Determination and Enforcement of Least-Privilege Architecture in Android

Mahmoud Hammad
Department of Informatics
University of California, Irvine
hammadm@uci.edu

Hamid Bagheri
Dept. of Computer Science and Engineering
University of Nebraska, Lincoln
bagheri@unl.edu

Sam Malek
Department of Informatics
University of California, Irvine
malek@uci.edu

***Abstract*—Modern mobile platforms rely on a permission model to guard the system's resources and apps. In Android, since the permissions are granted at the granularity of apps, and all components belonging to an app inherit those permissions, an app's components are typically over-privileged, i.e., components are granted more privileges than they need to complete their tasks. Systematic violation of *least-privilege principle* in Android has shown to be the root cause of many security vulnerabilities. To mitigate this issue, we have developed DELDROID, an automated system for determination of least privilege architecture in Android and its enforcement at runtime. A key contribution of our approach is the ability to limit the privileges granted to apps without the need to modify them. DELDROID utilizes static program analysis techniques to extract the exact privileges each component needs for providing its functionality. A *Multiple-Domain Matrix* representation of the system's architecture is then used to automatically analyze the security posture of the system and derive its least-privilege architecture. Our experiments on hundreds of real-world apps corroborate DELDROID's ability in effectively establishing the least-privilege architecture and its benefits in alleviating the security threats.**

I. INTRODUCTION

Modern mobile platforms, such as Android, rely on a permission-based model for controlling the resources that each app is allowed to access. Permissions are often granted to an app at the discretion of end user, who makes a decision based on its perceived trustworthiness and expected functionality.

Android's permission-based access control model, however, has shown to be ineffective in protecting system resources and apps from security attacks [15]. All components of an Android app inherit the permissions granted to the app, regardless of whether they need those permissions or not. As a result, a malicious component inside an app, such as a third-party library, can leverage privileges meant for other components for nefarious purposes [29]. Moreover, by default, a component in Android has significant leeway in terms of the components it can communicate with, both within and outside of its parent app. The over-privileged nature of components in Android is the root cause of various security attacks [15], [29], [11]. These kinds of attacks cannot be prevented by the platform at the moment, as they do not violate the security mechanisms supplied by Android.

To systematically thwart these threats, we have developed DELDROID[1], an automated system for determination of *least-privilege architecture* (LP architecture) in Android and its enforcement at runtime. An LP architecture is one in which the components are only granted the privileges that they require for providing their functionality [37]. An LP architecture, thus, reduces the risk of an Android system being compromised by limiting its attacks surface. In addition, when a component is compromised, the impact is localized within the scope of that component. A smaller attack surface also facilitates both manual and automated means of inspecting the system's security attributes.

Establishing the least privilege architecture is quite challenging as it demands mediation of all conceivable channels through which a component may interact with components within and outside its parent app, as well as the underlying system resources. DELDROID leverages static program analysis to automatically identify the architectural elements comprising an Android system, as well as the inter-component communication and resource-access privileges each component needs to provide its functionality. It then uses a *Multiple-Domain Matrix* (MDM) [24] to represent and derive the LP architecture for the system. MDM provides an elegant, yet compact, representation of all relationships between principal elements, such as components and permissions, in a system. DELDROID further allows a security expert to modify the architecture as needed to establish the proper privileges for each component. Finally, DELDROID enforces automatically obtained or expert-supplied LP architecture at runtime, thus ensuring components are not able to obtain more privileges than that prescribed by the architecture.

DELDROID can be used to limit the levels of access

[1]The name is intended to abbreviate "**d**etermination and **e**nforcement of **l**east privilege architecture in An**Droid**".

978-1-5090-5729-0/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSA.2017.18

available to an app and its components without modification of their implementation logic, thus allowing our approach to be applied to all existing Android apps. Our evaluation of DELDROID using hundreds of real-world apps corroborates its ability in significantly reducing the attack surface of Android systems and thwarting security attacks that would have succeeded otherwise.

The remainder of this paper is structured as follows. Section II provides a background on Android's access control model. Section III presents an Android system to motivate the research. Section IV describes DELDROID, while Section V describes its implementation. The evaluation results are presented in Section VI. Finally, the paper concludes with an overview of the related literature and areas of future research.

II. ANDROID'S ACCESS CONTROL MODEL

This section provides an in-depth description of the Android's access control model to help the reader follow the discussions that ensue. An Android system consists of a set of apps running on a device. Each app in Android consists of a set of software components. There are two kinds of privileges a component has: *inter-component communication (ICC) privilege*, allowing a component to communicate with other components in the same or different app, and *resource access privilege*, allowing a component to access the system resources, such as GPS, camera, telephony, etc.

A. Over-Privileged Inter-Component Communication

Each Android app includes a mandatory configuration file, called *manifest*. It specifies, among other things, the principal components that constitute the application, including their types and capabilities, as well as required and enforced permissions. Components are basic logical building blocks of apps. Android defines four types of components: *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*. The components in Android mainly communicate by means of *Intent* messages. An Intent can be either *explicit*, in which case the *target component* is specified, or *implicit*, in which case the *action* to be performed is specified. *Intent Filters* are the provided interfaces of a component and define the actions performed by the component. An implicit Intent is delivered to a component if the action specified in the Intent matches that specified in the component's Intent Filter.

