permissions $P_d(app)$, the permission attack surface is minimal.

Definition 9 (Inferred permission)**.** An inferred permission for an application app is a permission that an analysis technique found to be required for app. This paper presents such a technique and computes a set of inferred permissions noted $P_{ifrd}(app)$. Depending on the analysis technique used, the inferred permission list may be either an over- or an under- approximation of the required permission list. When using static analysis techniques, the inferred permission list may be an *over approximation* ($P_{req}(app) \subseteq P_{ifrd}(app)$). The inferred permission list may be an *under-approximation* of the required permission list ($P_{ifrd}(app) \subseteq P_{req}(app)$) when using testing techniques[2]

When developers write manifests, they write $P_d(app)$by trying to guess $P_{req}(app)$ based on documentation and trial-and-errors. In this paper, we propose to automatically infer a permission list $P_{ifrd}(app)$ in order to avoid this manual and error-prone activity. We take a special care in minimizing the difference between $P_{ifrd}(app)$ and $P_{req}(app)$.

Let us know that we can compute $P_{ifrd}(app)$ by simply expressing our definitions in a boolean matrix algebra.

Definition 10 (Access vector)**.** Let app be an application. The access vector for app is a boolean vector AV_{app} representing the entry points of the framework the application app can reach. An element of the vector is set to *true* if the corresponding entry point can be reached by the application. Otherwise it is set to *false*.

Example: Let us consider a framework with three entry points (e_1, e_2, e_3), and an application app with the following access vector:

$$AV_{app} = (0, 1, 1)$$

This access vector expresses that app's code may reach e_2 and e_3 but not e_1.

Definition 11 (Permission Access Matrix)**.** The permission access matrix M is a boolean matrix which represents the relation between entry points of the framework and permissions. Rows represent entry points of the framework and columns represent permissions. A cell $M_{i,j}$ is set to *true* if the corresponding entry point (at row i) accesses a resource protected by the permission represented by column j. Otherwise it is set to *false*.

Example: For a framework with three entry points (e_1, e_2 and e_3) and four permissions (p_1, p_2, p_3 and p_4), the permission access matrix could be:

$$M = \begin{array}{c} \\ e_1 \\ e_2 \\ e_3 \end{array} \begin{array}{c} \begin{array}{cccc} p_1 & p_2 & p_3 & p_4 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

[2]Indeed, testing will observe only the permissions it executes, potentially missing some permission checks.

This means that e_1 requires permissions p_3, e_2 requires permission p_2 and that e_3 requires permissions p_1, p_2 and p_3.

Definition 12 (Infered permissions vector)**.** Let app and $\mathcal{F}$ be an application and a framework respectively. The inferred permissions vector, IP_{app}, is a boolean vector representing the set of inferred permissions for application app. We have $IP_{app} = AV_{app} \times M$ by using the boolean operators AND and OR instead of arithmetic multiplication and addition in the matrix calculus. A cell $IP_{app}(k)$ is set to *true* if the permission at index k is required by app. Otherwise it is set to *false*. Note that $P_{ifrd}(app)$ is the set of all permissions set to *true* in IP_{app}, i.e. $P_{ifrd}(app)$= $\{permission_x | IP_{app}(x)\}$.

Example: Using AV_{app} and M from the two previous examples, the inferred permissions vector for app is:

$$\begin{aligned} IP_{app} &= \begin{pmatrix} 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix} \\ IP_{app} &= \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix} \end{aligned}$$

Application app should declare permissions p_1, p_2 and p_3. We believe that this model can describes most permission-based systems.

B. Step 1: Extraction of the Permission Access Matrix M

In this section we present a methodology to extract the permission access matrix M of a framework $\mathcal{F}$. This methodology is based on a static analysis of the framework $\mathcal{F}$. The idea is first to compute a call graph for every entry point of the framework and then to detect whether or not permission checks are present in the call graph.

1) Definitions:

Definition 13 (Call graph)**.** A call graph is a directed graph G containing a set of vertices V representing method calls and a set of arcs A representing links between method calls. In other terms, $G = (V, A)$, $A = \{a | a = (u, v), u, v \in V\}$.

