DelDroid: An Automated Approach for Determination and Enforcement of Least-Privilege Architecture in Android

Mahmoud Hammad[a,b,1], Hamid Bagheri[c], Sam Malek[b]

[a]*Department of Software Engineering, Jordan University of Science and Technology, Jordan*
[b]*Department of Informatics, University of California, Irvine, United States*
[c]*Department of Computer Science and Engineering, University of Nebraska-Lincoln, United States*

Abstract

Android is widely used for the development and deployment of autonomous and smart systems, including software targeted for IoT and mobile devices. Security of such systems is an increasingly important concern. Android relies on a permission model to secure the system's resources and apps. In Android, since the permissions are granted at the granularity of apps, and all components in an app inherit those permissions, an app's components are over-privileged, i.e., components are granted more privileges than they actually need. Systematic violation of *least-privilege principle* in Android is 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 DelDroid is the ability to limit the privileges granted to apps without modifying them. DelDroid utilizes static analysis techniques to extract the exact privileges each component needs. 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.

Keywords: Android security, Software Architecture, Multiple-Domain-Matrix (MDM)

1. Introduction

Android is widely used for the development and deployment of autonomous and smart software systems, including software intended for execution on a variety of mobile devices, as well as software targeted for Internet of Things (IoT) settings, such as smart homes. Security of such systems is an increasingly important concern. Permissions form the foundation of security in Android. Android relies 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 [1]. 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 [2]. 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 [1, 2, 3, 4, 5]. These kinds of attacks cannot be prevented by the platform at the moment, as they do not violate the security mechanisms supplied by Android.

Prior research efforts have proposed various solutions to help address certain instances of component-level attacks. Some of the proposed solutions have focused on isolating specific type of component-level threats, caused by for example advertisement [6, 7] or JNI libraries [8]; such approaches are narrowly targeted, and

[1]Corresponding author.

E-mail addresses: hammadm@uci.edu (M. Hammad), bagheri@unl.edu (H. Bagheri), malek@uci.edu (S. Malek).

thus, inappropriate for applying comprehensively to other types of component-level threats. Others have proposed component-level permission assignment for third-party components in an app [9, 10], yet they are incapable of controlling communications among components. They also often require application modification or developer intervention, significantly hindering their adoption in practice.

To systematically thwart these threats, we have developed DelDroid[2], a fully 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 [11]. 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) [12] 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.

By providing an efficient least-privilege determination process associated with a thorough enforcement system, DelDroid allows users to focus their analysis efforts on a very narrowed set of interactions in the architecture. This is especially valuable, since at the scale of a single device, the state-of-the-art inter-component communication analysis tools produce an enormous number of potential links between message-passing locations and possible message targets, making manual analysis required to confirm any potential threat rather tedious and error-prone.

DelDroid can be used to limit the levels of access 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.

This paper describes several new non-trivial extensions to the preliminary version of our work described in [13]: (1) We incorporate new security analysis rules in DelDroid to detect a broader range of inter-component communication (ICC) attacks. In addition to the *privilege escalation* analysis, DelDroid is now capable of analyzing the recovered architecture for potential *Intent spoofing* and *unauthorized Intent receipt* attacks [1]. (2) We enrich our representation of architecture in MDM to show the type of communication between various components of an Android system. DelDroid uses the additional information to analyze the system architecture for new security vulnerabilities. (3) We improve our algorithm for generating Event-Condition-Action (ECA) rules that collectively capture the determined least-privilege architecture, in turn reducing the size of rules that need to be stored in an Android device and monitored at runtime. (4) We report on new experiments to assess, among other things, the newly added security analysis capabilities. On top of these technical contributions, the paper provides an in-depth description of the determination and enforcement of least-privilege architecture in Android and a revamped discussion of this work in the context of related research.

To summarize, this paper makes the following contributions:

- *Automated derivation of LP architecture:* We develop a novel mechanism, called DelDroid, to automatically identify the LP architecture for an Android system. The run-time architecture captured in an MDM further helps users and security experts better understand and maintain the security posture of the entire system.

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

- *Dynamic enforcement:* We show how to exploit the LP architecture to safeguard the system against security attacks by enforcing it at runtime without modifying the current apps.
- *Experiments*: We present results from experiments run on hundreds of real-world apps, corroborating DelDroid's ability in (1) effectively reducing the attack surface of Android systems through the establishment of an LP architecture, and (2) efficiently detecting and preventing various security attacks through analyzing the established LP architecture and its dynamic enforcement.

The remainder of this paper is structured as follows. Section 2 provides an overview of the Android framework and its access control model to help the reader understand the discussion that follows. Section 3 motivates the research through an illustrative example. Section 4 describes DelDroid, while Section 5 describes its implementation. The evaluation results are presented in Section 6. Finally, the paper concludes with an overview of the related literature and discussions on limitations and directions for future work.

2. Android Background and Research Motivation

This section provides a brief overview of the Android framework, and the over-privileged nature of its access control model, to help the reader follow the discussions that ensue.

Android framework. Android is the most popular mobile platform accounting for 85% market share as of the first quarter of 2017 [14], and more than 3.0 million Android apps are available only on Google Play, the official Google Android app store, as of June 2017 [15]. The Android framework includes a full Linux OS based on the ARM processor, system libraries, middleware, and a suite of pre-installed applications. Android applications (apps) are mainly written in the Java programming language by using a rich collection of APIs provided by the Android Software Development Kit (SDK). An app's compiled code alongside data and resources are packed into an archive file, known as an Android package kit (APK). Once an APK is installed on an Android device, it runs by using the Android runtime (ART) environment.

Application configuration. Each Android APK includes a mandatory configuration file, called *manifest*. It specifies, among other things, the principal components that constitute the app, including their types and capabilities, as well as required and enforce permissions. The manifest file values are bound to the app at compile time, and cannot be changed afterwards, unless the app is recompiled.

Application components. Components are basic logical building blocks of apps. Each component can be invoked individually, either by its embodying app or by the system, upon permitted requests from other apps. Android defines four types of components: (1) *Activity* components provide the basis of the Android user interface. Each app may have multiple Activities representing different screens of the app to the user. (2) *Service* components provide background processing capabilities, and do not provide any user interface. Playing a music and downloading a file while a user interacts with another app are examples of operations that may run as a Service. (3) *Broadcast Receiver* components respond asynchronously to system-wide message broadcasts. A receiver component typically acts as a gateway to other components, and passes on messages to Activities or Services to handle them. (4) *Content Provider* components provide database capabilities to other components. Such databases can be used for both intra-app data persistence as well as sharing data across apps. Each component can declare a set of provided interfaces which can be invoked by other components.

Inter-component communication. Inter-component communication (ICC) in Android is mainly conducted by means of *Intent* messages. An Intent message is an event for an action to be performed along with the data that supports that action. Component capabilities are then specified as a set of *Intent Filters* that represent the kinds of requests handled by a given component. Intent Filters are the provided interfaces of a component. Component invocations come in different flavors, e.g., explicit or implicit, intra- or inter-apps, etc. An explicit Intent is delivered to the target component specified in the Intent, whereas 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 allows for late run-time binding between components in the same or different apps, where the calls are not explicit in the code, rather made possible through event messaging, a key property of event-driven systems.

Android's access control model. Permissions are the cornerstone of the Android access control model. 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. Android manages both types of privilege at the app level, meaning that the permissions are granted/revoked at the level of an app and inherited by all components in that app. This causes two kinds of over-privileges, discussed next.

2.1. 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`).

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 [16, 17], 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 [18, 19, 20], and shown to be the root cause of various security attacks, most notably privilege escalation [21, 22].

2.2. Over-Privileged Inter-Component Communication

The ICC mechanism in the Android framework provides a flexible component-based development. However, this mechanism gives the components more communication privileges than they actually need and hence violates the principle of least privilege. Specifically, 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 [1]. 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, most notably hidden communications [2], Intent Spoofing and Unauthorized Intent Receipt ICC attacks [1]. 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.

3. 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 hidden code [1, 2], and privilege escalation [3, 22].

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, and it allows a user to share messages with paired devices using Bluetooth. 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, and performing Bluetooth tasks requires Bluetooth permission. The `Messaging` app has these permissions, and hence all its components acquire them 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

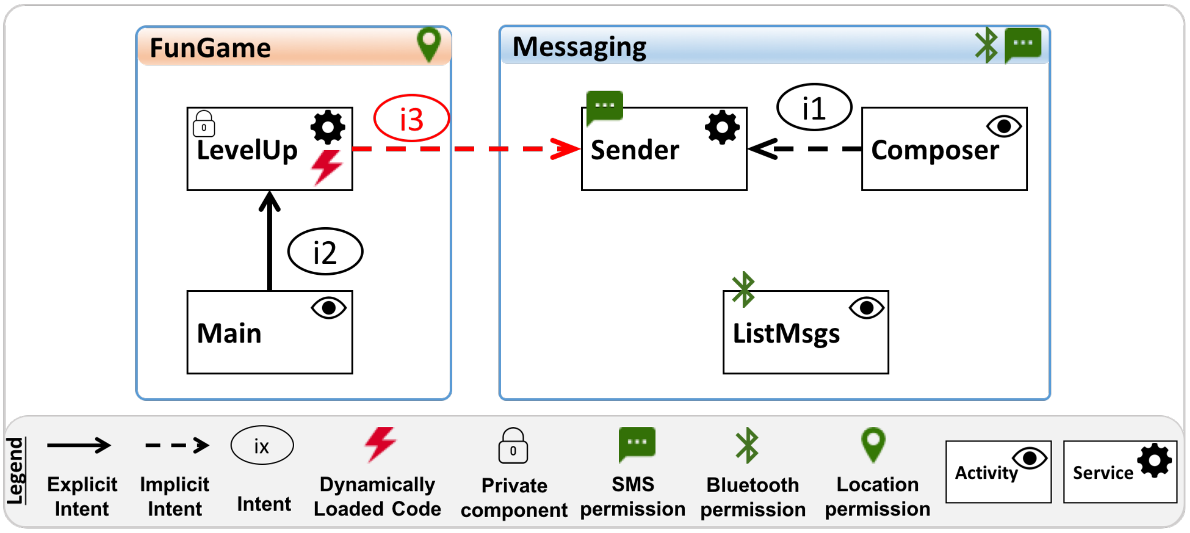


Figure 1: Component-based architecture of a vulnerable Android system.

