Technische Universität Darmstadt

Center for Advanced Security Research Darmstadt

Technical Report TR-2011-04

XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko,
Thomas Fischer, Ahmad-Reza Sadeghi

System Security Lab
Technische Universität Darmstadt, Germany

Technische Universität Darmstadt
Center for Advanced Security Research Darmstadt
D-64293 Darmstadt, Germany

TR-2011-04
First Revision: April 30, 2011
Last Update: June 30, 2011

XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks

Sven Bugiel
sven.bugiel@cased.de

Lucas Davi
lucas.davi@cased.de

Alexandra Dmitrienko
alexandra.dmitrienko@cased.de

Thomas Fischer
thomas.fischer@rub.de

Ahmad-Reza Sadeghi
ahmad.sadeghi@cased.de

System Security Lab
Center for Advanced Security Research Darmstadt (CASED)
Germany

ABSTRACT

Google Android has become a popular mobile operating system which is increasingly deployed by mobile device manufactures for various platforms. Recent attacks show that Android's permission framework is vulnerable to application-level privilege escalation attacks, i.e., an application may indirectly gain privileges to perform unauthorized actions. The existing proposals for security extensions to Android's middleware (e.g., Kirin, Saint, TaintDroid, or QUIRE) cannot fully and adequately mitigate these attacks or detect Trojans such as *Soundcomber* that exploit covert channels in the Android system. In this paper we present the design and implementation of *XManDroid* (eXtended Monitoring on Android), a security framework that extends the monitoring mechanism of Android to detect and prevent application-level privilege escalation attacks at *runtime* based on a system-centric system policy. Our implementation dynamically analyzes applications' transitive permission usage while inducing a minimal performance overhead unnoticeable for the user. Depending on system policy our system representation allows for an effective detection of (covert) channels established through the Android system services and content providers while simultaneously optimizing the rate of false positives. We evaluate the effectiveness of *XManDroid* on our test suite that simulates known application-level privilege escalation attacks (including Soundcomber), and demonstrate successful detection of attacks that use Android's inter-component communication (ICC) framework (standard for most attacks). We also performed a usability test to evaluate the impact of XManDroid on the user-experience with third party applications. Moreover, we analyze sources of false positives and discuss how this rate can be further significantly reduced.

1. INTRODUCTION

Google Android [1] is a modern operating system for smartphones with rapidly expanding market share[1] [18]. The popularity and open source character of Android facilitates its deployment on other hardware platforms, e.g., netbooks [20] and tablet PCs [3, 32].

The core security mechanisms of Android are application sandboxing, application signing, and a permission framework to control access to (sensitive) resources. The standard Android permission system limits access to sensitive data (SMS, contacts, etc.), resources (battery or log files) and to system interfaces (Internet connection, GPS, GSM). Once granted (by the end-user) the assigned permissions cannot be changed afterwards, and they are checked by the Android's reference monitor at runtime. This approach restricts the potential damage imposed by compromised applications.

However, Android's security framework exhibits serious shortcomings: On the one hand, the burden of approving application permissions is delegated to the end-user who in general does not have the appropriate skills. Hence, malware and Trojans can be installed on end-user devices as shown by very recent Android Trojans: such as unauthorized sending of text messages [25], malicious game updates [21], or location tracking and leaking of sensitive data in the background of running games [27].

On the other hand, Android's security framework is vulnerable to *privilege escalation attacks at the application-level* which are the main focus of this paper.

Application-level privilege escalation attacks.

The recent privilege escalation attacks [13, 8, 26, 37] show that in contrast to the general belief the damage imposed by Android malware is not limited to the application's sandbox. An unprivileged application under the adversary's control can perform operations indirectly by invoking other applications possessing desired privileges. The attacks published so far range from unauthorized phone calls [13] and text message sending [8] to illegal downloads of malicious files [26] and context-aware voice recording [37, 23].

Most privilege escalation attacks exploit vulnerable interfaces of privileged applications. This attack is often referred to as *confused deputy attack* [24, 9]. However, in general, the adversary can design his own malicious applications which collaborate to mount a *collusion attack* [37]: Although each application has a limited set of uncritical permissions, they can collude to generate a joint set of permissions that enables them to perform unauthorized malicious actions. Moreover, recent privilege escalation attacks based on colluding applications (in particular, Soundcomber [23]) may exploit covert or overt channels of the Android core system to avoid detection from tools only inspecting direct application communication.