Android's ICC mechanism leads to over-privileged architectures, where components needlessly have the ability to send Intent messages to invoke services of many other components within and outside their parent apps, and receive a variety of Intent messages implicitly exchanged in the system. A component is allowed to communicate with (1) all components in its parent app, (2) protected components in other apps as long as its parent app has the required permissions, and (3) any public (exported) component in other apps. A component is public if its *VISIBLE* attribute is set to true in the manifest file or declares at least one Intent Filter. Many developers are not aware of the fact that by specifying an Intent Filter for a component, Android by default makes that component public, thus allowing components from other apps to invoke its interfaces [15]. Inter-app communication (IAC) privileges are thus often granted implicitly. Finally, a component does not require a permission to specify an Intent Filter with arbitrary action, thereby allowing that component to receive all implicit Intents exchanged in the system with the specified action.

The over-privileged ICC mechanisms in Android are known to be the root cause of many security attacks [15], [29], [11]. Moreover, comprehending the security posture of an Android system in light of this privilege management scheme is rather tedious and error prone for a security architect.

B. Over-Privileged Resource Access

Android contains a plethora of sensitive system resources (e.g., GPS, camera, account manager, power manager) accessed by obtaining a handle to a system-level, long-running service (e.g., location service, camera service, account service, power manager service). System services are launched by `com.android.server.SystemServer` service, which is started at the boot time of the Android operating system. To use a system service, a component should have the appropriate permission that guards the service. For example, to track the user's location, a component needs to obtain a handle to the location service, which requires the location permission (either `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`).

Permissions are the cornerstone of the Android security model. The permissions stated in the app manifest enable secure access to sensitive resources. However, a permission granted to an app transfers to all of the components in the app. Android's coarse-grained permission model violates the principle of least privilege [14], [35], as often not all components of an app need access to the same sensitive resources. The shortcomings of Android's permission model have been widely discussed in the literature [34], [19], [18], and shown to be the root cause of various security attacks, most notably privilege escalation [17], [20].

III. ILLUSTRATIVE EXAMPLE

To further motivate our research and illustrate our approach, we provide an example of a malicious component that employs the extra privileges afforded by Android to launch two security attacks: information leakage through

Listing 1. Vulnerable component, Sender Service, sends a text message.

```
1  public class Sender extends Service {
2     ...
3     public int onStartCommand(Intent intent, int flags, int startId){
4          //if (checkCallingPermission("android.permission.SEND_SMS") == PackageManager.PERMISSION_GRANTED) {
5             String phoneNumber = intent.getStringExtra("PHONE_NUMBER");
6             String msg = intent.getStringExtra("MSG_CONTENT");
7             SmsManager smsManager = SmsManager.getDefault();
8             smsManager.sendTextMessage(phoneNumber, null, msg, null, null);
9          //}
10     ...
```

Listing 2. Malicious component, LevelUp Service, uses dynamic class loading to hide its malicious behavior.

```
1  public class LevelUp extends Service {
2     ...
3     public int onStartCommand(Intent intent, int flags, int startId){
4       ...
5       loadCode();
6     }
7     public void loadCode(){
8       // read a jar file that contains classes.dex file.
9       String jarPath=Environment.getExternalStorageDirectory().getAbsolutePath()+"/Download/hiddenCode.jar";
10      //load the code
11      DexClassLoader mDexClassLoader = new DexClassLoader(jarPath, getDir("dex", MODE_PRIVATE).
            getAbsolutePath(),null, getClass().getClassLoader());
12      //use java reflection to load a class and call its method
13      Class<?> loadedClass = mDexClassLoader.loadClass("HiddenBehavior");
14      Method methodGetIntent = loadedClass.getMethod("getIntent", android.content.Context.class);
15      Object object = loadedClass.newInstance();
16      Intent intent = (Intent) methodGetIntent.invoke(object, LevelUp.this);
17      startService(intent);
18      ...
```

Listing 3. Code downloaded after initial installation of app.

```
1  public class HiddenBehavior {
2      ...
3      public Intent getIntent(Context context){
4          LocationManager locMgr = (LocationManager) context.getSystemService(Context.LOCATION_SERVICE);
5          Location loc = locMgr.getLastKnownLocation(LocationManager.GPS_PROVIDER);
6          String msg = loc.getLatitude()+","+loc.getLongitude();
7          Intent i = new Intent("SEND_SMS");
8          i.putExtra("PHONE_NUMBER", phoneNumber);
9          i.putExtra("MSG_CONTENT", msg);
10         return i;
11     }
12 }
```

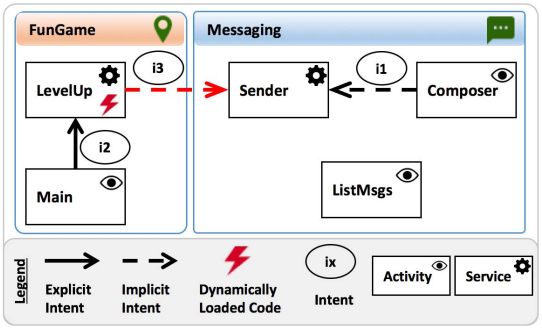


Figure 1. Component-based architecture of a vulnerable Android system.

hidden code [29], [15], and privilege escalation [20], [11].

Figure 1 shows an Android system with two apps: `FunGame` and `Messaging`. The `Messaging` app contains three components. The `ListMsgs` Activity lists all previously received messages. The `Composer` Activity allows a user to compose and send text messages using the `Sender` Service running in the background. Sending text messages requires SMS permission. The `Messaging` app has this permission and hence all its components have it as well. Listing 1 shows part of the `Sender`'s program logic for sending text messages.