Definition 14 (Permission Enforcement Point)**.** A Permission Enforcement Point (PEP) is a vertex of a call graph whose signature corresponds to a system method which checks permission(s). Each PEP is associated with a list of required permissions $perms_{PEP}$.

Example: In the callgraph starting from entry point e_4 represented in Figure 1, ck_2 is a call to a PEP. This PEP is a system method which checks permissions like `Context.checkPermission()` with the parameter `android.permission.FLASHLIGHT`.

2) PEP localization: To localize in which methods PEP are located, we traverse a call graph $G = (V, A)$ generated from the framework and check whether a vertex V_{PEP} is a PEP. Methods which directly check for a permissions are represented as vertices V_{M_i} ($i \in$ {1,2,...,k}), such that $(V_{M_i}, V_{PEP}) \in A$.

3) Generation of matrix M: We compute one call graph G_i per entry point e_i of the framework ($i \in$ {1,2,...,n}). Then , matrix M is constructed as follows:

- M is set as a matrix of size (|entry points| $\times$ |high level permissions|)
- all elements of M are initialized to false
- for each e_i that reaches one ore more PEP, and for each permission j in $perms_{PEP}$, $M(i,j) = true$.

Example: A framework with four entry points (e_1, e_2, e_3, e_4), and three permissions (p_1, p_2, p_3) is presented in the lower part of Figure 1. For every of those entry point a call graph is constructed. Three of those call graphs have a PEP node: e_1 and e_2 have PEP ck_1 which check permission p_1 and e_4 has PEP ck_2 which check permission p_2[3]. On the figure a dashed arrow connect each PEP to the permission(s) it checks. The framework matrix is then:

$$M_{ex} = \begin{array}{c} \\ e_1 \\ e_2 \\ e_3 \\ e_4 \end{array} \begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

C. Step 2: Extraction of the access vector AV

We now explain how to compute the list of methods (i.e. the list of entry points) of a framework $\mathcal{F}$ called by an application app in order to extract the access vector AV.

For that purpose we generate a call graph for every entry point of app and check whether or not a call to an entry point of $\mathcal{F}$ is present in the generated call graph. AV is a boolean vector and its element correspond to entry points of $\mathcal{F}$. Thus, the length L of vector AV is the number of entry points of $\mathcal{F}$. For every entry point of $\mathcal{F}$ called in app the equivalent element of AV is set to *true*.

Example: The application example in Figure 1 features a single entry point, s. From s a call graph G_{ex} is generated. All elements of vector AV_{ex} of length $n = 4$ are initially set to $false$:

$$AV_{ex} = (0, 0, 0, 0)$$

Then for every vertex of G_{ex} which is a call to the example framework, the corresponding element of AV_{ex} is set to $true$. In the example, there are three such vertices (represented as entry points e_1, e_2 and e_3 in the Figure). This leads to the following vector AV_{ex}

$$AV_{ex} = (1, 1, 1, 0)$$

D. Step 3: Inferred Permission List

Following Definition 12 we compute IP_{app} the inferred permission vector. The inferred permission list contains all permissions set to *true* in IP_{app}.

[3]Notice that p_3 is never checked in the framework. Note that the Android framework defines several permissions which are never checked.

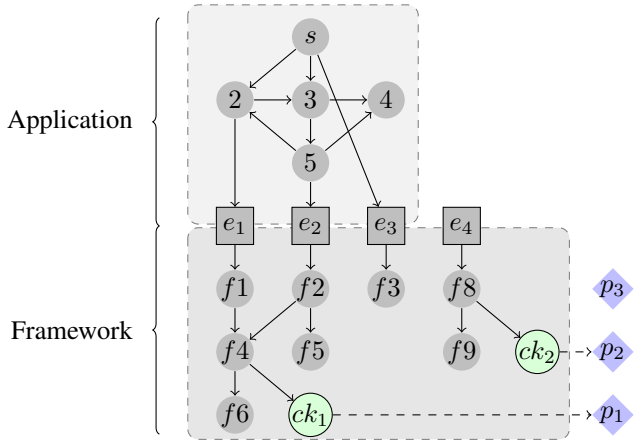

Figure 1. Application and Framework Example

Example: Using matrix M_{ex} and vector AV_{ex} generated above for the example framework and application represented in Figure 1, we obtain:

$$IP_{ex} = \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$IP_{ex} = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix}$$

This means application app should only declare permission p_1.