Note that the Android application distribution model allows anyone who has registered as an Android developer (and paid $25 fee) to publish applications on the Android

[1]Recently, the Android OS (36.0% market share) has overtaken Symbian (27.4% market share) and Apple's iPhone OS (16.8% market share) and is now the most popular mobile OS in the world.

market. This scheme allows adversaries to easily upload malicious applications on the market store: For instance, the recent Android DroidDream Trojan (containing a root exploit) has been identified in over 50 official Android market applications and has been downloaded more than 10,000 times before it has been detected [4]. In the light of recent Android Trojans [31, 21, 27, 4] (that also enable the installation of additional applications), it seems that collusion attacks will be increasingly deployed as soon as platforms enforce malware mitigation techniques (e.g., [13, 14]) to detect malware comprised in a single application.

Security extensions to Android.

Over the last years, several sophisticated security extensions to Android's security framework have been proposed [13, 14, 35, 38, 30, 11, 34, 36, 6, 9, 17]. However, to the best of our knowledge, and as we will elaborate on related work in Section 7, none of these solutions accurately addresses application-level privilege escalation attacks. In particular, tools such as Kirin [13, 14], Saint [35], and TaintDroid [11] that are the closest to our proposal fall short in combating these attacks. However, a recent very interesting work has been presented by Dietz et al. [9]: QUIRE is a lightweight provenance system that prevents the so-called confused deputy attack. Their approach is complementary to ours, however, QUIRE does not address privilege escalation that are based on maliciously colluding applications. Finally, none of the existing solutions addresses the recent privilege escalation attack that exploits covert channels of the Android system [37, 23].

Our goal and contributions.

We address the problem of privilege escalation attacks at the application-level in Android system. A very challenging issue is to effectively detect/prevent these attacks while avoiding high false positive rate which heavily limits the usability. Especially, some core Android components (e.g., content provider) provide various services to many other applications and are often used for legitimate tasks that should not be wrongly identified as attacks. In particular, our contributions are the following:

- We present the design and implementation of *XManDroid* (eXtended Monitoring on Android), a framework which extends the Android's reference monitor and enables *runtime* monitoring of communication links between applications on Android and verifies them against security rules defined in a system policy. In contrast to tools like Kirin [13, 14] (which verifies permissions requested by the applications at install-time), our solution is able to handle exceptional cases such as *pending intents*[2] and communication links among dynamically created components (e.g., *Broadcast Receivers*).

- Moreover, depending on the system policy, our system representation allows for an effective detection/prevention of (covert) channels that are established through the Android system services and system content providers, while simultaneously minimizing the rate of false positive policy decisions.

- Our reference implementation of *XManDroid* is efficient and requires only minimal performance overhead not noticeable to the user. We performed authomated testing of *XManDroid* on a NexusOne Dev Phone with 50 applications from the Android Market. Note that in contrast to [12, 16] we perform our evaluation at runtime, and hence, 50 applications are already sufficient for our purposes. Our measurements show that *XManDroid* requires on average a time overhead of 13.13 ms for mediating an uncached communication request and 0.11 ms for an already cached request. In addition, we conducted a usability test with 20 students to evaluate the user experience (with false positive rate).

- To show the effectiveness of our framework, we implemented and applied a malware test suite that simulates the known application-level privilege escalation attacks [13, 8, 26, 37]. In contrast to the existing solutions, *XManDroid* detects all escalation attacks which use Android's inter-component communication (ICC) framework for interaction, including attacks of Soundcomber [37, 23] (as long as they use ICC) which exploits covert channels in the Android system.

At this stage, we stress the following three aspects: First, we do not consider privilege escalation attacks at the *kernel-level* that exploit bugs in the system kernel to gain root access to the system (see [26, 33]), or application-level privilege escalation attacks which occur over the channels controlled by the underlying kernel. Such attacks can be mitigated, e.g., through the deployment of SELinux [28], which has been shown to be feasible on Android [38][3]. Second, as for any monitoring system the effectiveness and accuracy of *XManDroid* also depends on the underlying system policy. To evaluate our framework we aimed at deploying more fine-grained policy rules and policy matching partially inspired by the related work. However, we do not claim completeness of policy engineering which is not the scope of this paper. Third, we do not address a problem of malware comprising of a single application sandbox. This problem is orthogonal to privilege escalation, as applications within a single sandbox have equal privileges and cannot preform privilege escalation[4].