`LevelUp` is a Service in `FunGame`, a malicious Android game app, which once started, via the `Main` Activity, leverages dynamic class loading feature of Android to load a malicious behavior from an external JAR file placed at the location specified on line 9 of Listing 2. The dynamically loaded code allows `LevelUp` to communicate with the `Sender` Service as shown in Listing 2. On line 11 of Listing 2, `LevelUp` instantiates a `DexClassLoader` object and uses it to load the DEX (Dalvik Executable) file contained in the JAR file. Using Java reflection at line 13 of Listing 2, the `mDexClassLoader` object loads a class called `HiddenBehavior` and invokes `getIntent` method at line 16 of Listing 2. This method returns an implicit Intent, which `LevelUp` uses to communicate with `Sender`, as shown in line 17 of Listing 2.

Listing 3 shows the implementation of `getIntent` method in the `HiddenBehavior` class. On line 4, `getIntent` obtains a reference to the `Location Manager`, a service that provides periodic updates of the

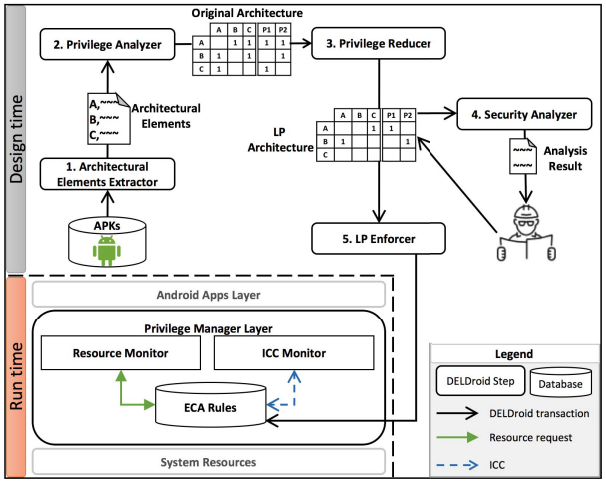


Figure 2. Overview of DELDROID.

device's geographical location. On line 5, the `Location Manager` is used to get the user's last known location. Finally, in lines 7-9, it creates an implicit Intent and adds a phone number and the user's location as the extra payload of the Intent. This code is compiled to a DEX format and archived in a JAR file using the *dx* tool, a tool that generates Android bytecode from *.class* files. The JAR file could be downloaded by the malicious app after installation.

On lines 5 and 6 of Listing 1, the `Sender` service extracts the phone number and the location information from the received Intent, respectively. The extracted information is used in line 8 to send a text message. The `Sender` component is vulnerable to a privilege escalation attack since it performs a privileged task, sending text messages, without checking if the caller component has the required SMS permission to perform the task. An example of such a check is shown in line 4 of Listing 1, but in this example it is commented.

The illustrative example described in this section allows `LevelUp` to hide its malicious behavior to exploit a privilege escalation vulnerability and leak the user's sensitive information (i.e., user's location) via text messaging without having the SMS permission. This kind of an attack is neither effectively detectable through static program analysis, since the malicious behavior is downloaded after installation, nor through dynamic program analysis, as malicious apps often incorporate complicated evasion tactics (e.g., timing-bombs [16]). We show how through establishment of an LP architecture, DELDROID can effectively mitigate such threats.

IV. APPROACH

As depicted in Figure 2, DELDROID consists of five steps. The rest of this section presents each step in detail.

A. Step 1: Architectural Elements Extractor

To obtain the system's architecture, we first need to determine the principal components that constitute the system, their properties, communication interfaces, and permission usages. Such information is obtained from two sources, an app's manifest file and its bytecode.

DELDROID utilizes *APKtool* [3], a reverse engineering tool for Android APK files, to recover an app's manifest file. By simply parsing the manifest file, we can extract certain information readily available about the components comprising an app. Table I partially shows the extracted information corresponding to our running example (recall Section III). The *Type* column represents the particular type of a component, which could be either Activity, Service, Broadcast Receiver, or Content Provider. The *Exported* column indicates whether a component can be launched from outside its hosting app or not. The *Intent Filter* column shows the interfaces provided by a component. Finally, the *Granted* column shows the permissions requested by an app, and subsequently granted by Android to all of its component.

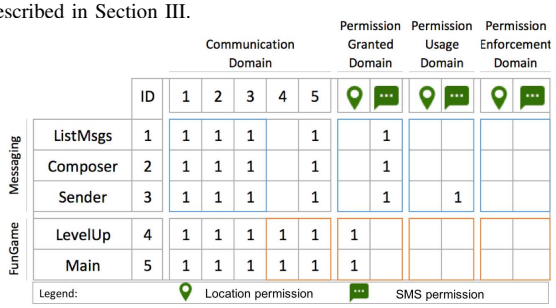
Not all information about an app can be obtained from its manifest file. For example, Broadcast Receivers can be registered in code without declaring them in the manifest file. Components can also programmatically define Intent Filters in code. In addition, all ICCs are latent in the app's bytecode. Components can communicate with one another in two ways: (1) using Unified Resource Identifiers (URIs) to access the encapsulated data in Content Providers, and (2) by sending Intents, either explicitly or implicitly. DELDROID utilizes IC3 [28] to analyze each app in the system and extract such latent information from its bytecode. IC3 is the state-of-the-art static program analysis tool for Android. For each Intent in bytecode, DELDROID extracts the sender component, receiver component, action, categories, and data. Table I shows the remaining information collected in this way for our running example. Intent `i3` is not shown, since the program logic that creates that Intent is not initially part of the `FunGame` (recall Listing 2).