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      ...
```

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 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. This type of vulnerability is quite common, as many developers fail to properly use the APIs or follow the best practices for secure programming. In fact, in the Android domain, since many apps are developed by novice programmers, misuse of APIs is rampant.

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

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

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

Listing 3: Code downloaded after initial installation of app.

```java
public class HiddenBehavior {
    ...
    public Intent getIntent(Context context){
        LocationManager locMgr = (LocationManager) context.getSystemService(Context.LOCATION_SERVICE);
        Location loc = locMgr.getLastKnownLocation(LocationManager.GPS_PROVIDER);
        String msg = loc.getLatitude()+","+loc.getLongitude();
        Intent i = new Intent("SEND_SMS");
        i.putExtra("PHONE_NUMBER", phoneNumber);
        i.putExtra("MSG_CONTENT", msg);
        return i;
    }
}
```

dynamic program analysis, as malicious apps often incorporate complicated evasion tactics (e.g., timing-bombs [23]). We show how through establishment of an LP architecture, DelDroid can effectively mitigate such threats.

4. Approach

As depicted in Figure 2, DelDroid consists of five steps (1) *Architectural Elements Extractor* uses static program analysis techniques to elicit the system's principal components along with their properties, latent communications, and permissions usages from the apps comprising a system. (2) *Privilege Analyzer* systematically examines each component to comprehensively determine its privileges, the permissions it can use as well as components with which it can communicate, both inside and outside the scope of its hosting app, as permitted by the Android runtime environment. The result of this step is captured in a *Multiple-Domain Matrix (MDM)*, representing the original architecture of system. (3) *Privilege Reducer* determines the exact permissions and communications each component needs to fulfill its functionality. The derived information is then captured in an *MDM*, representing the least privilege architecture for the system. (4) *Security Analyzer* evaluates the identified LP architecture apropos potential security threats, and presents the analysis results to the security architect who may further modify the architecture as needed to establish the proper privileges for each component. (5) Finally, *LP Enforcer* regulates interactions at the granularity of components through enforcing automatically generated or expert-supplied least-privilege architecture at runtime. It relies on two components, i.e., *Resource Monitor* and *ICC Monitor*, within the *Privilege Manager* layer that we have added to the Android runtime environment to check the conformance of ICC and resource-access transactions to the LP architecture, captured as Event-Condition-Action (ECA) rules. The rest of this section presents each step in detail.

4.1. 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.

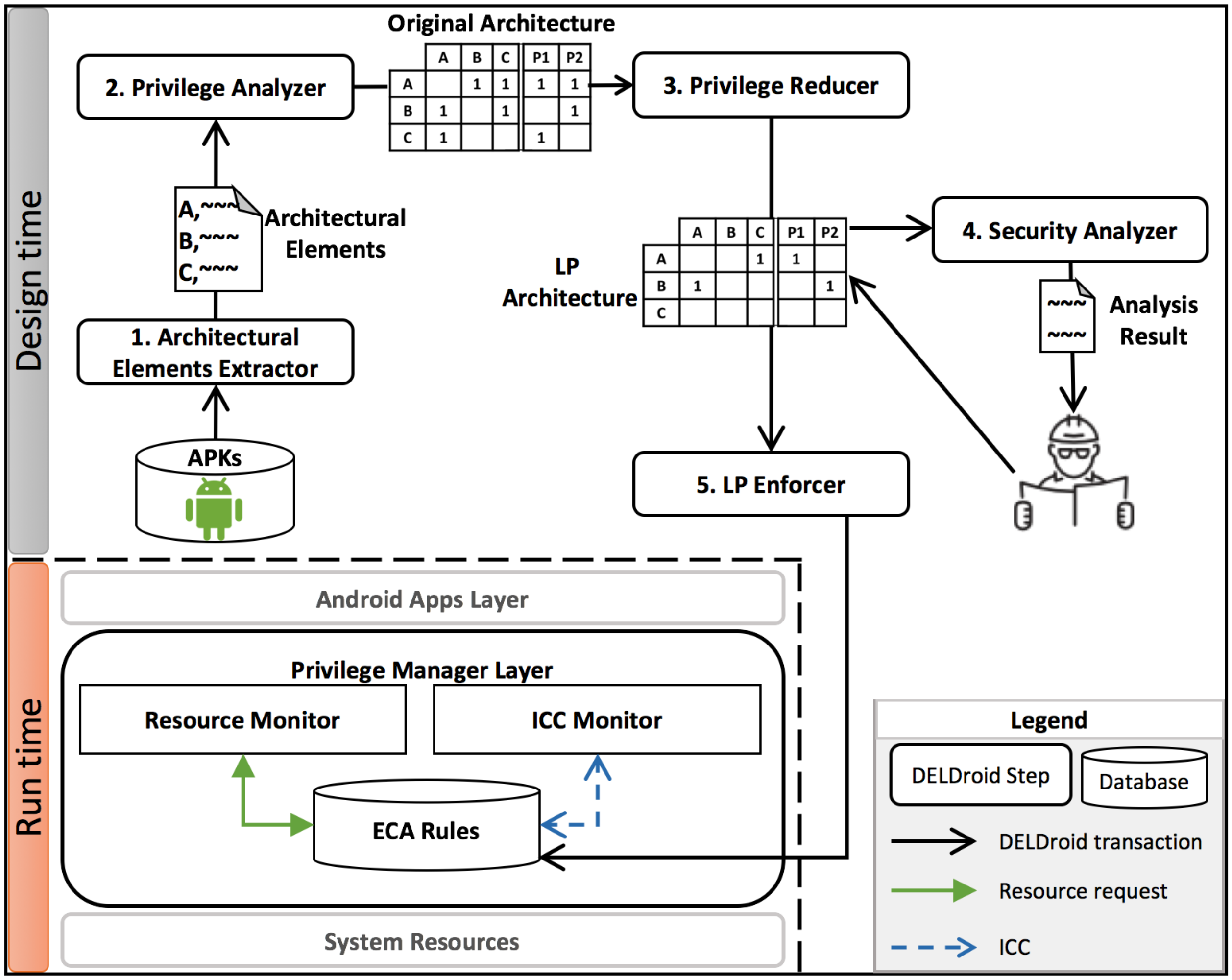


Figure 2: Overview of DelDroid.

DelDroid utilizes *APKtool* [24], 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, such as their names, types, visibility, permissions required by other components for interaction. Table 1 partially shows the extracted information corresponding to our running example (recall Section 3). The *Component 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. Among others, the three components of the `Messaging` app all have access to both the SMS (`android.permission.SEND_SMS`) permission and the Bluetooth (`android.permission.BLUETOOTH`) permission, given that the `Messaging` app acquires the SMS and the BLUETOOTH permissions.

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 [25] to analyze each app in the system and extract such latent information from its bytecode.

Table 1: The extracted architectural elements for the Android system shown in Figure 1

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

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 1 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). Moreover, the type of each extracted Intent, i.e., explicit or implicit, is indicated in the *Intent Type* column.

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. Using this mapping and the accessed Content Providers, our approach determines the actually used permissions for a component. Since IC3 does not extract the permissions used through API calls, for the latter case, DelDroid leverages PScout permission map [26], one of the most recently updated and comprehensive permission maps available for the Android framework. It specifies mappings 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 1.

Finally, DelDroid builds on our prior work [3] 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 1 is empty.

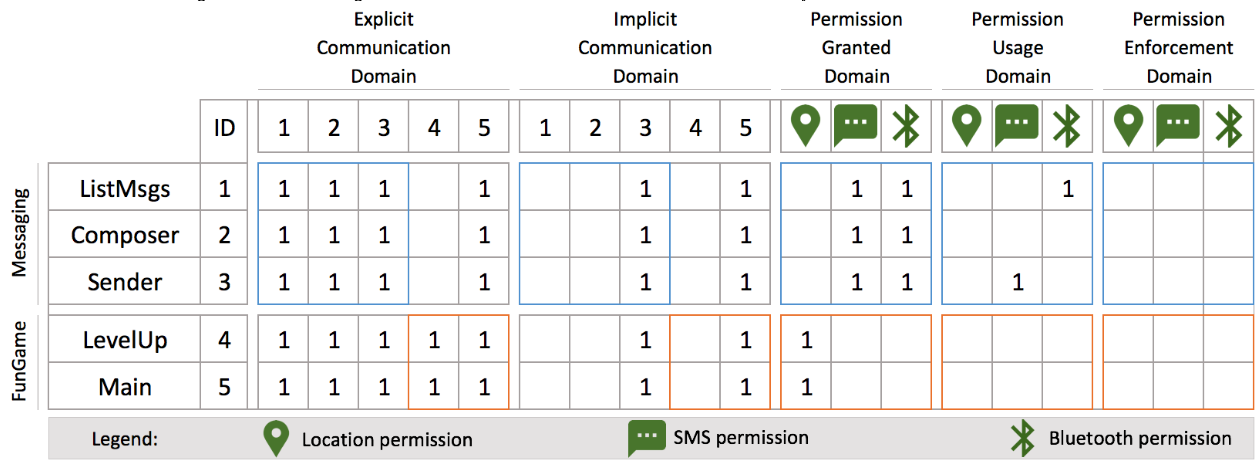
4.2. Step 2: Privilege Analyzer

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) [12]. MDM provides an elegant representation of complex systems with multiple concerns (domains). Each concern is modeled as a Design-Structure Matrix (DSM) [27]—a simple matrix that captures the dependencies of one relationship type. MDM is formed by connecting the DSMs together. We capture five domains in an MDM to represent an Android system's architecture for the purpose of privilege analysis.

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

Definition 1 (Allowed Explicit Communication)**.** *Let E be a set of all exported components, and* c_1

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



			Explicit Communication Domain					Implicit Communication Domain					Permission Granted Domain			Permission Usage Domain			Permission Enforcement Domain		
		ID	1	2	3	4	5	1	2	3	4	5	📍	💬	ᛒ	📍	💬	ᛒ	📍	💬	ᛒ
Messaging	ListMsgs	1	1	1	1		1			1		1		1	1			1			
	Composer	2	1	1	1		1			1		1		1	1						
	Sender	3	1	1	1		1			1		1		1	1		1				
FunGame	LevelUp	4	1	1	1	1	1			1		1	1								
	Main	5	1	1	1	1	1			1		1	1								

Legend: 📍 Location permission 💬 SMS permission ᛒ Bluetooth permission

and c_2 be two arbitrary components in the system. We say that c_1 can explicitly 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:

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

The Explicit Communication Domain in Figure 3 shows the result of applying Definition 1 to Table 1. According to the explicit 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. Components 4 and 5 can also communicate with all the other components in the system.

Allowed implicit communications are derived using the following rule.

Definition 2 (Allowed Implicit Communication). *Let F be a set of all declared public provided interfaces, i.e., Intent filters, and c_1 and c_2 be two arbitrary components in the system. We say that c_1 can implicitly communicate with c_2, if c_2 defines a public provided Interface and either both components belong to the same app or c_1 is granted the permissions enforced by c_2:*

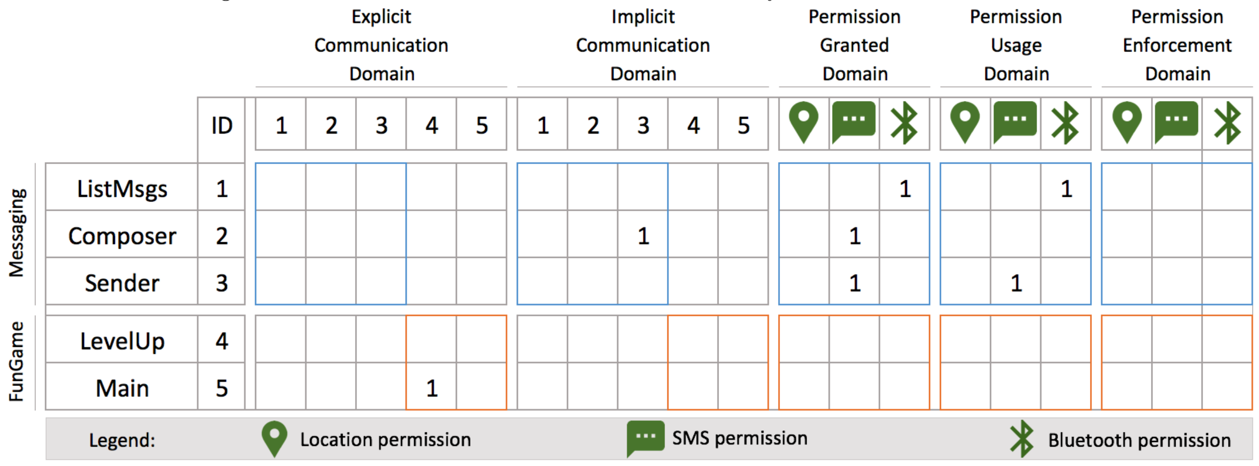
$$communicate_i(c_1, c_2) \equiv c_2.filters \subseteq F \wedge (app_{c1} = app_{c2} \vee enforced_{c2} \subseteq granted_{c1})$$

The Implicit Communication Domain in Figure 3 shows the result of applying Definition 2 to Table 1. According to the implicit communication domain, all components in the system can communicate with component 3 and component 5. Component 3 declares a public provided interface for sending text messages without enforcing any permission. Component 5 is the main entry point for *FunGame* app, i.e., declares a public Intent filter with *android.intent.action.MAIN* action.

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

Figure 4: LP architecture determined from the Android system described in Section 3.



			Explicit Communication Domain					Implicit Communication Domain					Permission Granted Domain			Permission Usage Domain			Permission Enforcement Domain		
		ID	1	2	3	4	5	1	2	3	4	5	📍	💬	ᛒ	📍	💬	ᛒ	📍	💬	ᛒ
Messaging	ListMsgs	1													1			1			
	Composer	2								1				1							
	Sender	3												1			1				
FunGame	LevelUp	4																			
	Main	5				1															

Legend: 📍 Location permission 💬 SMS permission ᛒ Bluetooth permission

manifest file, (2) used by a component, meaning that the component is actually making API calls or interacts with other apps that require 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 4.1 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 1). For example, the MDM shown in Figure 3 indicates that the first three components are granted the SMS and the BLUETOOTH permissions, while components 4 and 5 are granted the location permission.