We believe that *XManDroid* complements other existing security extensions: It can be built on top of SELinux [38] to cover privilege escalation attacks at the *kernel-level*; it can be combined with Kirin [13, 14] to mitigate malware (comprised within a single application); with TaintDroid [11] to prevent leakage of privacy-sensitive data, and with Saint [35] to allow more comprehensive permission definitions for application interfaces. Furthermore, it can be complemented by QUIRE [9] to enable RPC (Remote Procedure call) checking among distributed devices.

Outline.

The remainder of this paper is organized as follows: After we recall the Android architecture in Section 2), we introduce

[2]Intents are messages used for inter-application communication on Android. A pending intent is an intent where the intent contents do not originate from the sending application (delegation).

[3]In the light of current reports, which show that Android's underlying Linux kernel suffers from various (precisely 88) bugs [5], it seems to be strongly necessary to enable SELinux on Android.

[4]Preventing combinations of security critical permissions in a single sandbox is an install-time problem. Kirin [14] could principally prevent these attacks, however, it should be enhanced to evaluate permissions of a sandbox rather than of a single application.

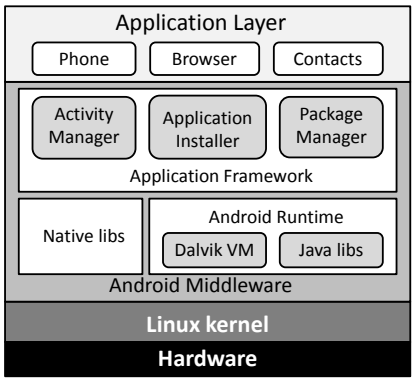

Figure 1: Android architecture

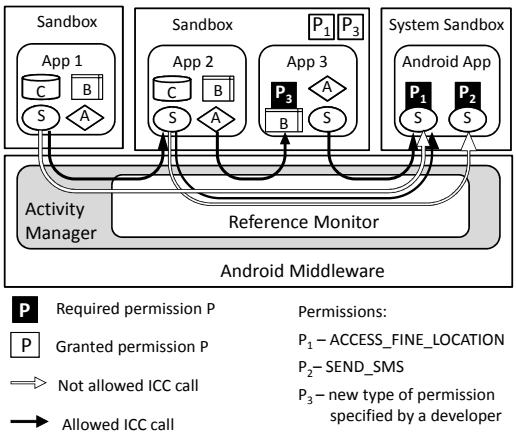

Figure 2: Android security mechanisms: Application sandboxes, ICC calls and permission assignments

the general problem of privilege escalation attacks, our adversary model and assumptions in Section 3. In Section 4 we present our design decisions and in particular describe how *XManDroid* deals with covert channel attacks. In Section 5 we present the implementation of *XManDroid* and evaluate its effectiveness and performance in Section 6. Finally, we elaborate on related work in Section 7, and conclude in Section 8.

2. ANDROID

In this section we briefly recall some basics of the Android architecture and security mechanisms.

2.1 Android Architecture

Android is an open source software platform for mobile devices. It includes a Linux kernel, a middleware framework, and an application layer (see Figure 1). The Linux kernel provides device drivers and low-level services to the rest of the system. A middleware layer consists of native Android libraries (written in C/C++), Android runtime module and an application framework. The application framework consists of system applications written in C/C++ or Java which provide system services, e.g., Activity Manager manages the life cycle of applications, Application Installer installs new applications, while Package Manager maintains information about all applications loaded in the system. Android Runtime includes an optimized version of a Java Virtual Machine called Dalvik Virtual Machine (DVM) and core Java libraries Java libs. The DVM executes binaries of applications residing in the application layer and system applications.