DELDROID also identifies the permissions actually used by components. These are the permissions that a component uses for (1) accessing a protected Content Provider, or (2) calling a protected API. For the former, we have created a mapping between protected Content Providers and the required permissions. For example, to read the contacts information from Android's Contacts Content Provider, a component needs `android.permission.READ_CONTACTS` permission. Since IC3 does not extract the permissions used through API calls, for the latter case, DELDROID leverages PScout permission map [10], one of the most recently updated and comprehensive permission maps available for the Android framework. It specifies map-

Table I
THE EXTRACTED ARCHITECTURAL ELEMENTS FOR THE ANDROID SYSTEM SHOWN IN FIGURE 1

ID	App	Component Name	Type	Exported	Intent Filter	Permissions			Intents
						Granted	Used	Enforced	
1	Messaging	ListMsgs	Activity	Yes		{SMS}			
2	Messaging	Composer	Activity	Yes		{SMS}			{i1}
3	Messaging	Sender	Service	Yes	SEND_SMS	{SMS}	{SMS}		
4	FunGame	LevelUp	Service	No		{Location}			
5	FunGame	Main	Activity	Yes	MAIN	{Location}			{i2}

Figure 3. The Original architecture derived from the Android system described in Section III.



pings between Android API calls/Intents and the permissions required to perform those calls. For example, `Sender` component in `Messaging` app uses the `sendTextMessage()` API for sending text messages (see line 8 of Listing 1), which requires SMS permission. We thus consider this to be a permission that is actually used by this component, as shown in the Used column of Table I

Finally, DELDROID builds on our prior work [11] to extract the permissions enforced by a component at two levels. While the coarse-grained permissions specified in the manifest file are enforced by the Android runtime environment over an entire component, it is possible to add permission checks, such as *checkCallingPermission*, throughout the code controlling access to specific parts of a component (see line 4 of Listing 1). DELDROID identifies both types of checks. Since the system of Figure 1 does not perform any checks (line 4 of Listing 1 is commented out), the corresponding column in Table I is empty.

B. Step 2: Privilege Analyzer

The next step is to derive the overall system architecture from the information obtained for individual components in the previous step. We call this the *Original* system architecture, as it represents the architecture of system if it were to be deployed on the official Android runtime environment. DELDROID models the system architecture as a Multiple-Domain Matrix (MDM) [24]. MDM provides an elegant representation of complex systems with multiple concerns (domains). Each concern is modeled as a Design-Structure Matrix (DSM) [36]—a simple matrix that captures the dependencies of one relationship type. MDM is formed by connecting the DSMs together. We capture four domains in an MDM to represent an Android system's architecture for the purpose of privilege analysis.

The communication domain shows all potential com ponent-to-component interactions. Each non-empty cell in this domain indicates the fact that the architecture of system allows for potential interaction between two components. Rows represent sender components; columns represent receiver components. Allowed communications are derived using the following rule.

Definition 1 (Allowed Communication)**.** *Let E be a set of all exported components, and c_1 and c_2 be two arbitrary components in the system. We say that c_1 can communicate with c_2, if either both components belong to the same app or c_2 is an exported component and c_1 is granted the permissions enforced by c_2:*

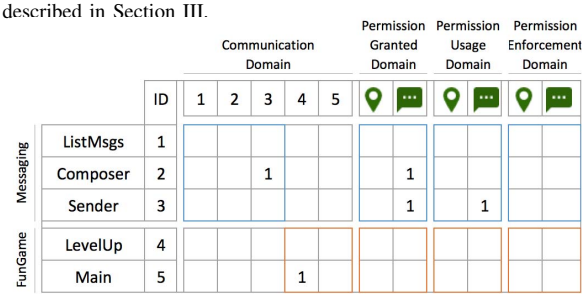
$communicate(c_1, c_2) \equiv (app_{c1} = app_{c2}) \vee (c_2 \in E \wedge enforced_{c2} \subseteq granted_{c1})$

Figure 3 shows the result of applying Definition 1 to Table I. According to the communication domain, components 1, 2, and 3 can communicate with one another because they belong to the same app, as well as component 5 since it is exported, but not component 4.

Note that the communication domain also includes interactions between the Android framework and components of third-party apps. Android provides over 230 protected broadcast Intents that can only be sent by the system to the registered components. For example, when a user installs an app, the system sends a broadcast Intent including the package name of the newly installed app to all components that listen to the PACKAGE_ADDED broadcast Intent action. Figure 3 shows no such interactions with the system, as no component in our running example is registered to receive protected broadcast Intents.

The three permission domains in the MDM model of Figure 3 represent the component-to-permission relationships. Each non-empty cell corresponds to a permission that is either (1) granted to a component, meaning that the component has that permission, as its hosting app has requested the permission in its manifest file, (2) used by a component, meaning that the component is actually making API calls or interacts with other apps that require

Figure 4. LP architecture determined from the Android system described in Section III.



the permission, or (3) enforced by a component, meaning that either the Android runtime environment or the component itself check the permission of callers (as you may recall from Section IV-A there are two ways of enforcing permissions in Android). The permission domains in the MDM are populated based on the information obtained in the first step (i.e., Granted, Used, and Enforced columns of Table I).

C. Step 3: Privilege Reducer

The Original architecture derived in the previous step clearly violates the principle of least privilege. This step aims to derive the LP architecture by granting only the privileges required by each component to fulfill its tasks.

DELDROID uses the extracted inter-component communications (information in the Intents column of Table I) to determine the communication privileges that are needed for each component to provide its functionality, and removes communication privileges that are unnecessary. For instance, as shown in Figure 4, the LP architecture allows the `Composer` component to communicate with the `Sender` component to send text messages (indicated by "1" in row 2, column 3). On the other hand, the LP architecture prohibits the `LevelUp` component to communicate with the `Sender` component.