Before applying this generic methodology to the Android system, we will first have an overview of Android. Understanding the target software system is a necessary step to correctly identify the framework part of the system and know where and how permissions are checked.

IV. Overview of Android

This Section describes the Android system. We detail how and where access control is enforced regarding high-level permissions and how applications access the framework$_{\mathcal{F}}$.

A. Software Stack

Android is a software stack as shown Figure 2. It features a modified Linux kernel, C/C++ libraries, a virtual machine called Dalvik, a Java framework and a set of basic applications (including a phone application). Applications for Android are written in Java. An android application is packaged into a .apk (android package) file which contains the Dalvik bytecode, data (pictures, sounds, ...) and the Android Manifest file. The developer defines permissions the application may use in this Manifest.

B. Structure of an Application

An Android application is made of *components* which can be:

- an *Activity* which is a user interface

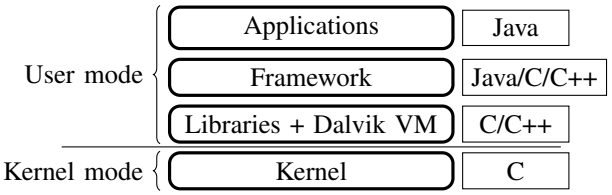

Figure 2. Android software stack

- a *Service* which runs in the background
- a *BroadcastReceiver* which receives Intents (a kind of message comparable to inter processes communication, aka IPC)
- a *ContentProvider* which is a kind of database backend used to store and share raw data.

An application uses URI (Uniform Resource Identifiers [24]) to locate and work with local or remote (from other applications) content providers. Moreover, applications asynchronously communicate with services using Intents through a system called *Binder* (e.g. an Intent may be "display the phone dialer"). We detail services in the following Section IV-C, as they are widely used by the system for enforcing permission checks. When performing the static analysis in Section V we rely on the understanding of the Binder, described bellow, to reconstruct RPC calls.

C. Services

1) Building a Service: Services are identified in the source code by two files: (1) an .aidl (Android Interface Description Language) and (2) a .java file. The AIDL describes the interface and the Java file implements the service. From the AIDL file, two static Java classes are automatically generated at compile time: a proxy (3) and a stub (4). The stub extends the Binder class and implements the service's interface, it is located on the service side and is extended by (2). The proxy is used on the application side to call a remote method on the service. A proxy and its stub communicate through the Binder implemented as a Linux kernel module and available through */dev/binder*.

2) Calling a Service's Method: The first step for the application wanting to use a remote service is to dynamically get a reference to the service. The next step is to call a method on the reference. The binder intercepts that call and performs the actual call on the remote service.

Example: We use the AccountManagerService to exemplify Binder communication. An application wanting to use the account manager service has first to call `AccountManager.get(Context)` a static method which returns an initialized AccountManager object. Context is an abstract class implemented by ContextImpl. It allows an application to access to resources, to launch intents or activities and more. The `get()` method of class AccountManager calls `Context.getSystemService()` to fetch a reference to the binder of the service, retrieves the proxy of the service and initializes AccountManager with the account service's proxy. Internaly, `getSystemService()` calls `ServiceManager.getService()`. This first step is represented as "step 1" in Figure 3. The application then uses the initialized AccountManager instance to interact with the account manager service. The AccountManager instance uses the service's proxy to make a method call towards the service. The proxy in turn uses the binder driver to forward the call to the remote service's stub which executes the remote method and return an answer the client using the reverse path stub-binder-proxy ("step 2" in Figure 3).