Android application layer includes core (i.e., installed by default) applications such as Browser, the dialer Phone and the contact provider Contacts. Android applications are written in Java, but they can also incorporate C/C++ native libraries through the Java Native Interface (JNI). Applications consist of separated modules, so-called *components*. There are four basic types of components: *Activities* (A), *Services* (S), *Content Providers* (C) and *Broadcast Receivers* (B) (as shown in Figure 2). *Activities* are responsible for the user interface, typically each screen shown to a user is represented by a single *Activity* component. *Services* implement functionality of background processes which are invisible to the user. *Content Providers* are special purpose components which are used for sharing data among applications. *Broadcast Receivers* serve for receiving event notifications from the system and from other applications.

Components can communicate to each other and to components of other applications through an *inter-component communication* (ICC) mechanism provided by the middleware. Applications initiate ICC links by sending a special message called *Intent*. One can differ four types of ICC: (i) starting activities, (ii) intent broadcasts, (iii) accessing content provides, and (iv) binding components to services Directed ICC calls may result in data flow in opposite direction. For instance, the ICC call querying a content provider with a read query will return the requested data, or the ICC call binding component to a service can result in establishing bi-directional data interface.

2.2 Android Security Mechanisms

In the following we briefly describe the core security mechanisms of Android: (i) sandboxing, (ii) permission framework and (iii) application signing. More detailed information can be found in [13, 15, 40, 39].

Sandboxing.

Sandboxing is a mechanism to isolate applications from each other and from system resources. Application isolation is done by means of assigning a unique user identifier (UID) to each application, while the underlying Linux kernel enforces discretionary access control to resources (files and devices) by user ownership. System resources are owned by either *system* or *root*. Applications can only access own files or files of others that are explicitly marked as world-wide readable.

Application Signing.

Android enforces application signing, however, not centrally, i.e., developers themselves have to sign the application code with the self-certified key. Thus, application signing does not provide protection against malware, but helps to establish trust relationships among applications originating from the same developer. Applications signed with the same key may request to share the same UID, i.e., they will be placed into the same sandbox (e.g., App 2 and App 3 in Figure 2).

Android Permission Framework.

The Android permission framework is provided by the

middleware layer. It includes a reference monitor which enforces mandatory access control (MAC) on ICC calls (see Figure 2)[5]. Security sensitive application programming interfaces (APIs), here referred to as *interfaces*, are protected by permissions. These are security labels which can either be *required* to enforce access control, or *granted* to allow access.

Granted permissions are assigned to application sandboxes and inherited by all application components. Unlike this, required permissions are assigned to application components. Both, required and granted permissions are explicitly specified in an application's *Manifest file* which is included into application installation package. Granted permissions are approved at installation time based on user confirmation. Once granted, permissions cannot be modified.

Android core services are provided by a system application called Android (see Figure 2). Access to Android components is protected by standard permissions such as ACCESS_FINE_LOCATION (P_1) and SEND_SMS (P_2) meaning that applications have to possess these permissions to be able to access the user's location or to send text messages. Additionally, application components may define and require new types of permissions in order to limit access to own interfaces (e.g., permission P_3 in Figure 2).

At runtime the reference monitor checks permission assignments. For instance, in Figure 2, components of App 2 and App 3 are able to access a component of Android providing the functionality to access user location, because their sandbox is assigned the appropriate permission ACCESS_FINE_LOCATION (P_1). However, the reference monitor denies access from components of these applications to Android providing the functionality to send text messages, because the corresponding permission SEND_SMS (P_2) is not granted to the sandbox of Apps 2 and 3.

The list of standard Android permissions includes 110 items [22], and most of them are required by Android.[6]

3. PROBLEM DESCRIPTION

In this section, we will describe the general problem of privilege escalation attacks, define our adversary model and assumptions.

Application-level privilege escalation attacks.

The general idea of a privilege escalation attack at the application-level can be illustrated in Figure 2: App 1 has no permissions to access location information, because it is not granted the corresponding permission P_1. However, it can access it transitively, e.g., through components of App 2 (two hops). Indeed, components of App 2 do not require any permissions to be accessed, and components of App 2 can access location information, because their sandbox is granted the permission P_1. This transitive flow can be used to escalate the privileges of App 1 with the goal to access location information without possessing P_1.