4.3. 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 *Intent* and *Intent Type* columns of Table 1) 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 of Implicit Communication Domain). 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 3 (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:*

$$requiredPermissions_{c1} = \{p : Permission \mid \exists\ c_2 : Component \bullet p \in used_{c1} \vee ((communicate_e(c_1, c_2) \vee communicate_i(c_1, c_2)) \wedge p \in used_{c2} \wedge p \in granted_{c1})\}$$

According to Definition 3, 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. The `ListMsgs` component, on the other hand, has a legitimate reason to hold the *BLUETOOTH* permission since it uses that permission, while `Sender` and `Composer` do not need it. The `ListMsgs` component, 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.

The amount of privilege reduction achieved through enforcing LP architecture can be quantified by calculating the distance between the LP architecture (L) and the Original architecture (O) as shown in Equation 1.

$$Reduction(O, L) = 1 - \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} L_{ij}}{\sum_{i=1}^{n} \sum_{j=1}^{m} O_{ij}} \tag{1}$$

In Equation 1, i and j represent the *ith* column and *jth* row of an MDM with n rows (components) and m columns (components and permissions). In our running example, comparing the Original architecture (cf. Figure 3) with the LP architecture (cf. Figure 4) shows 83.3% reduction in granted privileges.

4.4. 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 three prominent types of vulnerabilities due to the interaction of multiple apps, i.e., privilege escalation [22], unauthorized Intent receipt [1], and Intent spoofing [1, 4].

Definition 4 (Privilege Escalation)**.** *Let p be a permission, c_m be a malicious component that does not hold p, and c_v be a vulnerable 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.*

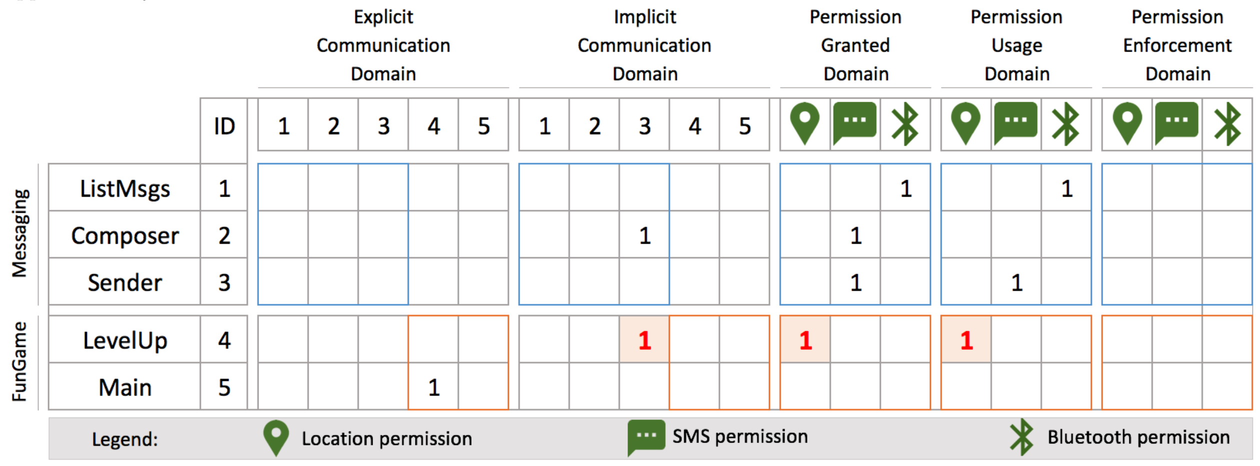
$$(communicate_e(c_m, c_v) \lor communicate_i(c_m, c_v)) \land p \in used_{c_v} \land p \notin granted_{c_m} \land p \notin enforced_{c_v}$$

According to Definition 4, 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 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.

Definition 5 (Unauthorized Intent Receipt)**.** *Let c_m, c_v, and c_x be three components, where c_v and c_x*

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



			Explicit Communication Domain					Implicit Communication Domain					Permission Granted Domain			Permission Usage Domain			Permission Enforcement Domain		
		ID	1	2	3	4	5	1	2	3	4	5	📍	💬	ᛒ	📍	💬	ᛒ	📍	💬	ᛒ
Messaging	ListMsgs	1													1			1			
	Composer	2								1				1							
	Sender	3												1			1				
FunGame	LevelUp	4								**1**			**1**			**1**					
	Main	5				1															

Legend: 📍 Location permission 💬 SMS permission ᛒ Bluetooth permission

belong to the same app, and c_x declares a public provided interface, i.e., an Intent filter, through which c_v aims to communicate with c_x by means of an implicit Intent. In the unauthorized Intent receipt attack, c_m can intercept an implicit Intent sent by c_v through declaring a provided interface similar to the one declared by c_x. As such, c_m may gain access to all enclosed data in any matching Intents meant to be received by c_x.

$$communicate_i(c_v, c_m) \land (app_{c_v} \neq app_{c_m}) \land \exists\, communicate_i(c_v, c_x) \land (app_{c_v} = app_{c_x})$$

Unauthorized Intent receipt is an ICC attack in which a malicious component intercepts an implicit Intent by declaring an Intent Filter that matches the sent Intent [1, 28]. In such an attack, a malicious component can access all enclosed data in the intercepted Intent and, possibly perform a phishing attack [29].

There are three different forms of unauthorized Intent receipt based on the type of the malicious component (c_m in Definition 5) [1]: (1) *Broadcast theft* in which c_m can read the content of broadcast Intents without interrupting the broadcast, (2) *Activity hijacking* in which c_m is launched instead of a legitimate Activity, and (3) *Service hijacking* in which c_m is bound to/started instead of a legitimate one. In case a hijacking attack is successful, c_v may also be a victim of *false response attack* [1, 28] in which c_m can return a malicious result to c_v.

As a concrete example of unauthorized Intent receipt attack, consider a legitimate application that processes financial payments. When a user clicks on a "Pay" button, the application sends an implicit Intent to start another Activity that processes the payment. If a malicious Activity hijacks the implicit Intent, then the attacker could receive sensitive information from the user (e.g., card number, billing address, and payment amount). In this Activity hijacking attack, the malicious component can also perform a phishing attack to get even more information from the user after stealing the interface of the legitimate Activity. Phishing attacks cannot be easily determined by users since the Android UI does not specify the currently running application. By applying Definition 5 to the MDM representation of the system's architecture, DelDroid identifies communications that may result in unauthorized Intent receipt ICC attack.

Definition 6 (Intent Spoofing). *Let c_m, c_v, and c_x be three components, where c_v and c_x belong to* *the same app and c_v declares a public provided interface, i.e., an Intent filter, through which it aims to communicate with c_x. In the Intent spoofing attack, c_m can communicate with the exported component of c_v that is not expecting an Intent from c_m. In this attack, if the vulnerable component c_v performs an action upon receiving an Intent, the malicious component c_m can trigger that action at will for nefarious purposes.*

$$(communicate_e(c_m, c_v) \vee communicate_i(c_m, c_v)) \wedge (app_{cv} \neq app_{cm}) \wedge \exists communicate_i(c_x, c_v) \wedge (app_{cv} = app_{cx})$$

Intent spoofing is an ICC attack in which a malicious component can communicate with an exported component that is not expecting a communication from it [1, 28]. If a victim component blindly trusts the received Intent, this attack allows a malicious component to cause a victim component to perform some actions.

There are three different forms of the Intent spoofing attack based on the type of the victim component (c_v in Definition 6) [1]: (1) *Malicious Broadcast injection* in which c_m can send a malicious broadcast Intent to an exported Broadcast Receiver. Since most Broadcast Receivers act as gateways to other components, and pass messages to Activities and Services [30], the malicious Intent can propagate throughout an app. A more risky scenario can happen if the Broadcast Receiver c_v is registered to receive protected broadcast Intents that only the system can send. In such a scenario, c_m still can send an explicit Intent to c_v. If c_v blindly trusts the received Intent without checking the Intent action, c_v may perform a task that only the system is supposed to trigger.(2) *Malicious Activity launch*, analogous to cross-site request forgeries (CSRF) in websites [31], occurs when a victim component c_v is launched by a malicious component c_m that it does not expect communication from. Since Activities provide GUI interfaces, this attack can be an annoyance to the users. Successfully launching the c_v Activity can cause c_v to change data in the background using the data enclosed in the malicious Intent sent by c_m. (3) *Malicious Service launch* is similar to *malicious Activity launch* except that the interaction between c_m and c_v occurs in the background. If a *malicious Activity launch* or a *malicious Service launch* attack is successful, c_v may return sensitive information to the malicious component c_m.

As a concrete example of Intent spoofing attack, consider an application that contains an advertisement (ad) library. Once a user clicks on an ad, the application sends an implicit Intent to an Activity, referred to as AdActivity here, which displays details of that ad on a web page. In this case, a malicious component can exploit an Intent spoofing attack by sending a carefully crafted implicit Intent to the AdActivity. If the AdActivity does not properly handle the received implicit Intent, the malicious component can deny the service of AdActivity and crash its app resulting in an *inter-process denial-of-service* (IDOS) attack. Moreover, if the AdActivity blindly trusts the incoming implicit Intent, a malicious component can redirect the user to a web page with malicious JavaScript code resulting in a *cross-application scripting (XAS)* attack. We refer the interested readers to [4] for more details on these kinds of Intent spoofing attacks.