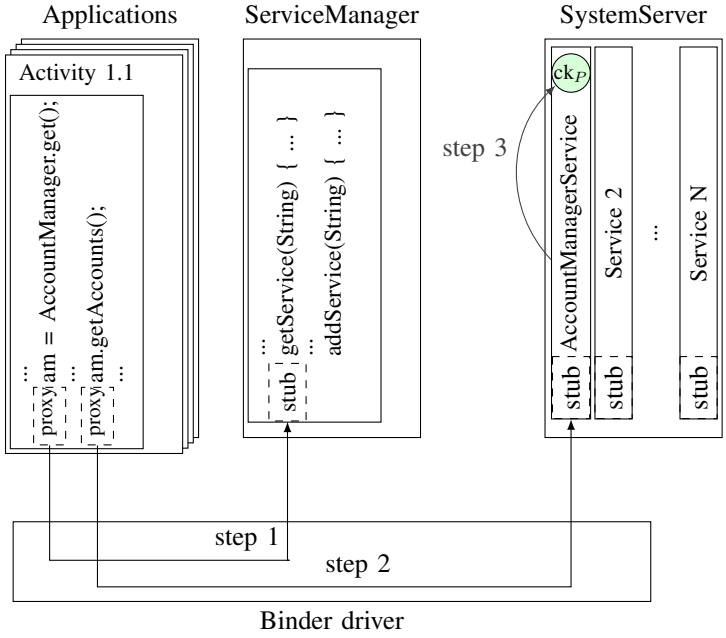

Figure 3. Android Binder

3) System Services and Permission Checks: System services are specific services running in the *system server*. They allow an application to access system resources. Those resources are protected by android permissions (step 3 in Figure 3). *Consequently, it is there that we extract permission enforcement points during the static analysis phase described in Section V.* At this point we know how applications are structured and how they communicate within the Android system. Note that android permissions are also checked in C++ services, content providers and when using intents but we do not consider those checks. This limitation is discussed in Section VI.

D. Permissions and Application Installation

This section explains that during installation, applications are given a unique identifier and sometimes specific low-level permissions in addition to high-level permissions. Android 2.2 declares in total 142 high-level permissions. Permissions are enforced at the framework level (ex: to read contact information an application must have the `READ_CONTACT` permission). However, some permissions

Permission	GID
BLUETOOTH_ADMIN	net_bt_admin
BLUETOOTH	net_bt
INTERNET	inet
CAMERA	camera
READ_LOGS	log
WRITE_EXTERNAL_STORAGE	sdcard_rw
ACCESS_CACHE_FILESYSTEM	cache
DIAGNOSTIC	input, diag

Table II
PERMISSIONS ASSOCIATED WITH A LOW-LEVEL GID

could also be associated with low-level group ID of specific resources (camera, bluetooth, ...). Those permissions, listed in table II, are indirectly enforced at the kernel level by checking group IDs (ex: to create a socket an application has to have the `INTERNET` permission to be in the `inet` group). Note that Android makes it possible for developers to write native code by using the Native Development Kit (NDK). However, system resources are either protected by a low-level permission or not accessible since access to the filesystem is restricted.

When installing an android application the official way (through the android market), the user has to approve (or reject) all the permissions the application has declared in its manifest. If all permissions are approved, the application is installed and mapped with the corresponding permissions. Moreover, it receives a device unique user id (UID) and a group id (GID) for permissions mapped with a low-level GID. For instance, an application Foo is given two GIDs `net_bt` and `inet` when associated with permissions `BLUETOOTH` and `INTERNET`, respectively. In other terms, the standard Unix ACL is used as an implementation means for checking high-level permissions.

In Section III, we have defined a generic model and methodology to generate a matrix M which maps entry points of a framework$_{\mathcal{F}}$ to permissions. We have seen in Section IV that the Android system fits in the model and contains a framework corresponding to $_{\mathcal{F}}$. The next Section presents the static analysis methodology to extract M from the Android framework$_{\mathcal{F}}$ and to infer the list of required permissions (as opposed to declared permissions) for an Android application.

V. STATIC ANALYSIS CODE FOR ANDROID

Our approach to detecting permission gaps which was presented in III is implemented with two tools. One extracts from a permission-based framework a binary matrix that maps framework methods to permissions, we call it the *mapper*. The other extracts from application code the list of framework methods used, we call it the *sniffer*. In COPES, both tools are designed as static analyses.

Implementing both tools was much more difficult than expected. In other terms, there was a significant gap between the regularity and the conciseness of the approach presented in III and the actual implementation. We came across different technical issues for which we had to find creative solutions. This section presents the most important ones in order to 1) enable other researchers to replicate our results, and 2) facilitate the implementation of the approach for another platform.