Furthermore, DELDROID reduces the granted permissions for each component in the Permission Granted Domain of the LP architecture using the following rule:
Definition 2 (Required Permission)**.** *Let c_1 be a component, and $used_{c1}$ be a set of permissions directly used by component c_1. We define the required permissions for c_1 as permissions either directly used by c_1 or used by component c_2 with which c_1 communicates:*
$requiredPermissions_{c1} = \{p : Permission \mid \exists\ c_2 : Component \bullet p \in used_{c1} \vee communicate(c_1, c_2) \wedge p \in used_{c2} \wedge p \in granted_{c1}\}$

According to Definition 2, a component legitimately needs a permission in two cases: 1) the permission is directly used by the component through, among other things, making protected API calls; 2) another component with which the given component is interacting is using that permission. The latter may be a legitimate case, since a component that uses a permission may require the calling component to also have that permission. In fact, failing to check if the calling component has the necessary permission may result in a privilege escalation attack, as discussed in the next section.

In our running example, DELDROID determines that the `Sender` component has a legitimate reason to hold the SMS permission, since it uses it. The `Composer` component also has a legitimate reason to hold the *SMS* permission, since the app it belongs to has that permission and it communicates with the `Sender` component that uses that permission. `ListMsgs`, however, does not need the SMS permission, since it neither uses it nor does it communicate with a component that uses that permission. Similarly, the `LevelUp` and `Main` components do not use the Location permission, and thus do not have a legitimate reason to hold it.

Finally, a security architect can adjust the resulting architecture by manually granting and revoking permissions in the MDM. For example, a security architect can revise the privileges granted to apps and their components based on their reputation. This capability could also be useful in a forward-engineering setting, where an Android system is developed from scratch.

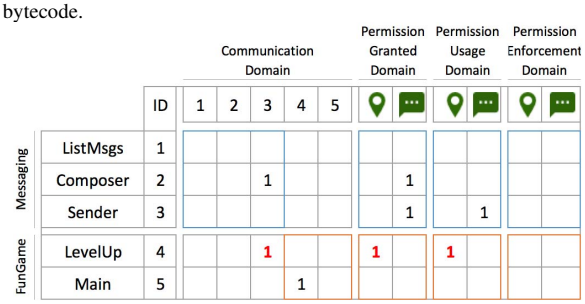
D. Step 4: Security Analyzer

The previous sections present derivation of the LP architecture for an Android system captured in an MDM. Here, we describe how the resulting architecture can be used to effectively perform security analysis of Android apps. In particular, we focus on one of the most prominent vulnerabilities due to the interaction of multiple apps, i.e., privilege escalation, defined as follows:
Definition 3 (Privilege Escalation)**.** *Let p be a permission, c_m be a component that does not hold p, and c_v be a component that holds and uses p but does not enforce (check) the components that may be using its services also hold p. In the privilege escalation attack, c_m is able to indirectly obtain p by interacting with c_v.*
$communicate(c_m, c_v) \wedge p \in used_{cv} \wedge p \notin granted_{cm} \wedge p \notin enforced_{cv}$

According to Definition 3, in privilege escalation, a malicious app is able to *indirectly* perform a privileged task, without having a permission to do so, by interacting with a component that possesses the permission. By applying the privilege escalation rule to the MDM representation of the system's architecture, DELDROID identifies communications that may result in privilege escalation attack.

To illustrate this, let us assume that instead of `LevelUp` using dynamic class loading to communicate with the `Sender` component, the logic for this interaction is part of the component's implementation analyzed by DELDROID. The LP architecture for such

Figure 5. The LP architecture for an alternative system, where the communication between Sender and LevelUp is part of the app's initial bytecode.



an alternative system is shown in Figure 5. Applying the privilege escalation rule to the LP architecture of Figure 5 reveals that `LevelUp` is not granted the SMS permission, and communicates with the `Sender` that uses the SMS permission without enforcing it. As a result, this interaction is potentially a privilege escalation attack, and DELDROID raises a warning for further inspection.

E. Step 5: LP Enforcer

This step regulates component interactions by enforcing the LP architecture at runtime. DELDROID transforms the LP architecture to a set of Event-Condition-Action (ECA) rules suitable for rapid evaluation as the system executes. It then relies on two components, i.e., ICC Monitor and Resource Monitor, within the Privilege Manager layer that we have added to the Android runtime environment, as shown in Figure 2.

1) ICC Monitor: This component extends the capabilities of the Android framework by intercepting each ICC transaction passed to the *ActivityManager*—an Android component that administers the ICC transactions—to check whether the transaction is allowed to run or not. Specifically, DELDROID extends the *ActivityManager* to send the ICC transaction's information to the *ICC Monitor* component and executes the action provided by *ICC Monitor*. In case, an ICC is prevented, *ICC Monitor* records the transaction for further inspection by a security analyst.

For example, the following ECA rule is produced, from the LP architecture shown in Figure 4, to prevent the `LevelUp` component from communicating with the `Sender` component:
Event: $i \in ICC$ occurs
Condition: $i.senderPkg$ = `FunGame` $\wedge$ $i.senderComp$ = `LevelUp` $\wedge$ $i.receiverPkg$ = `Messaging`
Action: *prevent*

2) Resource Monitor: As we explained in Section II-B, components need permissions to access various system resources. Such system resources are accessed via the *Context* component, an Android component that holds information about the application environment and controls access to resources. DELDROID modifies *Context* to extract information from each resource access request, and passes it to the *Resource Monitor* to check whether the requester is allowed to access the requested service.