By applying Definition 6 to the MDM representation of the system's architecture, DelDroid identifies communications that may result in Intent spoofing ICC attack. Applying the Intent spoofing rule (Definition 6) to the LP architecture of Figure 5 reveals that the communication between `LevelUp` and `Sender` satisfies the Intent spoofing rule. Since both `LevelUp` and `Sender` belong to different apps and also there is a communication between `Composer` and `Sender`, two components that belong to the same app. However, since this communication is already marked as potential privilege escalation attack via applying the Privilege escalation rule (Definition 4), DelDroid will not raise another warning for this communication.

It is worth mentioning that all violations to the determined LP architecture are recorded and accessible to the security architect through an Android app that we have developed, not shown in Figure 2 to reduce the clutter in the figure. This app allows a security architect to understand the running system and adjust the architecture as needed.

4.5. Step 5: LP Enforcer

This step regulates component interactions by enforcing the LP architecture at runtime. DelDroid efficiently 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.

4.6. Efficiently Generating ECA Rules

Event-condition-action (ECA) rules allow the system to automatically perform actions in response to events given that the stated conditions hold. Each ECA rule reads as follows: "when an event occurs, check the condition, if it holds, execute the action". ECA rules make the system efficiently adapt while the rules are stored in a single rule base instead of encoding them in many modules, thus improving the maintainability and the manageability of the system. ECA rules have been widely used in the literature, including self-adaptive systems [32, 33, 34], databases [35, 36], business process modeling and analysis tools [37, 38, 39], and web technologies [40, 41].

Since the identified LP architecture will be stored and monitored in resource-constrained mobile devices in terms of a set of ECA rules, it is significantly important for such rules to be efficient in a way that would minimize the number of required ECA rules. A naïve approach for generating ECA rules that capture an LP architecture of n rows and m columns would result in $n \times m$ ECA rules, where each cell is captured by an ECA rule. However, such an approach results in the generation of a large number of rules, many of which are very similar.

DelDroid generates ECA rules more efficiently. As for ICC ECA rules, i.e., the rules that capture the explicit and implicit communication domains of an LP architecture, if a component has no legitimate reason to communicate with any component of another app, DelDroid generates only one ECA rule that entirely prevents that particular component from communicating with that app. This, in turn, reduces the number of generated ECA rules from the number of components in the target app to merely one ECA rule. Similarly, if no component of an app is allowed to communicate with any component of another app, DelDroid generates just one ECA rule that prevents all components of the former app from communicating with components of the latter app. Generating ECA rules in this way not only reduces the number of generated rules but also makes the search process for an ECA rule governing a specific component or a specific app faster. Once DelDroid finds a coarse-grained ECA rule, i.e., a rule that restricts one app from communicating with another app, DelDroid stops the search and executes the action specified in that ECA rule.

In the case of resource access ECA rules, i.e., ECA rules that capture the Permission Granted Domain, DelDroid generates resource access ECA rules only for the granted permissions, i.e., ECA rules that capture only the "1"s in the Permission Granted Domain. It is worth mentioning that, in Android, it is possible for one permission to protect more than one system resource. In such a case, DelDroid generates more than one resource access ECA rule per granted permission. For example, the `android.permission.READ_PHONE_STATE` permission is required to request `CARRIER_CONFIG_SERVICE` in order to access the carrier configuration values, and the same permission is required to request the `TELEPHONY_SERVICE` to access the `TelephonyManager`, which provides access to information about the telephony services on a device.

4.6.1. 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: $i.senderPkg = \texttt{FunGame} \wedge i.senderComp = \texttt{LevelUp} \wedge i.receiverPkg = \texttt{Messaging}$
Action: *prevent*

At runtime, when `LevelUp` tries to communicate with `Sender`, line (17) in Listing 2, the Android framework passes the request to the *ActivityManager* which sends the ICC transaction's information (sender, receiver, and the Intent's attributes) to the *ICC Monitor* component. After that, *ICC Monitor* vets the ICC transaction in light of the stored ECA rules. If a matched ECA rule is found, *ICC Monitor* prompts the *ActivityManager* to execute the associated action (*prevent* the communication in this particular example).

4.6.2. Resource Monitor

As we explained in Section 2.1, 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 grant `ListMsgs` permission to access the Bluetooth service:

Event: *resourceaccessrequest*
Condition: $requester = \texttt{ListMsgs} \wedge service = \texttt{Context.BLUETOOTH\_SERVICE}$
Action: *allow*

When the `ListMsgs` component performs Bluetooth management tasks such as initiating device discovery or listing all paired devices, it tries to obtain a handle to the *BluetoothManager* service. 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 (*allows* the request in this particular case).

As another example, 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*. Since there is no ECA rule that grants `LevelUp` access to the device's location, *Resource Monitor* prevents `LevelUp` from obtaining a handle to the Location Manager service.

5. 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 [3, 25, 26]. 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) [42] 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. Moreover, our modifications to the Android version 6 are not restricted to this version and we expect that they can be applied to the other versions of the framework without technical difficulties.

We built the modified AOSP on an Ubuntu server with a 64-core AMD processor and 264GB RAM. It took about an hour to complete the build process. We have successfully installed the modified Android system image on a Nexus 5X phone and on the Android emulator using Android Fastboot tools [43] and Android debug bridge [44].

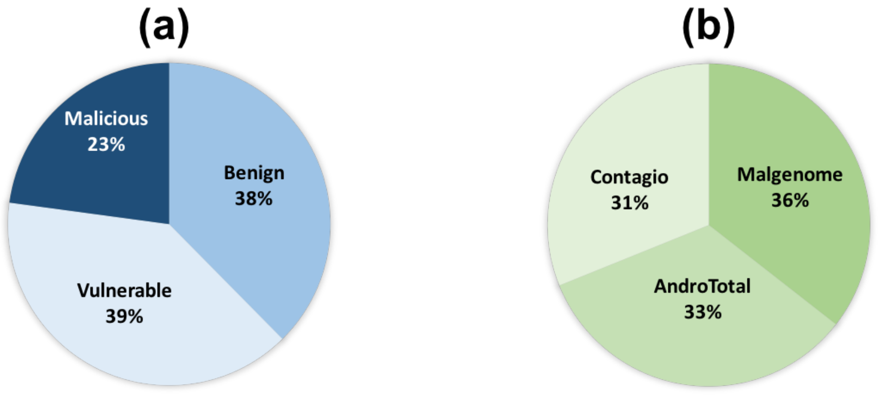


Figure 6: (a) Distribution of the entire experimental subjects across various repositories from which the subject apps are downloaded; (b) distribution of apps from various malware repositories that were used in our experiments.

6. Experimental Evaluation

This section presents the experimental evaluation of DelDroid. Our evaluation addresses the following research 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 well does DelDroid perform in practice? Can it detect and prevent security attacks in real-world apps?
- **RQ3.** How efficient is DelDroid in generating ECA rules that capture the determined LP architecture?
- **RQ4.** What is the performance of DelDroid?

We constructed datasets of benign, malicious, and vulnerable Android apps as shown in Figure 6(a). The benign dataset is a collection of 370 apps, randomly selected from the Google Play store. To prevent any bias in the results, we did not use any particular criteria, such as high ranking or high downloads, in selection of the Google Play apps. Therefore, these apps vary in terms of their 5-star ranking, as depicted in Figure 7 (a), as well as their number of downloads, as depicted in Figure 7 (b). The second dataset is a collection of 389 vulnerable apps identified in prior literature [45]. Finally, the malware dataset contains 225 apps obtained from various malware repositories [46, 47, 48]. Figure 6(b) illustrates the distribution of apps from various malware repositories that were used in our experiments.

6.1. RQ1. 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 non-overlapping 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 [49]. Each bundle contains apps randomly selected from the app datasets as follows: 24 benign apps, 3 vulnerable apps, and 3 malicious apps. Figure 8 depicts a histogram of the Google Play categories of the benign apps.

Table 2 shows the structure of the bundles, including the number of entries in the Communication Domains, i.e., the Explicit Communication Domain and the Implicit Communication Domain, as well as the Permission Granted Domain for both the Original and LP architectures. To measure the degree to which DelDroid reduces the attack surface of Android systems, we used Equation 1. For example, in bundle 1, the

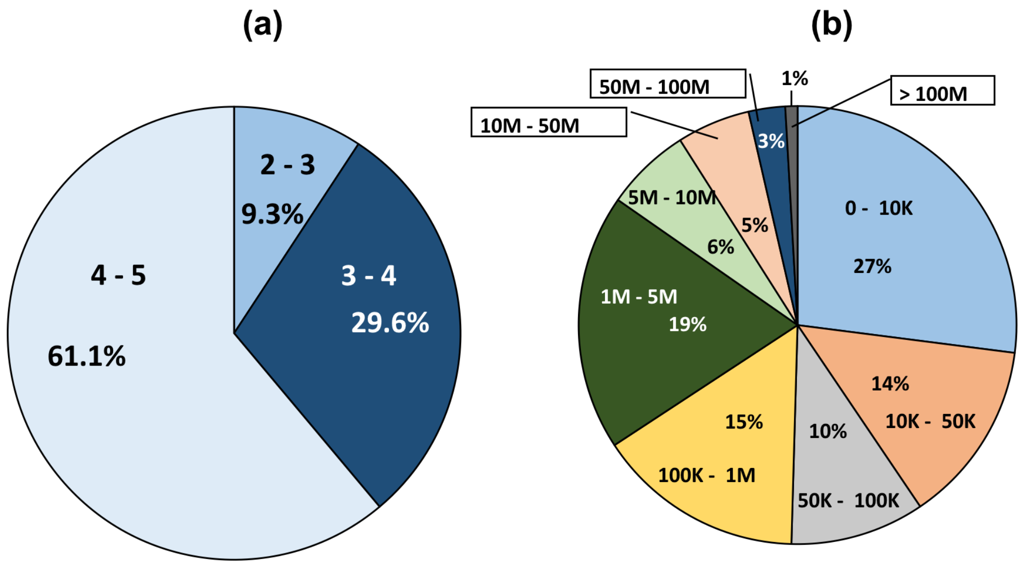