A. Choosing a bytecode manipulation toolkit

We had to write the mapper and the sniffer on top of two different toolkits: the mapper uses the Soot analysis framework developed at McGill University [25]; the sniffer uses the ASM framework [3]. We had to use two different toolkits for the following reasons. On the one hand, the code of the framework is open-source and written in Java, which is perfectly appropriate for an analysis using Soot. On the other hand, the application bytecode (we do not have the source code of applications) is available as Dalvik bytecode, and this bytecode is transformed to Java bytecode using a tool called "ded" developed at Penn State University[4]. Unfortunately, the resulting bytecode is often not compatible with Soot for obscure reasons. The lower-level API of ASM enabled us to overcome these problems.

B. Generating the Call Graph*S* of a Framework

Generally, a static analysis has a unique entry point to build a call graph: the "main" method of a program. In the case of a framework, there is no such thing as a main. Hence, we had to generate not one call graph but N call graph where N is the number of different entry points.

Solution: For every public class of the framework (for Android, `android.*` and `com.android.*`), we create a fake main, consisting of one instance on which all framework methods are called. Then we run the Soot call graph analysis Spark [16] on each of the generated "fake" entry points.

C. Extracting Actual Checked Permission Names in Permission Enforcement Points

Permission Enforcement Points in Android are method calls to certain method of classes `Context` and `ContextWrapper` (for instance method `checkPermission`). While those method calls can easily be resolved statically, the actual permission(s) that are checked are dynamically set by a String parameter or worse, an array of strings. Thus, when a check permission system method is found in the call graph, the exact permission(s) has(have) to be extracted.

Solution: We made a Soot plugin which finds PEPs and extract the corresponding permission(s). This plugin performs an intra-method analysis and manages the following scenarios: either (1) the permission is directly given as a string as a parameter, or (2) the permission is stored in a variable which is given as a parameter, or (3) an array

[4]http://siis.cse.psu.edu/ded/

is initialized with several permissions and is given as a parameter. In every case we do a backward analysis of the method's bytecode using Soot's Unit Graphs which describe relations among statements of a method. In the case where only one permission is given to the method, the first statement in the unit graph containing a reference to a valid Android permission String is extracted and the permission added to the list of the permissions needed by the method under analysis. In case of an array, all permissions of references to Android permission Strings are added to the list.

D. Handling Inter-Process Communication through the Binder

Static analysis can not handle call to services since they are done again dynamically through the binder (see IV). Consequently, the resulting call graphs are incomplete.

Solution: Our key insight is that the binding uses a lookup table that is instantiated once at runtime. We intercepted this lookup table and use it in a Soot plugin to redirect every proxy call to the concrete class which implements the service. In other terms, we feed the call graph engine with this domain specific information that it does not know.

E. Service Identity Inversion

In Android, services can call other services either with the identity of the initial caller (by default) or with the identity of the service itself. In the later case, remote calls are within `clearIdentity()` and `restoreIdentity()` method calls. When using the service identity, the permission checks are not done against the caller's declared permissions, but against the service's declared permissions. This mechanism is sometimes used for privilege escalation attacks. In our context, this mechanism yield too many inferred permissions. For instance, let us assume that service A requires permission θ which is not declared by service B, if B calls A with the identity of A itself, there is no reason to add θ in the list of required permissions

Solution: We analyze the call graphs to find permission checks that occur between calls to `clearIdentity()` and `restoreIdentity()`. Those PEPs are discarded.

F. Reflection in the Framework

If the framework uses reflection, then the call graph construction is incomplete by construction.

Solution:

Fortunately, the Android framework uses reflection in only 7 classes. We manually analyzed their source code. Five of those classes are debugging classes. The *View* class uses reflection for handling animations. Finally, the *VCardComposer* uses reflection in a branch that is only executed for testing purpose. In all cases, the code is not related to system resources hence no permission checks at all are done. This does not impact the static analysis of the Android framework.

G. Dynamic Class Loading in the Framework

The Java language has the possibility to load classes dynamically. When used this features makes static analysis impossible since the loaded class is only known at runtime.