Privilege escalation attacks can be classified into two categories. First category includes confused deputy attacks, where applications under control of the adversary (either malicious or compromised) leverage unprotected interface of a benign application (e.g., exploit Android browser to download files [26], or misuse Android Scripting Environment to send unauthorized text messages [8]). In a second class of attacks, both applications are malicious, they collude to get objectionable set of permissions. Further, collusion attacks can be classified in those where applications communicate directly with each other, or indirectly, by using another application or a component in between. In the latter case, a mediating component can provide either covert (e.g., via power manager [37]) or overt (e.g., via a user contacts database) channels to communicating applications.

Adversary Model.

The adversary's goal is to escalate privileges and to get unauthorized access to protected interfaces. The adversary is able to exploit vulnerable interfaces of benign applications and perform confused deputy attack. Moreover, several malicious applications (two or more) can collaborate (collude) in order to obtain a merged permission set. Hence, our adversary model also includes *malicious developers*[7].

Assumptions.

We assume that the highly-privileged Android application (see Section 2) which is already provided in the default configuration of a clean device is not malicious. This is reasonable, since in general one may have more trust in genuine vendors (e.g., Google, Microsoft, and Apple) not to maliciously attack end-users. However, the Android application may suffer from design deficiencies that allow the adversary to establish covert channels and launch privilege escalation attacks [37].

Further, we consider ICC channels as the only mechanism available in Android to establish communication channels among applications. However, as we elaborate in Section 4.4, in some cases applications may communicate by other means than ICC. For instance, applications can establish covert channels which completely bypass Android's middleware layer, where our detection mechanism resides. An example of these channels are file locks [29] (also exploited in [37]). Moreover, currently we do not consider communication via overt channels controlled by Linux, such as Internet sockets or the file system. To consider these channels, kernel-level extensions of *XManDroid* are required.

4. XManDroid

In this section, we present the design and architecture of *XManDroid* (eXtended Monitoring on Android). *XManDroid* performs runtime monitoring and analysis of communication links across applications in order to prevent potentially malicious ones based on the defined policy.

The idea is as follows: *XManDroid* maintains a system state which contains the applications installed on the platform and established communication links (control and data flows) among them. *XManDroid* is invoked when the default Android reference monitor grants an ICC call, and validates whether the requested ICC call can potentially be exploited (in combination with other communication links which have ever occurred in the system) for escalation attacks (based on the underlying system policy). To minimize performance overhead, *XManDroid* stores decisions that depend

[5]Particularly, the functionality of the reference monitor is implemented within Activity Manager component.

[6]Although Android offers many different kinds of permissions, only a few of them are actively used [2].

[7]Note that the presence of several colluding malicious applications is a realistic assumption in the case of Android, since Android's code distribution policy does not hinder the registration of malicious developers: Everyone can become an authorized Android developer by simply paying a fee of $25.

on permanent conditions and thus never change over time. Those decisions are applied next time when the same ICC is requested.

A system policy consists of security rules that must be satisfied in order to prevent privilege escalation attacks. In Section 4.3 we will present policy rule examples.

4.1 Architecture

The architecture of *XManDroid* is shown in Figure 3. It extends the application framework of Android and consists of three modules: (i) Runtime Monitor, (ii) Application Installer, and (iii) System Policy Installer.

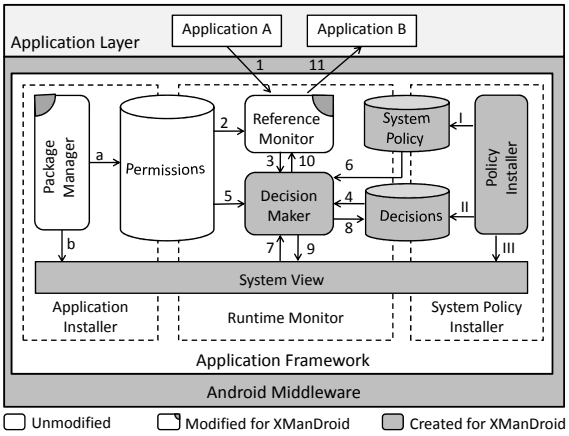

Figure 3: *XManDroid* architecture

Runtime Monitor.