As a concrete example, the following ECA rule is produced, from the LP architecture shown in Figure 4, to prevent `LevelUp` from requesting the location service:
Event: $resourceaccessrequest$
Condition: $requester$ = `LevelUp` $\wedge$ $service$ = `Context.LOCATION_SERVICE`
Action: *prevent*

When the `LevelUp` component executes the dynamically loaded code shown in Listing 3, it tries to obtain a handle to the *LocationManager* service (recall line 4 of Listing 3). The Android framework dispatches the request to the *Context*, which then sends the request to the *Resource Monitor*. Upon receiving the resource access request, *Resource Monitor* checks it against the ECA rules and performs the corresponding action (prevents the request in this particular case).

V. IMPLEMENTATION

DELDROID is a Java application that takes as input an Android system consisting of a set of APK files. As described earlier, the architecture extraction capability was built on top of several prior static program analysis tools [28], [10], [11]. Each tool provides specific information that DELDROID uses to tailor the LP architecture. After that, DELDROID conducts a security analysis on the established LP architecture and records the security vulnerabilities that are found. The derived LP architecture and results of analysis are stored in a comma separated values (CSV) file. The implementation of DELDROID consists of more than 4,000 lines of code (LOC), not counting the existing tools on which it relies.

The enforcement mechanism in DELDROID is implemented on top of the Android Open-Source Project (AOSP) [2] version 6 (Marshmallow), API level 23. AOSP is the open-source repository for Android system maintained by Google. The *Privilege Manager Layer* introduced a new package in the Android runtime environment. We also modified other components such as *ActivityManager* and *ContextWrapper*. The total framework changes account for approximately 400 LOC. The changes were made such that any existing Android app could continue to run in our version of Android runtime environment without modification. We have successfully installed the modified Android system image on a Nexus 5X phone and on the Android emulator using Android Fastboot tools [6] and Android debug bridge [1].

VI. EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of DELDROID. Our evaluation addresses the following re-

Table II
SUMMARY OF THE APP BUNDLES AND THE ATTACK SURFACE OF BOTH ORIGINAL AND LP ARCHITECTURE.

Bundle	Components	Intent		Intent	Communication Domain			Permission Granted Domain			Priv. Esca. Security Analysis	
		Explicit	Implicit	Filter	Original	LP	Reduction (%)	Original	LP	Reduction (%)	Original	LP
Bundle 1	306	344	79	176	29,031	42	99.86	1,642	45	97.26	25,944	0
Bundle 2	432	468	379	287	78,237	625	99.20	2,954	61	97.94	35,601	110
Bundle 3	422	574	212	200	65,709	173	99.74	2,510	54	97.85	22,721	2
Bundle 4	449	348	370	511	80,372	205	99.74	4,234	78	98.16	33,551	0
Bundle 5	353	304	277	292	56,868	345	99.39	1,536	51	96.68	26,914	2
Bundle 6	541	890	476	4919	85,556	661	99.23	4,461	181	95.94	24,745	2
Bundle 7	562	412	38	324	82,863	137	99.83	1,577	58	96.32	15,503	1
Bundle 8	362	417	267	242	50,208	250	99.50	1,946	24	98.77	27,663	14
Bundle 9	265	180	98	166	25,817	129	99.50	1,568	30	98.09	19,428	8
Bundle 10	421	322	1231	185	50,001	74	99.85	2,386	28	98.83	16,953	3
Average	411.3	425.9	342.7	730.2	60,466.2	264.1	99.58	2,481.4	61.0	97.58	24,902.3	14.2
Avg. (per app)	13.7	14.2	11.4	24.3	2,015.5	8.8	99.56	82.7	2.0	97.54	498.0	0.3

search questions:

- **RQ1.** How effective is DELDROID in reducing the attack surface of Android systems and aiding the architect with understanding their security posture?
- **RQ2.** How effective is DELDROID in detecting and preventing security attacks in real-world apps?
- **RQ3.** What is the performance of DELDROID?

We downloaded a total of 984 apps in our experiments coming from three different datasets representing benign, vulnerable, and malicious apps. The benign dataset is a collection of 370 apps, randomly selected from the Google Play store. The second dataset is a collection of 389 vulnerable apps identified in prior literature [23]. Finally, the malware dataset contains 225 apps obtained from various malware repositories [40], [4], [26].

A. Attack Surface Reduction

By reducing the privileges granted to software components, DELDROID helps the security architects (or automated analysis tools) to focus their analysis effort on a narrowed set of interactions. To evaluate the degree to which DELDROID reduces the attack surface of Android systems, we ran DELDROID on 10 bundles of apps, each containing 30 apps. We chose this number of apps, since it represents the average number of apps a smartphone user regularly uses per month, as shown in a recent study [8]. Each bundle contains apps randomly selected from the app datasets as follows: 24 benign apps, 3 vulnerable apps, and 3 malicious apps.

Table II shows the structure of the bundles, including the number of entries in the Communication Domain as well as the Permission Granted Domain for both the Original and LP architectures. For example, in bundle 1, the LP architecture contains 42 inter-app communication (IAC) and 45 resource access permissions, whereas the Original architecture contains 29,031 IAC and 1,642 resource access privileges. On average, across all bundles, 99.56% of IAC and 97.54% of resource access privileges are reduced.

Table II also shows the number of potential inter-app privilege escalation attacks in both the Original and LP architectures. For example, in bundle 5, the Original architecture contains 26,914 possible privilege escalation attacks, whereas the LP architecture contains only 2 such attacks that need investigation. On average, an analyst needs to verify 14 potential security issues for a bundle of 30 apps using our approach. In fact, in the case of bundles 1 and 4, all potential privilege escalation attacks are already resolved with the LP architecture, eliminating the need for further investigation.

The results confirm the effectiveness of our approach in reducing the attack surface and hence reducing the effort required to assess the security properties of an Android system.