Solution:

We found that eight classes of the Android system are using the `loadClass` method. After manual check, six of them are system management classes and either are not linked to permission checks (ex: instrumenting an application) or have to be accessed through a service. Two are related to the *webkit* packaged. They are used in the `LoadFile` and `PluginManager` classes. In both cases, permissions are checked *before* loading classes, and not inside the loaded classes. Thus, there is no missed permission enforcement points either.

H. Recapitulation

We have presented the core technical issues we encountered while implementing our approach. We think that those problems may arise in other permission-based platforms than Android, and that identifying them and their solutions can be of great help for future work. Last not but not least, those points are crucial for replication of our results.

VI. Evaluation

This section presents an evaluation of our approach. First, we discuss the permission map extracted by static analysis, and compare it to the map extracted by Felt et al.[13] by testing. Then, we show that our approach actually detects permission gaps in real applications published on an application store.

A. Evaluation of Extracted Permission Map M

We ran our automatic tool on the Android v2.2 bytecode and obtained a matrix M composed of 3957 methods which check at least one of 96 high-level permissions. Since one method checks at most an handful methods, this binary matrix is very sparse, it mostly contains zeros and a few ones (the number of permission checks). We find that the Android framework v2.2 contains a total 4852 permission checks. The computation of the full matrix is done with two hours on a Desktop Dell dual quad-core 2.4GHz with 24 Go RAM.

Let us now compares our map with Felt et al.'s one. They have 1282 methods which check permissions in their table. We have more than the double number of methods because in our analysis we even take into account methods which can not be reached directly by applications (ex: internal methods of services). This has no impact on the correctness of when inferring permission gaps in Android applications, our table is only more exhaustive. We have similar results (same permission set for a specific method signature) for 918 methods. However, the table differs for 393 methods. Let us now discuss this discrepancy.

Case #1: We find more permission checks In our matrix, there is one ore more additional permissions for 146 methods (1 additional permission for 127 methods, 2 for 13 methods, 3 for 5 and 4 for 1 method). Those permissions are either never checked or checked within a specific *environment context*. This result is typical when comparing a static analysis approach against a testing one: static analysis sometimes suffers from analyzing all code (including debugging and dead code), but is strong at abstracting over input data. For instance, our static analysis yields permissions which are in the call graph (mostly due to debugging code), yet never checked in production. Hence, when Felt et al. simulate the production environment, they do not find those checks.

On the contrary, we are able to find permissions that are checked within specific contexts that were not taken into account by the generated tests. For instance, `MOUNT_UNMOUNT_FILESYSTEMS` is only checked for method `MountService.shutdown()` if the media (storage device) is "present not mounted and shared via USB mass storage". Another permission, `READ_PHONE_STATE` is needed for method `CallerInfo.getCallerId()` only if the phone number passed in parameter is the voice mail number. This test case was not generated by Felt's testing approach. In real applications, test generation techniques can not guarantee a comprehensive exploration of the input space.

Case #2: We find less permission checks As already mentioned, we only analyze the permission checks that are at the level of the Java framework. Consequently, our approach misses permission checks performed at the level of C++ libraries related to services, content providers or intents. In total, 247 methods in our matrix miss at least one permission. For instance `MODIFY_AUDIO_SETTINGS` is checked in a C++ service and `WRITE_SECURE_SETTINGS` is associated with content providers.

Recapitulation Those results highlight the key conceptual differences between static analysis and testing in the context of permission inference. We think that that the static analysis approach is complementary to the testing approach. Indeed, the testing approach yields an under-approximation which misses some permission checks whereas the static analysis approach yields an over-approximation in which those missing permission checks are found. Using both approaches in collaboration would enable developers to obtain a lower and a upper bound of the permission gap.

B. Permission Gaps in Real Applications

When analyzing permissions on Android appplications, we want to guarantee that our inferred permissions are correct. Hence, we do not claim inferring low-level permissions. Furthermore, our comparison with Felt's table has shown that 15 high-level permissions are checked in the Java framework but not only. In the following, we discard those permissions, and from a initial set of 96 permissions, we infer permission gaps related to 71 permissions (96-15). Hence we claim that we never miss a required permission in our inference (given the assumptions discussed in V).