The Runtime Monitor provides the core functionality of *XManDroid*. It involves the components ReferenceMonitor, DecisionMaker, SystemView, Permissions, SystemPolicy and Decisions. ReferenceMonitor is the standard reference monitor of Android, which performs permission checks specified by the standard Android permission mechanism. DecisionMaker is a component which is responsible for making a decision if the requested ICC call imposes a threat (of the privilege escalation attack). SystemView maintains the state of the running system. Permissions is a standard component of Android which maintains a permission database. SystemPolicy is a database of the system policy rules. Decisions is a database which stores decisions made by DecisionMaker.

Application Installer.

Application Installer enhances standard Android application installation procedure and is responsible for installation and un-installation of applications. It involves standard Android components PackageManager and Permissions and the new component SystemView. As installing of a new application changes the system state, PackageManager is extended to communicate with SystemView component to reflect these changes.

System Policy Installer.

System Policy Installer provides a mechanism to install (or update) the system policy into the Android middleware and involves the components PolicyInstaller, SystemPolicy, Decisions and SystemView.

4.2 Components Interaction

Figure 3 illustrates the interaction between components of *XManDroid* in three scenarios: ICC call handling (steps 1-11), application (un)installation (steps a-b) and policy installation (steps I-III).

ICC call handling.

At runtime, when ReferenceMonitor intercepts ICC call (step 1), it validates permission assignments. For that, it obtains information about permissions from Permissions database (step 2). In case it disallows the ICC call, process is terminated. Otherwise, ReferenceMonitor invokes DecisionMaker (step 3) to verify if this communication link complies to security rules defined in a system policy. DecisionMaker first requests a record corresponding to this particular ICC call from Decisions component (step 4). If the corresponding record is found, it means this ICC call has occurred before and previously made decision can be applied. Otherwise, DecisionMaker has to make a fresh decision. To make a decision, DecisionMaker requires inputs from Permissions (step 5), SystemPolicy (step 6) and SystemView (step 7). The resulting decision is stored in Decisions (step 8), and if it is positive, SystemView is updated (step 9) reflecting that a communication link exists among the components of applications of A and B. Further, DecisionMaker informs ReferenceMonitor about the decision it has made (step 10), and ReferenceMonitor either allows (step 11) or denies the ICC call.

Note, the SystemView component maintains allowed communication links which have been requested at runtime. Once an ICC is allowed, SystemView stores the corresponding communication link. SystemView state persist across reboot and can be reset only upon update of SystemPolicy.

Application (Un-)Installation.

During the installation procedure, PackageManager extracts application permissions from the *Manifest file* and stores them in Permissions (step a). This step is also typical for the standard Android application installation process. Additionally, PackageManager adds the new application in SystemView (step b).

During application un-installation, PackageManager performs the following: It revokes permission labels granted to the un-installing application from the Permissions database (step a) and removes this application from SystemView (step b).

Policy Installation.

During the policy installation process, PolicyInstaller writes and updates the system policy rules to the SystemPolicy database (step I). Next, it flushes all decisions previously made by DecisionMaker, as those may not comply to the new system policy (step II). Also, SystemView component is reset into a clean state (step III), i.e., all previously allowed communication links among applications are removed.

4.3 XManDroid Policy

Success in detection of malware by *XManDroid* depends on policies defined in a system policy database. Inadequate policy rules can result in both, overlooking malware and affecting functionality of genuine applications.

Deriving appropriate policy rules is an important and non-trivial task[8]. Promising work has been already done in this

[8]Defining a system policy for XManDroid requires expertize in security and goes beyond the skills of average users. In

(1)	An application that is notified about *incoming or outgoing calls* and can *record audio* must not communicate to an application with *network access*.
(2)	An application that can obtain *location information* must not communicate to an application that has *network access*.
(3)	An application that has *read access to the user contact database* must not communicate to an application that has *network access*.
(4)	An application that has *read access to user SMS database* must not communicate to an application that has *network access*.
(5)	An application that has no explicit *network access* must not be able to *download archived files* and *install packages*.
(6)	An application that has no explicit permission to perform *outgoing voice calls* can perform such calls only upon *user confirmation*.
(7)	An application that has no explicit permission to send *text messages* can send them only upon *user confirmation*.

Table 1: Sample policy rules to mitigate application-level privilege escalation attacks

area [13, 14]. Authors discuss realistic security requirements for mobile phones [13] and provide a methodology for deriving appropriate security policies for Android [14].