B. Attack Detection and Prevention

To evaluate DELDROID's ability to detect and prevent security attacks, we used 54 malicious and vulnerable apps for which the steps and inputs required to create the attacks were known. In total, the resulting combination of apps had 18 privilege escalation and 24 dynamically loaded ICC attacks. We created a bundle of these 54 apps, ran DELDROID to obtain and analyze the LP architecture, and deployed the apps on our version of Android runtime environment. We then exercised the apps to create the attacks and determined whether DELDROID was able to prevent them. We report on the *precision* and *recall* of both detection and prevention.

DELDROID marked 19 inter-app communications as potential privilege escalation attacks, correctly detecting 18 attacks, i.e., true positive. Our manual inspection of the behavior that was wrongly classified as an attack showed that this was due to the shortcomings of the underlying static program analysis tools used in DELDROID. In particular, since the analysis tools relied upon in our work are not path-sensitive, DELDROID is bound to over-approximate the behavior of Android architectures, sometimes leading to such false positive

Table III
DELDROID'S OFFLINE PERFORMANCE.

	Recovery (min)	LP Determination (sec)	Analysis (sec)	ECA Rules (sec)
Average	69.5	1.61	0.002	0.45
Std Dev	2.7	0.69	0.001	0.99

outcomes. Overall, DELDROID achieves 94.7% precision and 100% recall in detection of privilege escalation attacks. Given DELDROID's reliance on static program analyses, it is unable to detect security attacks launched via dynamically loaded code. In spite of that, as shown next, our experiments show that such attacks are effectively thwarted by an LP architecture.

To evaluate DELDROID's ability to thwart security attacks, we configured DELDROID to prevent all 19 detected privilege escalation attacks during the analysis step. We then manually exercised all known privilege escalation (19 cases) and dynamically loaded ICC (24 cases) attacks. DELDROID was able to prevent all of the attacks from succeeding by intercepting either the ICC or resource access calls. However, one of the prevented ICCs was a legitimate communication that corresponded to the erroneously detected privilege escalation attack. Overall, DELDROID achieves 97.7% precision and 100% recall in prevention of security attacks.

C. Performance

We measured the execution time of running DELDROID on the 10 bundles of app shown in Table II. These experiments were conducted on a MacBook Pro with 2.2 GHz Intel Core i7 processor and 16 GB DDR3 RAM. We repeated our experiments 33 times to achieve a 95% confidence interval. Table III summarizes the results. On average, for an Android system with 30 apps, it takes less than 70 minutes to execute DELDROID and obtain the ECA rules, but the great majority of this time is spent in the one-time effort of recovering the architecture of system from its implementation artifacts. A less precise but more efficient forms of program analysis could be substituted for architecture recovery, at the expense of a higher rate of false positives.

To evaluate the runtime overhead of DELDROID, we measured the time it takes to check the ECA rules for an intercepted ICC. To that end, we created a script that sends 363 requests (e.g., start an app, click a button) to an Android system, simulating its use. Each request causes the system to perform an ICC of some sort. We found that, on average, the performance overhead is 25 milliseconds with 10 milliseconds standard deviation. Most users cannot perceive delays of this magnitude, per Android development guidelines [7], and thus, we believe DELDROID poses an acceptable overhead.

VII. RELATED WORK

A large body of research [15], [9], [25], [20], [21], [13], [27] has focused on Android security. Here, we provide a discussion of the related efforts in light of our research.

Numerous techniques have been developed for ICC analysis [22], [23], [39], [11]. DidFail [22] introduces an approach for tracking data flows between Android components. IccTA, similarly, leverages an intent resolution analysis to identify inter-component privacy leaks [23]. Along the same line, COVERT [11] presents an approach for compositional analysis of Android inter-app vulnerabilities. While these research efforts are concerned with the analysis of information/permission leakage between Android apps, they do not really address the problem that we are addressing, namely the automated detection and dynamic enforcement of least-privilege architecture in Android. DELDROID, to our knowledge, is the first tool with this capability.

Others have focused on enforcing policies at runtime [12], [33], [38]. SEPAR [12] provides an automatic scheme for formal synthesis and enforcement of Android inter-component security policies. Kynoid [33] performs a dynamic taint analysis over a modified version of Dalvik VM. DeepDroid [38] presents an enforcement extensions based on dynamic memory instrumentation of system processes. These research efforts share with ours the emphasis on dynamic enforcement of security policies. Our work differs fundamentally in its emphasis on both providing an architectural solution and allowing a security architect to adjust the privileges at the architectural level.

Schmerl et al. [32] describe an architectural style for Android in ACME that, among other capabilities, supports analysis of certain security properties. Unlike DELDROID, their work does not provide a mechanism for determining the LP architecture, nor does it provide any runtime enforcement mechanism.

Finally, the importance of enforcing the principle of least privilege was introduced in the seminal work of Saltzer et al. [30], and is well recognized by many researchers. Notably, Scandariato et al. [31] lays the formal definition of the least privilege violation and provides a technique to identify such violation in UML models. To the best of our knowledge, DELDROID is the first solution capable of automatically recovering the architecture of an Android system to derive and enforce an LP variant of it.

VIII. CONCLUSION

This paper presents DELDROID, an automated approach for determining the least-privilege architecture for an Android system and its enforcement at runtime. The least-privilege architecture narrows the attack surface

of an Android system, making it easier to evaluate its security posture, and thwarts certain class of security attacks. Our experiments on hundreds of real-world apps show between 97% to 99% reduction of attack surface and the ability to thwart security attacks exploiting the over-privileged nature of Android with a recall of 100% and a precision of 97%.