Permission set		Number of Methods†
#Methods in [13]		1282
#Methods (us)		3957
Identical		918 (71.6%)
Different	we find less permission checks	247 (19.2%)
	we find more permission checks	146 (11.4%)

Table III
COMPARISON WITH [13]. THE DISCREPANCY IS DUE TO THE CONCEPTUAL DIFFERENCES BETWEEN STATIC ANALYSIS AND TESTING.

† the total is 102.2% because there are about 2% of methods which have, at the same time, missing or additional permissions.

We ran our tool on 1329 android applications from an alternative android market[5]. From those, the 587 that are using reflection and/or class loading are not checked. On the 742 remaining applications, 94 are declaring one or more permissions which they do not use. Consequently, we identify a permission gap for 94 Android applications. We define the "area of the attack surface" (related to permission gaps) as the number of unnecessary permission. In all, among applications suffering from a permission gap, 76.6% have an attack surface of 1 permission, 19.2% have an attack surface of 2 permissions, 2,1% of 3 permissions and also 2,1% of 4 permissions.

Table IV represents the top ten declared but not used permissions among all 94 applications. The associate frequency corresponds to the ratio of the number of applications which declare but do not use the permission, over the number of applications which declare the permission. For example, 70.59% of the applications declaring permission `ACCESS_LOCATION_EXTRA_COMMANDS`, do not actually use it.

We used the online tool from Felt et al. available at http://android-permissions.org/index.html to analyze the same set of applications. Their results are similar as ours. All applications suffering from a permission gap are also found by Felt's tool. This means that the divergence in inferred map are not crucial for this dataset.

To sum up, those results show that permission gaps exists, and that our tool allows developers to correct the declared permission list in order to reduce the attack surface of permission-based software.

VII. RELATED WORK

Indeed, we have presented an approach to reduce the attack surface of Android applications. The concept of "attack surface" was introduced by Manadhata and colleagues [17], it describes all manners *in which an adversary can enter the system and potentially cause damage*. This paper describes a method to identify the attack surface of Android

[5] www.freewarelovers.com/android

Permission	Wrong Usage
ACCESS_LOCATION_EXTRA_COMMANDS	70.59%
BATTERY_STATS	62.50%
ACCESS_MOCK_LOCATION	38.46%
SET_ORIENTATION	35.71%
GET_TASKS	10.26%
ACCESS_WIFI_STATE	7.89%
WAKE_LOCK	5.13%
VIBRATE	3.61%
ACCESS_COARSE_LOCATION	2.84%
ACCESS_FINE_LOCATION	1.47%

Table IV
TOP TEN OF DECLARED BUT NOT USED PERMISSIONS

applications, which is a important research challenge given the sheer popularity of the Android platform. In the context of Android, reducing the attack surface is adhering to the principle of least privileges introduced by Saltzer [22].

The Android security model has been described as much in the gray literature [9, 23] as in the official documentation [1]. Different kinds of issues have been studied such as social engineering attacks [15], collusion attacks [18], privacy leaks [14] and privilege escalation attacks [7, 12]. In contrast, this paper does not describe a particular weakness but rather a software engineering approach to reduce to potential vulnerabilities.

However, we are not describing a new security model for Android as done by [4, 6, 8, 19, 20]. For instance, Quire [8] maintains at runtime the call chain and data provenance of requests to prevent certain kinds of attacks. In this paper, we do not modify the existing Android security model and we devise an approach to mitigate its intrinsic problems.

Also, different authors empirically explored the usage of the Android model. For instance, Barrera et al. [2] presented an empirical study on how permissions are used. In particular, they used visualizing techniques such as self-organizing maps to identify patterns of permissions depending on the application domain, and patterns of permission grouping. Other empirical studies include Felt's one [11] on the effectiveness of the permission model, and Roesner's one [21] on how users react to permission-based systems. While our paper also contains an empirical part, it is also operational because we devise an operational software engineering approach to tame permission-based security models in general and Android's one in particular.

Enck et al [10] presented an approach to detect dangerous permissions and malicious permission groups. They devised a language to express rules which are expressed by security experts. Rules that do not hold at installation time indicate a potential security problem hence a high attack surface. Our goal is different, we don't aim at identifying risks identified from experts, but to identify the gap between the application's permission specification and the actual usage of platform resources and services. Contrary to [10], our approach is fully automated and does not involve an expert in the process.