Figure 7: The popularity of the Google Play apps in terms of their (a) 5-star ranking and (b) number of downloads as of June of 2018.

Table 2: Summary of app bundles, and Original and LP architecture obtained from running DelDroid over the bundles.

Bundle	Components	Intent		Intent	Communication Domains			Permission Granted Domain		
		Explicit	Implicit	Filter	Original	LP	Reduction (%)	Original	LP	Reduction (%)
Bundle 1	306	344	79	176	29,031	42	99.86	1,642	178	89.16
Bundle 2	432	468	379	287	78,237	625	99.20	2,954	143	95.16
Bundle 3	422	574	212	200	65,709	173	99.74	2,510	109	95.66
Bundle 4	449	348	370	511	80,372	205	99.74	4,234	146	96.55
Bundle 5	353	304	277	292	56,868	345	99.39	1,536	81	94.73
Bundle 6	541	890	476	4919	85,556	661	99.23	4,461	329	92.63
Bundle 7	562	412	38	324	82,863	137	99.83	1,577	109	93.09
Bundle 8	362	417	267	242	50,208	250	99.50	1,946	92	95.27
Bundle 9	265	180	98	166	25,817	129	99.50	1,568	57	96.36
Bundle 10	421	322	1231	185	50,001	74	99.85	2,386	127	94.68
Average	411.3	425.9	342.7	730.2	60,466.2	264.1	99.58	2,481.4	137.1	94.33
Avg. (per app)	13.7	14.2	11.4	24.3	2,015.5	8.8	99.56	82.7	4.6	94.47

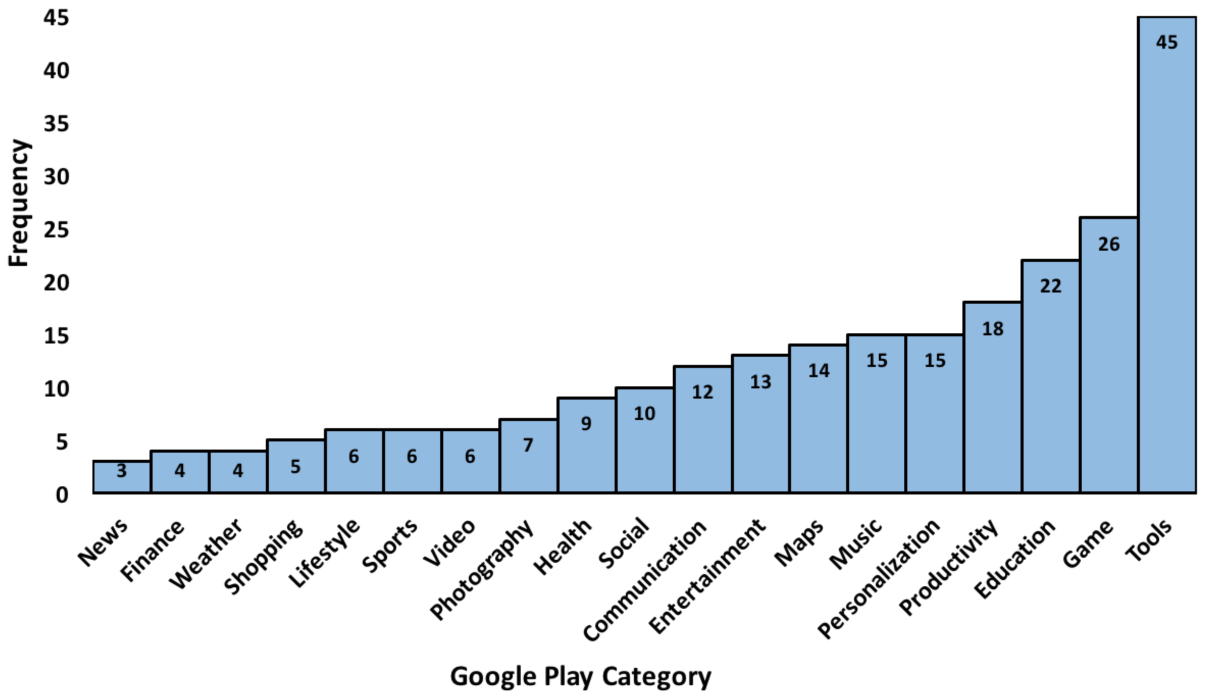


Figure 8: Histogram of Google Play categories.

LP architecture contains 42 inter-app communication (IAC) and 178 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 94.47% of resource access privileges are reduced.

Table 3 shows the number of potential ICC attacks in both the Original and LP architectures. Recall from Section 4.4 that DelDroid analyzes both the Original and LP architectures and pinpoints potential ICC attacks including privilege escalations, unauthorized Intent receipts, and Intent spoofing attacks. 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 privilege escalation 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. Similar patterns can be observed for unauthorized Intent receipt and Intent spoofing attacks. For example, in bundle 10, the Original architecture contains 2,015 potential Intent spoofing and 214 potential unauthorized Intent receipt ICC attacks, whereas the LP architecture contains only 1 potential Intent spoofing and 3 potential unauthorized Intent receipt attacks that need investigation. On average, an analyst needs to investigate 28 potential Intent spoofing and 8 potential unauthorized Intent receipt for a bundle of 30 apps using our approach. Note that an analyst needs to verify less than 2 security issues per app on average. Even in some cases, such as in bundle 1, all potential ICC attacks are already resolved with the LP architecture, entirely 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.

6.2. RQ2. Attack Detection and Prevention

To evaluate DelDroid's ability to detect and prevent security attacks, we used 54 malicious and vulnerable open-source apps for which the steps and inputs required to create the attacks were known. To validate the attacks, we manually reviewed the code and affirmed the existence of security issues. In total, the resulting combination of apps had 18 privilege escalation and 24 dynamically loaded ICC attacks. We

Table 3: Summary of ICC attack surfaces in both Original and LP architectures across app bundles.

Bundle	Privilege Escalation			Intent Spoofing			Unauthorized Intent Receipt		
	Original	LP	Reduction (%)	Original	LP	Reduction (%)	Original	LP	Reduction (%)
Bundle 1	25,944	0	100.00	2,242	0	100.00	297	0	100.00
Bundle 2	35,601	110	99.69	1,980	65	96.72	204	21	89.71
Bundle 3	22,721	2	99.99	3,132	0	100.00	299	7	97.66
Bundle 4	33,551	0	100.00	4,020	57	98.58	599	4	99.33
Bundle 5	26,914	2	99.99	12,402	24	99.81	1,646	7	99.57
Bundle 6	24,745	2	99.99	1,416	17	98.80	33	24	27.27
Bundle 7	15,503	1	99.99	1,077	1	99.91	78	0	100.00
Bundle 8	27,663	14	99.95	6,283	115	98.17	297	4	98.65
Bundle 9	19,428	8	99.96	4,638	4	99.91	371	10	97.30
Bundle 10	16,953	3	99.98	2,015	1	99.95	214	3	98.60
Average	24,902.3	14.2	99.94	3,920.5	28.4	99.28	403.8	8	98.02
Avg. (per app)	498.0	0.3	99.94	130.7	0.9	99.28	13.5	0.3	98.02

Table 4: The ability of DelDroid to detect ICC security attacks.

Actual ICC attacks	Malicious ICC detected (TP)	Malicious ICC not detected (FP)	Benign ICC detected (FP)	Precicion (%) TP / (TP + FN)	Recall (%) TP / (TP + FP)
18	18	0	1	94.74	100.00

Table 5: The ability of DelDroid to prevent ICC security attacks at runtime.

Actual ICC attacks	Malicious ICC prevented (TP)	Malicious ICC not prevented (FP)	Benign ICC prevented (FP)	Precicion (%) TP / (TP + FN)	Recall (%) TP / (TP + FP)
42	42	0	1	97.67	100.00

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. The *precision* shows the ability of DelDroid to detect/prevent system transactions that are actually malicious. On the other hand, the *recall* shows the ratio of the detected/prevented security attacks to all known attacks in the system.

As shown in Table 4, 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 outcomes. Overall, DelDroid achieves 94.74% 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. As shown in Table 5, DelDroid

Table 6: Comparing the number of generated ECA rules between DelDroid and the Naïve approach.

Bundle	Communication ECA rules			Pemission granted ECA rules		
	Naïve	DelDroid	Improvement (%)	Naïve	DelDroid	Improvement (%)
Bundle 1	93,636	1,035	98.89	1,917	211	88.99
Bundle 2	186,624	1,534	99.18	3,573	257	92.81
Bundle 3	178,084	893	99.50	3,094	115	96.28
Bundle 4	201,601	1,416	99.30	5,556	161	97.10
Bundle 5	124,609	1,238	99.01	1,840	99	94.62
Bundle 6	292,681	1,687	99.42	5,593	344	93.85
Bundle 7	315,844	1,027	99.67	2,046	151	92.62
Bundle 8	131,044	1,039	99.21	2,307	92	96.01
Bundle 9	70,225	1,051	98.50	1,964	69	96.49
Bundle 10	177,241	1,069	99.40	2,794	172	93.84
Average	177,159	1,199	99.21	3,068	167.10	94.26

Table 7: DelDroid's offline performance.

	Recovery (min)	LP Determination (sec)	Analysis (sec)	ECA Rules (sec)
Average	69.5	0.787	0.001	0.008
Std Dev	2.7	0.299	0.001	0.002

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 97.76% precision and 100% recall in prevention of security attacks.

6.3. RQ3. Efficiently Generating ECA Rules

Table 6 compares the numbers of generated ECA rules by DelDroid and the Naïve approach (recall Section 4.6). For example, in bundle 1, the Naïve approach would generate 93,636 ICC ECA rules, whereas DelDroid generates 1,035 ICC ECA rules showing more than 98% reduction in the number of rules that need to be monitored. On average, for an Android system with 30 apps, the Naïve approach would generate 177,159 ICC ECA rules, whereas DelDroid generates 1,199 ICC ECA rules to capture the communication domains in the LP architecture. Similarly, the Naïve approach would generate 1,917 resource access ECA rules for bundle 1, whereas DelDroid generates 211 resource access ECA rules for the same bundle. On average, for an Android system with 30 apps, the Naïve approach would generate 3,068 resource access ECA rules, whereas DelDroid generates 167 resource access ECA rules to capture the Permission Granted domain.

The results presented in Table 6 confirm the efficiency of DelDroid in generating ECA rules to capture an LP architecture and hence reducing the time required to validate components' communications and resource access requests at runtime.

6.4. RQ4. Performance

We measured the execution time of running DelDroid on the 10 bundles of apps shown in Table 2. 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 7 summarizes the results. On average, for an Android system with 30 apps, it takes less than 70 minutes to execute DelDroid and

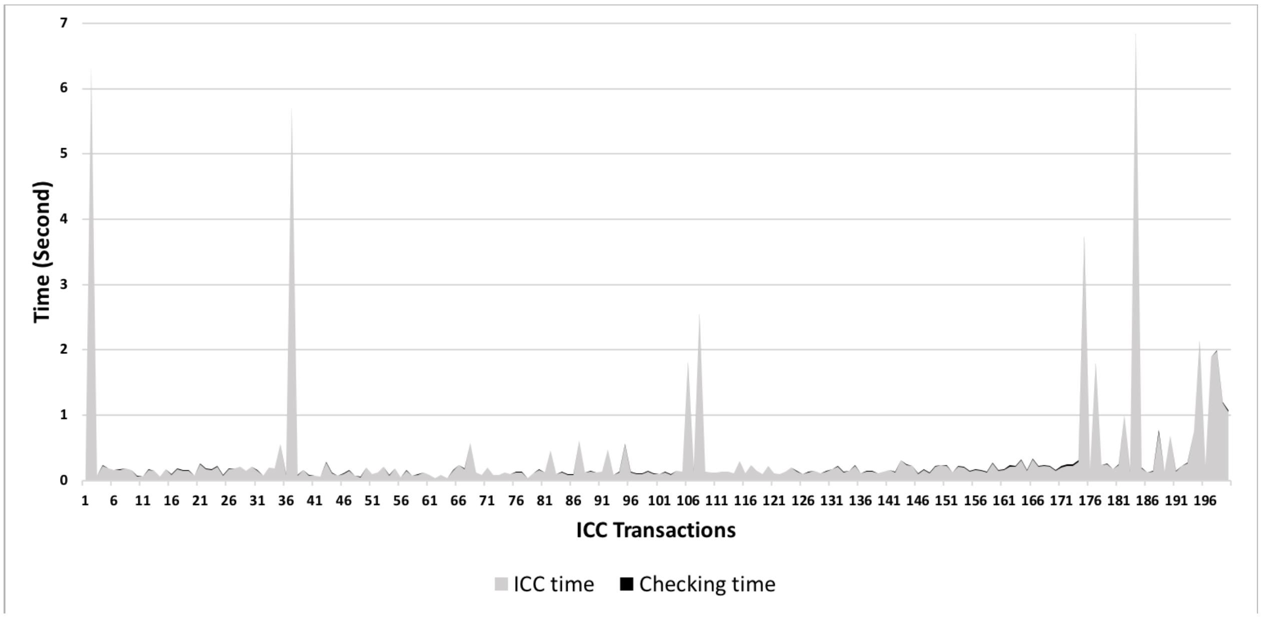


Figure 9: The performance overhead for validating ICC transactions.

obtain the ECA rules, but the great majority of this time is spent in the one-time effort of recovering the architecture of an Android 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 transaction on a Nexus 5X phone. To that end, we created a script that sends 200 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 6.45 milliseconds with 5.35 milliseconds standard deviation, which accounts for 3.95% performance overhead as depicted in Figure 9. Most users cannot perceive delays of this magnitude, per Android development guidelines [50], and thus, we believe DelDroid poses an acceptable overhead.

6.5. Threats to Validity

We provide an overview of the threats to validity of our experimental setup and the evaluation results as well as the actions we have taken to mitigate these threats.

One threat to validity of our work is whether the obtained results can be generalized to apps outside our study. To mitigate this threat, we derived benign, vulnerable, and malicious apps from diverse sources. Benign apps vary across application domains (see Figure 8), application popularity (see Figure 7), and in terms of app size [51]. For example, *Gemmy Lands* app is one of the included apps in our dataset. The size of this app is 57MB and it has 10,000,000 downloads with 4.5 star-rating [52]. The vulnerable apps in our study have been discovered and verified in a previous study [45]. Similarly, our malicious apps are drawn from repositories containing apps manually labeled as malicious by security experts.