Rather than preventing all dynamically loaded ICCs, an avenue of future work is leveraging techniques that can check the integrity of loaded code [29]. Moreover, static analysis tools including the ones that DELDROID leverages [28], [11], [9] cannot analyze obfuscated apps. Our future work involves integration of dynamic analysis techniques to mitigate limitations of a purely static approach for recovering the system's architecture.

Our research artifacts, including tools and evaluation data, are available publicly [5].

IX. ACKNOWLEDGMENT

This work was supported in part by awards CCF-1618132 and CCF-1252644 from the National Science Foundation, W911NF-09-1-0273 from the Army Research Office, HSHQDC-14-C-B0040 from the Department of Homeland Security, and FA95501610030 from the Air Force Office of Scientific Research.

REFERENCES

[1] Android debug bridge. https://developer.android.com/studio/command-line/adb.html.
[2] Android open source project. https://source.android.com/.
[3] Apktool. https://ibotpeaches.github.io/Apktool/.
[4] Contagio malware repository. http://contagiodump.blogspot.it.
[5] Deldroid. http://www.ics.uci.edu/~seal/projects/deldroid.
[6] Fastboot. https://source.android.com/source/running.html.
[7] Keeping your app responsive. https://developer.android.com/training/articles/perf-anr.html,.
[8] So many apps, so much more time for entertainment. http://www.nielsen.com/us/en/insights/news/2015/so-many-apps-so-much-more-time-for-entertainment.html.
[9] S. Arzt et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Edinburgh, United Kingdom, June 2014.
[10] K. W. Y. Au et al. Pscout: analyzing the android permission specification. In *ACM CCS*, Raleigh, NC, October 2012.
[11] H. Bagheri et al. Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, 41(9):866–886, September 2015.
[12] H. Bagheri et al. Practical, formal synthesis and automatic enforcement of security policies for android. In *Int'l Conf. on Dependable Systems and Networks*, Toulouse, France, June 2016.
[13] H. Bagheri, E. Kang, S. Malek, and D. Jackson. Detection of design flaws in the android permission protocol through bounded verification. In *FM 2015: Formal Methods*, volume 9109 of *Lecture Notes in Computer Science*, pages 73–89. 2015.
[14] S. Bugiel et al. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *USENIX Security Symposium*, Washington DC, August 2013.
[15] E. Chin et al. Analyzing inter-application communication in android. In *International Conference on Mobile Systems, Applications, and Services*, Bethesda, Maryland, June 2011. ACM.
[16] K. Coogan et al. Automatic static unpacking of malware binaries. In *Working Conf. on Reverse Engineering*, Washington, DC, October 2009.
[17] L. Davi et al. Privilege escalation attacks on android. In *Int'l Conf. on Information Security*, Boca Raton, FL, October 2010.
[18] A. Egners et al. Messing with Android's permission model. In *Int'l Conf. on Trust, Security and Privacy in Computing and Communications*, Liverpool, United Kingdom, June 2012.
[19] Z. Fang et al. Permission based Android security: Issues and countermeasures. *Computers & Security*, 43:205–218, June 2014.
[20] A. P. Felt et al. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, San Francisco, California, August 2011.
[21] A. P. Fuchs et al. Scandroid: Automated security certification of android. *University of Maryland, Tech. Rep. CS-TR-4991*.
[22] W. Klieber et al. Android taint flow analysis for app sets. In *International Workshop on the State of the Art in Java Program Analysis*, Edinburgh, United Kingdom, June 2014. ACM.
[23] L. Li et al. Iccta: Detecting inter-component privacy leaks in android apps. In *Int'l Conf. on Software Engineering*, Florence, Italy, May 2015. IEEE.
[24] U. Lindemann and M. Maurer. Facing multi-domain complexity in product development. In *The future of product development*. Springer, Berlin, Germany, 2007.
[25] L. Lu et al. Chex: statically vetting android apps for component hijacking vulnerabilities. In *conference on Computer and communications security*, New York, NY, October 2012. ACM.
[26] F. Maggi et al. Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors. In *Workshop on Security and Privacy in Smartphones and Mobile Devices*, Berlin, Germany, November 2013.
[27] D. Octeau et al. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *USENIX Sec. Symp.*, Washington DC, Aug. 2013.
[28] D. Octeau et al. Composite constant propagation: Application to android inter-component communication analysis. In *Int'l Conf. on Software Engineering*, Florence, Italy, May 2015. IEEE.
[29] S. Poeplau et al. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, San Diego, California, February 2014.
[30] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *IEEE Computer Society Press*, 63(9):1278–1308, April 1975.
[31] R. Scandariato et al. Automated detection of least privilege violations in software architectures. In *European Conference on Software Architecture*, Copenhagen, Denmark, August 2010.
[32] B. Schmerl et al. Architecture modeling and analysis of security in android systems. In *European Conference on Software Architecture*, Copenhagen, Denmark, November 2016.
[33] D. Schreckling et al. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. *Information Security TR.*, 17(3):71–80, February 2013.
[34] W. Shin et al. A Small But Non-negligible Flaw in the Android Permission Scheme. In *Int'l Symp. on Policies for Distributed Systems and Networks*, Fairfax, VA, July 2010.
[35] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS*, San Diego, California, February 2013. The Internet Society.
[36] D. V. Steward. The design structure system: A method for managing the design of complex systems. *IEEE transactions on Engineering Management*, (3):71–74, 1981.
[37] R. N. Taylor et al. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.
[38] X. Wang et al. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *NDSS*, San Diego, California, February 2015.
[39] F. Wei et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *ACM CCS*, Scottsdale, Arizona, November 2014.
[40] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109, San Francisco, California, May 2012. IEEE.