Finally, Felt et al. [13] concurrently worked on the same topic as this paper. They published a very first version of the map between developer's resources (e.g. API calls) and permissions. Interestingly, we took two completely different approaches to identify the map: while they use testing, we use static analysis. As a result, our work validates most of their results although we found several discrepancies that we discussed in details in Section VI. But the key difference is that our approach is fully automated while theirs requires manually providing testing "seeds" (such as input values). However, in the presence of reflection, their approach works better if the tests are appropriate. Hence, we consider that both approaches are complementary, both at the conceptual level for permission-based architectures, and concretely for the reverse-engineering Android permissions.

VIII. CONCLUSIONS AND PERSPECTIVES

In this paper, we have presented a generic approach to reduce the attack surface of permission-based software. We have extensively discussed the problematic consequences of having more permissions than necessary and showed that the problem can be mitigated using static analysis. The approach has been fully implemented for Android, a permission-based platform for mobile devices. Our prototype implementation is able to automatically find 4852 permission checks in the Android framework. In a permission-based framework, all those checks have to be documented, hence our approach does a significant job in achieving this task in a systematic manner. For end-user applications, our evaluation revealed that 94/742 crawled applications from an application store for Android indeed suffer from permission gaps. We have also shown that our static analysis based approach is complementary to concurrent work [13] based on testing.

The security architecture of permission based software in general and Android in particular is complex. In this paper, we abstracted over several characteristics of the platform such as low-level permissions. We are now working on a modular approach that would be able to analyze native code and bytecode and to combine the permission information from both. Furthermore, we are exploring how to express permission enforcement as a cross cutting concern, in order to automatically add or remove permission enforcement points at the level of application or the framework, according to the security specification.

ACKNOWLEDGEMENTS

The present project is supported by the National Research Fund, Luxembourg.

REFERENCES

[1] The android developer's guide. http://developer.android.com/guide/index.html, 2011. Last-accessed: 2011-09-10.

[2] David Barrera, Hilmi Günes Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *ACM Conference on Computer and Communications Security (CCS 2010)*, pages 73–84, Chicago, Illinois, USA, October 4-8, 2010.

[3] Eric Bruneton. ASM 3.0, a Java bytecode engineering library. http://download.forge.objectweb.org/asm/asm-guide.pdf, 2007.

[4] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, Apr 2011.

[5] Canalys. Android takes almost 50% share of worldwide smart phone market, 2011. http://www.canalys.com/static/press_release/2011/canalys-press-release-010811-android-takes-almost-50-share-worldwide-smart-phone-market_0.pdf.

[6] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: context-related policy enforcement for android. In *Proceedings of the 13th international conference on Information security*, ISC'10, pages 331–345, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, August 2011.

[9] W. Enck and FOO. Understanding android security. 2009.

[10] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[11] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

[12] Adrienne Porter Felt, Helen Wang, Alex Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[13] A.P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. Technical Report UCB/EECS-2011-48, University of California, Berkeley, 2011.

[14] Clint Gibler, Jonathan Crussel, Jeremy Erickson, and Hao Chen. Androidleaks detecting privacy leaks in android applications. Technical report, UC Davis, 2011.

[15] Stefanie Hoffman. Zeus banking trojan variant attacks android smartphones. *CRN*, 2011.

[16] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[17] P.K. Manadhata and J.M. Wing. An attack surface metric. *Software Engineering, IEEE Transactions on*, 37(3):371 –386, may-june 2011.

[18] Claudio Marforio, Aurélien Francillon, and Srdjan Čapkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical Report 724, ETH Zurich, April 2011.

[19] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[20] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDanie. Semantically rich application-centric security in androi. *Journal of Security and Communication Networks*, 2011.

[21] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. Technical Report MSR-TR-2011-91, Microsoft Research, 2011.

[22] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, 1975.

[23] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, and Shlomi Dolev. Google android: A state-of-the-art review of security mechanisms. *CoRR*, abs/0912.5101, 2009.

[24] L. Masinter T. Berners Lee, R. Fielding. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396, August 1998.

[25] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Etienne Gagnon Patrick Lam, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[26] Wikipedia. Bipartite graph. http://en.wikipedia.org/wiki/Bipartite_graph.