A threat regarding RQ4 is the selection of Nexus 5X phone to measure the performance of DelDroid at runtime. The runtime performance using another Android device might be different than the reported one. However, since this device has been released in 2015, it is not the most advanced Android device. Therefore, we believe that the reported performance would be similar or even better on the currently available Android devices in the market.

Finally, the reported accuracy of DelDroid, in terms of precision and recall, depends on the quality of our experimental dataset, e.g., whether vulnerabilities and attacks are representive of true vulnerabilities and attacks in real world. To reduce this threat and to also challenge DelDroid, we did not use benchmarks that contain hand-crafted apps such as DroidBench [53] or ICC-Bench [54], instead we used real-world benign and malicious Android apps with security attacks implemented by experts from outside of our research group.

7. Limitations of DELDroid

There are of course limitations in our approach. Despite numerous benefits of giving the security architect the ability to adjust the architecture, including the ability to grant/revoke privileges to/from the apps based on their corresponding trust level, such manual adjustments are subject to unintentional errors. For instance, the architect's revision of the system may result in granting unnecessary permissions, which in turn breaks the principle of least privilege, or revoking a necessary permission, which may lead to an app malfunction. To reduce the risk of such an error-prone human intervention, we recommend limiting it to situations where the adjustments are necessary; recall from Section 2 that the manual adjustment feature is entirely optional in DelDroid and the enforcement process can exclusively rely on automatically determined least-privilege architecture.

Although DelDroid is compatible with the existing apps, the user needs to install our modified version of Android on a mobile device, which potentially voids the manufacturer warranty. Conceivably, DelDroid could be adopted in future versions of Android or by Original Equipment Manufacturer companies, e.g., Samsung and Huawei, for installation on devices.

Another limitation of our approach is the possible false positives our approach may produce. These possible false positives are due to two facts. The first fact is that the current prototype implementation of DelDroid does not support analysis of dynamically loaded code. We believe a fruitful avenue of future research is to complement DelDroid with dynamic analysis techniques that can check the integrity of loaded code [2] and hence reducing the possible false positives.

The second fact is that the static analysis tools [3, 25, 53] that DelDroid relies upon are not (1) path-sensitive and (2) they cannot analyze obfuscated code nor ICC calls made by native binaries within an Android app leading to possible false positives. Our future work involves integration of dynamic analysis techniques as well as analysis of native binaries to effectively support recovery of the architecture from, and enforcing policies on, those aspects of the system.

This paper introduces a technique that broadly supports detection and mitigation of a wide range of ICC-based vulnerabilities [1, 22]. Android apps, however, can communicate through other types of mechanisms, including remote procedure calls. While this paper provides substantial supporting evidence for addressing permission-induced vulnerabilities that arise due to the Intent-based event messaging—shown to be the primary communication mechanism in Android—it would be interesting to see how DelDroid fares when applied to other types of vulnerabilities, which forms a thrust of our future work.

8. Related Work

A large body of research has focused on Android security. Here, we provide a discussion of the related efforts in light of our research.

Much work focuses on performing program analysis over Android applications for security [55]. Epic [56] is a static analysis technique for detecting ICC attacks in Android apps. CHEX [57] is a static analysis tool for detecting component hijacking vulnerabilities. FlowDroid [53] is another precise static taint analysis approach for Android apps. Chin et al. [1] discussed several ICC attacks that can be achieved through receiving an Intent by unauthorized receipt or spoofing an Intent, and they have provided ComDroid, a tool that is meant to be used by developers to analyze their apps before releasing them. Felt et al. [22] studied permission re-delegation security attacks (aka, privilege escalation) in mobile systems and web browsers; they showed the wide spread of this attack and provided an IPC inspection mechanism to prevent such attacks. ScanDroid [58] is a data-centric static analysis tool for reasoning about the data flow in Android apps; it creates security specifications from the app's manifest file. These studies focus on a single app or require the source code for their analysis. Moreover, all of these studies are architecture-agnostic.

Numerous techniques have been developed for ICC analysis [3, 45, 59, 60]. DidFail [59] introduces an approach for tracking data flows between Android components. IccTA, similarly, leverages an Intent resolution analysis to identify inter-component privacy leaks [45]. Amandroid [60] is a taint static analysis tool for detecting Intent-based data leak and data injection. Along the same line, COVERT [3] presents an approach for compositional analysis of Android inter-app vulnerabilities. More recently, LetterBomb [4] presents an approach for automatic exploit generation for vulnerabilities exposed in an Android app's Intent-based interface. 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 determination 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 [61, 62, 63, 64, 65, 66]. SEPAR [61] is a recent work for automatic synthesis and enforcement of security policies allowing the end-users to safeguard the apps installed on their devices from ICC attacks. SEPAR's policy enforcement relies on the Xposed framework [67] that requires root access to the device. Further, unlike our approach, SEPAR cannot prevent malicious hidden behaviors. Kirin [64] extends the application installer component of Android's middleware to check the permissions requested by applications against a set of security rules. These predefined rules are aimed to prevent unsafe combination of permissions that may lead to insecure data flows. Kynoid [63] performs a dynamic taint analysis over a modified version of Dalvik VM. DeepDroid [65] presents an enforcement extensions based on dynamic memory instrumentation of system processes. ASM (Android Security Modules) [66] is a framework that provides a programmable interfaces for defining reference monitors for Android similar to the proposed reference monitors for Linux [68] and TrustedBSD [69]. 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.

The importance of limiting the privileges assigned to Android components have also been discussed in the literature [9, 10, 28, 70, 71, 72, 73, 74, 75]. Kantola et al.[28] described heuristics to allow the Android framework distinguish between inter-app and intra-app communications and hence detect any unintentional inter-app communication. Unlike DelDroid, the proposed heuristics are not totally backward compatible with the existing apps and they require modifications by the apps' developers. Shehab and AlJarrah [70] proposed a policy-based approach for controling the access of different pages in web-based Android apps to mitigate potential attacks. However, unlike DelDroid, their approach requires source code and it is limited only to web-based multi-page apps generated by the Apache Cordova framework [76]. Wang et al. [9] proposed Compac, an approach for reducing the permissions assigned for third-party components in an app. Similar to Compac [9], FlexDdroid [10] is an Android security model and isolation mechanism for limiting the permissions granted to third-party libraries. Dietz et al. [73] presented Quire, an approach that adds two security mechanisms into Android to prevent privilege escalation attack. The first security mechanism tracks the inter-process communications (IPCs) in a device to either allow an app to run with reduced privilege of its caller or with its full privileges by acting explicitly on its own behalf. The second security mechanism allows an app to create a signed statement that can be verified by any app on the same phone. Shekhar et al. developed AdSplit [74] on top of Quire. AdSplit is an approach that runs an advertising library and its hosting app in separate processes with different user identifiers. This separation eliminate the need for an app and its advertising library to share the same permissions. Similar to AdSplit, AdDroid [75] introduces advertising API and corresponding advertising permissions as part of the Android platform. AdDroid allows for permission separation between advertising libraries and their hosting apps. Unlike DelDroid, these approaches do not control interactions among components and they also require developer intervention to modify their apps, significantly hindering their adoption in practice.