However, the discussion about security policies in [13, 14] has been centered around Android permissions, while we identified that policy rules considering only permissions is too coarse grained for our system. For instance, consider an example with the following permission-based rule: *An application must have an explicit permission to access the Internet*. This rule would restrict an application without INTERNET permission to access applications with access to the network. On one hand, such a rule would prevent a recent confused deputy attack against Android browser [26] (where a malicious application misuses the web-browser to download a malicious content/file). On the other hand, it would also affect functionality some of the benign applications, e.g., those apps that invoke the Android web-browser to open web-links promoted via advertisements.

To make policies of XManDroid more fine grained, we go beyond of permission-based approach and enable XManDroid to inspect data transferred over ICC and make policy decisions based on content of *Intents* (messages passed via ICC calls). Moreover, XManDroid policies can request the user confirmation in order to allow/deny the particular ICC call. The downside of the data-dependent policy rules is performance downgrade, as decisions taken on such rules can not be cashed (due to possible change of transferred data). Also, involving a user to confirm/deny ICC is not always a good idea, as users may not understand underlying security mechanisms and can make wrong decisions. However, as we will show in examples below, in some particular cases user confirmation is an excellent choice and does not require expert knowledge from the user.

In the following, we describe some useful illustrative policy rules for *XManDroid* (see Table 1) that are based on examples of [13, 14] and extended to consider ICC inspection and user confirmation. Our testing results show (see Section 6) that these policy rules are effective in preventing known attacks. Further, we consider policy engineering for *XManDroid* as an open area for research and plan to investigate it deeply in our further work.

Rules (1)-(5) in Table 1 define a fine-grained transitive access to the Internet interface. For instance, rules (2)-(4) concern on some privacy aspects of the user and target security objectives similar to TaintDroid [11]. These rules generally allow applications to transitively access the network, unless they have access to user private data, such as user contacts and SMS databases or user location. Rule (1) targets an eavesdropping malware similar to Soundcomber [37]. Rule (5) restricts applications without Internet access to download archived and application package files. Rule (6) restricts applications without CALL_PHONE permission to perform outgoing phone calls without user confirmation. Rule (7) prevents applications without SEND_SMS permission to send text messages without user confirmation.

Rules (5)-(6) are data dependent. For rule (5), ICC calls will be denied if Intent messages include, e.g., ".zip", ".rar" or ".apk" sub-strings. For rule (6), ICC calls will be allowed if *Intent* messages include a specific parameter that enforces user confirmation for a voice call to proceed (particularly, android.intent.action.DIAL string). Note, in this case user confirmation is requested not by *XManDroid*, but by security mechanisms of Android. Unfortunately, Android does not provide a possibility to request user confirmation for outgoing text messages. Thus, for rule (7), *XManDroid* requests such a confirmation itself. It is reasonable to assume that an ordinary user is able to take a correct decision in this case, as the user is aware of messages he is going to send.

4.4 System Representation

In this section we describe our representation of applications and ICCs, and discuss particularities of the representation of the Android system components.

4.4.1 Third Party Applications

We use a graph representation of the system, where vertices in the graph represent application sandboxes for 3^{rd} party applications and undirected unweighted edges show granted ICCs (cf. Figure 4). The graph representation allows us to apply algorithms from graph theory in order to analyze transitive communication links among applications and to define powerful policies based on properties of applications and Intents (cf. Section 4.3).

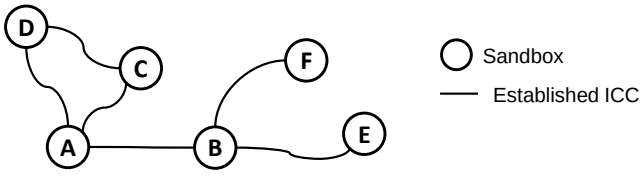

Figure 4: System representation of 3^{rd} party apps

Generally, a graph based representation allows different levels of system abstraction, e.g., vertices can be seen as (i) application sandboxes (level of user identifiers (UIDs)), (ii) applications (level of application packages), or (iii) application components (level of application interfaces). As our framework has no means to track data flows within a single application (e.g., tainting data [11]), and since multiple applications, residing in a single sandbox (by sharing the same UID), can establish channels which are invisible to *XManDroid* (e.g., via shared memory), we decided to represent 3^{rd} party applications on the level of sandboxes.

our view, the system policy should be written by Android developers and included into the default platform configuration. Further it can be updated by security patches when necessary.