Schmerl et al. [77] describe an architectural style for Android in ACME [78] 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. [79], and is well recognized by many researchers. Notably, Scandariato et al. [80] 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.

9. Conclusion

Many autonomous and smart software systems, particularly those intended for execution in mobile and IoT settings, are developed and deployed on top of Android. As such systems permeate every facet of our society, their security grows in prominence. 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.

DelDroid utilizes static analysis techniques to automatically extract the inter-component communication and resource-access privileges each component needs to fulfill its task. The determined LP architecture is elegantly represented as an MDM matrix. This representation further allows a security architect to adjust the identified LP architecture as needed to establish the proper privileges for each component. DelDroid, finally, enforces automatically obtained/expert-supplied LP architecture at runtime, governing privileges obtained by each component as prescribed by the architecture.

Our experiments on hundreds of real-world apps show between 94% 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%.

Android apps increasingly use both dynamically loaded code and native binaries. Being able to model those aspects of the apps in MDMs and building related security rules for their associated vulnerabilities, along with modeling the interactions among managed and native code in MDMs can provide further attack detection and prevention. At the same time, it may complicate analyses and in turn may lead to scalability issues. Such challenges constitute interesting avenues of future work.

Our research artifacts, including tools and evaluation data, are available publicly [51].

10. Acknowledgment

This work was supported in part by awards CCF-1755890, 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] E. Chin, A. P. Felt, K. Greenwood, D. Wagner, Analyzing inter-application communication in Android, in: International Conference on Mobile Systems, Applications, and Services, ACM, Bethesda, Maryland, 2011.

[2] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, G. Vigna, Execute this! analyzing unsafe and malicious dynamic code loading in Android applications, in: NDSS, San Diego, California, 2014.

[3] H. Bagheri, A. Sadeghi, J. Garcia, S. Malek, Covert: Compositional analysis of Android inter-app permission leakage, IEEE Transactions on Software Engineering 41 (9) (2015) 866–886.

[4] J. Garcia, M. Hammad, N. Ghorbani, S. Malek, Automatic generation of inter-component communication exploits for Android applications, in: Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2017), ACM, PADERBORN, GERMANY, 2017, pp. 661–671.

[5] H. Bagheri, J. Wang, J. Aerts, S. Malek, Efficient, evolutionary security analysis of interacting android apps, in: Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 386–397.

[6] P. Pearce, A. P. Felt, G. Nunez, D. Wagner, Addroid: Privilege separation for applications and advertisers in Android, in: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, ACM, 2012, pp. 71–72.

[7] S. Shekhar, M. Dietz, D. S. Wallach, AdSplit: Separating Smartphone Advertising from Applications, in: T. Kohno (Ed.), Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012, SEC'12, USENIX Association, Bellevue, WA, 2012, pp. 553–567.

[8] M. Sun, G. Tan, NativeGuard: protecting Android applications from third-party native libraries, in: G. Ács, A. Martin, I. Martinovic, C. Castelluccia, P. Traynor (Eds.), 7th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec'14, Oxford, United Kingdom, July 23-25, 2014, WISEC'14, ACM, 2014, pp. 165–176.

[9] Y. Wang, S. Hariharan, C. Zhao, J. Liu, W. Du, Compac: Enforce component-level access control in Android, in: Fourth ACM Conference on Data and Application Security and Privacy (CODASPY), San Antonio, TX, 2014.

[10] J. Seo, D. Kim, D. Cho, I. Shin, T. Kim, Flexdroid: Enforcing in-app privilege separation in android., in: The Network and Distributed System Security Symposium (NDSS), San Diego, CA, 2016.

[11] R. N. Taylor, N. Medvidovic, E. Dashofy, Software architecture: foundations, theory, and practice, Wiley Publishing, 2009.

[12] U. Lindemann, M. Maurer, Facing multi-domain complexity in product development, in: The future of product development, Springer, Berlin, Germany, 2007.

[13] M. Hammad, H. Bagheri, S. Malek, Determination and Enforcement of Least-Privilege Architecture in Android, in: IEEE International Conference on Software Architecture (ICSA), IEEE, Gothenburg, Sweden, 2017, pp. 59–68.

[14] Smartphone os market share, 2017 q1, http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[15] Number of available apps in the google play store, `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`.

[16] S. Bugiel, S. Heuser, A.-R. Sadeghi, Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies, in: USENIX Security Symposium, Washington DC, 2013.

[17] S. Smalley, R. Craig, Security Enhanced (SE) Android: Bringing Flexible MAC to Android, in: NDSS, The Internet Society, San Diego, California, 2013.

[18] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, T. Tanaka, A Small But Non-negligible Flaw in the Android Permission Scheme, in: Int'l Symp. on Policies for Distributed Systems and Networks, Fairfax, VA, 2010. `doi:10.1109/POLICY.2010.11`.

[19] Z. Fang, W. Han, Y. Li, Permission based Android security: Issues and countermeasures, Computers & Security 43 (2014) 205–218. `doi:10.1016/j.cose.2014.02.007`.

[20] A. Egners, U. Meyer, B. Marschollek, Messing with Android's permission model, in: Int'l Conf. on Trust, Security and Privacy in Computing and Communications, Liverpool, United Kingdom, 2012.

[21] L. Davi, A. Dmitrienko, A.-R. Sadeghi, M. Winandy, Privilege escalation attacks on Android, in: Int'l Conf. on Information Security, Boca Raton, FL, 2010.

[22] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, E. Chin, Permission re-delegation: Attacks and defenses., in: USENIX Security Symposium, San Francisco, California, 2011.

[23] K. Coogan, S. Debray, T. Kaochar, G. Townsend, Automatic static unpacking of malware binaries, in: Working Conf. on Reverse Engineering, Washington, DC, 2009.
URL `http://dx.doi.org/10.1109/WCRE.2009.24`

[24] Apktool: A tool for reverse engineering Android apk files, `https://ibotpeaches.github.io/Apktool/`.

[25] D. Octeau, D. Luchaup, M. Dering, S. Jha, P. McDaniel, Composite constant propagation: Application to Android inter-component communication analysis, in: Int'l Conf. on Software Engineering, IEEE, Florence, Italy, 2015.

[26] K. W. Y. Au, Y. F. Zhou, Z. Huang, D. Lie, Pscout: analyzing the Android permission specification, in: ACM CCS, Raleigh, NC, 2012.

[27] D. V. Steward, The design structure system: A method for managing the design of complex systems, IEEE transactions on Engineering Management (3) (1981) 71–74.

[28] D. Kantola, E. Chin, W. He, D. Wagner, Reducing attack surfaces for intra-application communication in android, in: Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices, ACM, Raleigh, NC, 2012, pp. 69–80.

[29] A. P. Felt, D. Wagner, Phishing on mobile devices, in: Web 2.0 security and privacy workshop (W2SP), IEEE, Oakland, CA, 2011.

[30] H. Bagheri, J. Garcia, A. Sadeghi, S. Malek, N. Medvidovic, Software architectural principles in contemporary mobile software: from conception to practice, Journal of Systems and Software 119 (2016) 31–44.

[31] A. Barth, C. Jackson, J. C. Mitchell, Robust defenses for cross-site request forgery, in: Proceedings of the 15th ACM conference on Computer and communications security, ACM, Alexandria, VA, 2008, pp. 75–88.

[32] M. C. Huebscher, J. A. McCann, A survey of autonomic computing—degrees, models, and applications, ACM Computing Surveys (CSUR) 40 (3) (2008) 7.

[33] N. Bencomo, S. Hallsteinsen, E. S. De Almeida, A view of the dynamic software product line landscape, Computer 45 (10) (2012) 36–41.

[34] J. Kramer, J. Magee, Self-managed systems: an architectural challenge, in: 2007 Future of Software Engineering, IEEE Computer Society, 2007, pp. 259–268.

[35] J. Widom, S. Ceri, Active database systems: Triggers and rules for advanced database processing, Morgan Kaufmann, 1996.

[36] N. W. Paton, O. Díaz, Active rules in database systems, in: Active Rules in Database Systems, Springer, 1999, pp. 3–27.

[37] S. Abiteboul, V. Vianu, B. Fordham, Y. Yesha, Relational transducers for electronic commerce, Journal of Computer and System Sciences 61 (2) (2000) 236–269.

[38] S. Ceri, P. Fraternali, Designing database applications with objects and rules: the IDEA Methodology, Addison-Wesley, 1997.

[39] F. Bry, M. Eckert, P.-L. Pătrânjan, I. Romanenko, Realizing business processes with eca rules: Benefits, challenges, limits, in: International Workshop on Principles and Practice of Semantic Web Reasoning, Springer, 2006, pp. 48–62.

[40] G. Papamarkos, A. Poulovassilis, P. T. Wood, Event-condition-action rule languages for the semantic web, in: Proceedings of the First International Conference on Semantic Web and Databases, Citeseer, 2003, pp. 294–312.

[41] E. Behrends, O. Fritzen, W. May, F. Schenk, Combining eca rules with process algebras for the semantic web, in: Rules and Rule Markup Languages for the Semantic Web, Second International Conference on, IEEE, 2006, pp. 29–38.

[42] AOSP: Android open source project, `https://source.android.com/`.

[43] Fastboot: A special diagnostic and engineering protocol for booting Android devices., https://source.android.com/source/running.html.

[44] Adb: Android debug bridge, https://developer.android.com/studio/command-line/adb.html.

[45] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, P. McDaniel, Iccta: Detecting inter-component privacy leaks in Android apps, in: Int'l Conf. on Software Engineering, IEEE, Florence, Italy, 2015.

[46] Y. Zhou, X. Jiang, Dissecting Android malware: Characterization and evolution, in: IEEE Symposium on Security and Privacy, IEEE, San Francisco, California, 2012, pp. 95–109.

[47] Contagio malware repository, http://contagiodump.blogspot.it.

[48] F. Maggi, A. Valdi, S. Zanero, 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, 2013.

[49] 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.

[50] Keeping your app responsive, https://developer.android.com/training/articles/perf-anr.html,.

[51] DELDroid website, http://www.ics.uci.edu/~seal/projects/deldroid.

[52] Gemmy lands app, https://play.google.com/store/apps/details?id=com.nevosoft.mylittleplanet.

[53] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, 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, 2014.

[54] Icc bench, https://github.com/fgwei/ICC-Bench.

[55] A. Sadeghi, H. Bagheri, J. Garcia, S. Malek, A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software, IEEE Transactions on Software Engineering 43 (6) (2017) 492–530.

[56] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, Y. Le Traon, Effective inter-component communication mapping in Android: An essential step towards holistic security analysis, in: USENIX Security Symposium, Washington DC, 2013.

[57] L. Lu, Z. Li, Z. Wu, W. Lee, G. Jiang, Chex: statically vetting Android apps for component hijacking vulnerabilities, in: conference on Computer and communications security, ACM, New York, NY, 2012.

[58] A. P. Fuchs, A. Chaudhuri, J. S. Foster, Scandroid: Automated security certification of Android, University of Maryland, Tech. Rep. CS-TR-4991.

[59] W. Klieber, L. Flynn, A. Bhosale, L. Jia, L. Bauer, Android taint flow analysis for app sets, in: International Workshop on the State of the Art in Java Program Analysis, ACM, Edinburgh, United Kingdom, 2014.

[60] F. Wei, S. Roy, X. Ou, Robby, Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps, in: ACM CCS, Scottsdale, Arizona, 2014.

[61] H. Bagheri, A. Sadeghi, R. Jabbarvand, S. Malek, Practical, formal synthesis and automatic enforcement of security policies for android, in: Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Toulouse, France, 2016, pp. 514–525.

[62] A. Sadeghi, R. J. Behrouz, N. Ghorbani, H. Bagheri, S. Malek, A temporal permission analysis and enforcement framework for android, in: Proceedings of the 40th International Conference on Software Engineering (ICSE), 2018, pp. 846–857.

[63] D. Schreckling, J. Köstler, M. Schaff, Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for Android, Information Security TR. 17 (3) (2013) 71–80.

[64] W. Enck, M. Ongtang, P. McDaniel, On lightweight mobile phone application certification, in: Proceedings of the 16th ACM conference on Computer and communications security, Chicago, Illinois, 2009.

[65] X. Wang, K. Sun, Y. Wang, J. Jing, Deepdroid: Dynamically enforcing enterprise policy on Android devices., in: NDSS, San Diego, California, 2015.

[66] S. Heuser, A. Nadkarni, W. Enck, A.-R. Sadeghi, Asm: A programmable interface for extending Android security, in: USENIX Security Symposium, San Diego, California, 2014.

[67] Xposed module repository, `http://repo.xposed.info/`.

[68] J. Morris, S. Smalley, G. Kroah-Hartman, Linux security modules: General security support for the linux kernel, in: USENIX Security Symposium, ACM, Berkeley, CA, 2002.

[69] R. N. Watson, Adding trusted operating system features to freebsd, in: USENIX Technical Conference, Boston, MA, 2001.

[70] M. Shehab, A. AlJarrah, Reducing attack surface on cordova-based hybrid mobile apps, in: Proceedings of the 2nd International Workshop on Mobile Development Lifecycle, Portland, Oregon, 2014.

[71] H. Bagheri, E. Kang, S. Malek, D. Jackson, A formal approach for detection of security flaws in the android permission system, Formal Aspects of Computing 30 (5) (2018) 525–544.

[72] H. Bagheri, E. Kang, S. Malek, D. Jackson, Detection of design flaws in the Android permission protocol through bounded verification, in: FM 2015: Formal Methods, Vol. 9109 of Lecture Notes in Computer Science, 2015, pp. 73–89.

[73] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, D. S. Wallach, Quire: Lightweight provenance for smart phone operating systems., in: USENIX Security Symposium, Vol. 31, San Francisco, California, 2011.

[74] S. Shekhar, M. Dietz, D. S. Wallach, Adsplit: Separating smartphone advertising from applications., in: USENIX Security Symposium, Vol. 2012, Bellevue, WA, 2012.

[75] P. Pearce, A. P. Felt, G. Nunez, D. Wagner, Addroid: Privilege separation for applications and advertisers in android, in: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, Acm, Seoul, Republic of Korea, 2012, pp. 71–72.

[76] Apache cordova: develop mobile apps with html, css and js, `https://cordova.apache.org/`.

[77] B. Schmerl, J. Gennari, A. Sadeghi, H. Bagheri, S. Malek, J. Cámara, D. Garlan, Architecture modeling and analysis of security in Android systems, in: European Conference on Software Architecture, Copenhagen, Denmark, 2016.

[78] D. Garlan, R. Monroe, D. Wile, Acme: An architecture description interchange language, in: CASCON First Decade High Impact Papers, IBM Corp., Toronto, ON, Canada, 2010, pp. 159–173.

[79] J. H. Saltzer, M. D. Schroeder, The protection of information in computer systems, IEEE Computer Society Press 63 (9) (1975) 1278–1308.

[80] R. Scandariato, K. Buyens, W. Joosen, Automated detection of least privilege violations in software architectures, in: European Conference on Software Architecture, Copenhagen, Denmark, 2010.

Mahmoud Hammad is an Assistant Professor in the Software Engineering Department at Jordan University of Science and Technology. He received his Ph.D. in Software Engineering from the University of California, Irvine on August of 2018 under the supervision of Dr. Sam Malek. Hammad received his M.S.c. in Software Engineering from George Mason University in 2013, and his B.S.c. in Computer Science from Yarmouk University in 2005. He conducts research in software engineering with a focus on mobile security, software architecture, and autonomic computing. He is a member of ACM and ACM SIGSOFT.

Hamid Bagheri is an Assistant Professor in the Department of Computer Science and Engineering at University of Nebraska-Lincoln. He is a co-director of the ESQuaReD Laboratory at UNL. Prior to joining UNL, he was a project scientist at University of California, Irvine, and also a postdoctoral research fellow at MIT. He obtained his PhD in Computer Science from University of Virginia, the M.Sc. in Software Engineering from Sharif University of Technology, and his B.Sc. in Computer Engineering from University of Tehran. His research interest lies in advancing software reliability through practical software analysis and synthesis.

Sam Malek is an Associate Professor in the School of Information and Computer Sciences at the University of California Irvine (UCI). He is also Director of the Institute for Software Research at UCI. He received the B.S. degree in Information and Computer Science from the University of California, Irvine, and the MS and Ph.D. degrees in Computer Science from the University of Southern California. His general research interests are in the field of software engineering, and to date his focus has spanned the areas of software architecture, autonomic computing, software security, and software analysis and testing.