Undirected representation of graph edges simplifies the system, however, at the cost of coarse granularity. Currently, we do not consider ICC direction, because many directed ICC calls result in establishing bidirectional communication links. However, directed representation would increase precision of analysis[9]. We are planning to consider directed edges in our future work.

Considering weighted edges would also improve the system, as more advanced graph algorithms can be applied that take into account the trust or risk level one assigns to certain ICCs. For instance, one could assign more trust to applications properly secured with Saint [35] policies. We leave this issue open for the future work.

4.4.2 Android System Representation

Android provides a number of system applications (e.g., phone or browser), system services (e.g., power manager or audio manager), and system content providers (e.g., contacts, system settings, or SMS database), which provide functionality to the user and other applications. By design these system applications provide overt channels between applications, but can also be misused as covert channels [37]. For instance, applications can insert data into and read data from system content providers such as the contacts database (Figure 5(a)), use broadcasts as covert channel or to directly transmit data as Intent (Figure 5(b)), or perform synchronized write-read operations on the settings of a system service such as the audio manager to establish a covert channel (Figure 5(c)). Some of these ICC based covert channels have been demonstrated in [37]. Thus, in order to enforce policies on both, overt and covert channels through those system applications, one has to assume that system applications provide transitive closure to other nodes connected to a system component node in the graph, e.g., application sandboxes A to F in Figure 5, for which the explicit channel between A and D is not visible, but A might communicate with all nodes that are connected to the Android system.

All system services and service providers share a common UID. If represented this way in our graph, those providers and services would form a monolithic core vertex to which all installed applications are connected and thus cause transitive closure for the policy enforcement. Thus, UID level of representation is too coarse grained for representation of system applications. However, the design of the system core applications (particularly, content and service providers) does not follow design principles established for third party applications and cannot be split up into component level.

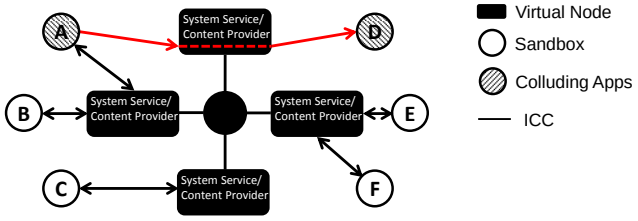

Figure 6: *Virtual Nodes* for System Services and System Content Providers

To resolve this issue, *XManDroid* features two important design extensions: (i) extraction of the system content providers and services from the monolithic core in the system graph as *virtual nodes* (cf. Figure 6) and (ii) fine-grained policy-based filtering of data in those providers and services (cf. Section 5). Virtual nodes are denoted as such, because they have a UID that is not actually present in the system, but solely defined by *XManDroid* as an internal identifier for the particular provider and service, used during policy enforcement. Moreover, our framework provides insight into the data flows inside those content providers/services. Thus, we are able to enforce policy based filtering of the read/write operations on data and values of these components, which constitute overt and potential covert channels. *XManDroid* filters the data instead of denying access to the content provider/service, because such a denial provides information to applications and thus can be used as a covert channel.

With this improved Android system representation, we mitigate transitive closure and optimize policy enforcement on overt/covert channels through Android's system components.

4.5 Enhanced Decision Storage

In the following, we describe an enhanced decision storage for *XManDroid* decisions which leverages the default Android permission system. A positive decision to allow a particular ICC call can be interpreted as a permission granted to a caller to access a callee. We call such a permission "auto permission" (AP). APs can be stored in the standard Android permission database and checked by the Android ReferenceMonitor. Thus, once an AP is granted, no invocation of *XManDroid* is required anymore to proceed the ICC call.

APs share many characteristics of default Android permission labels, but differ in the following points: (i) AP labels are *automatically* assigned by the PackageManager to each application component at install time (in contrast, standard Android permissions are assigned by developers); (ii) APs are granted at *runtime* by the DecisionMaker rather than by the user; (iii) AP labels are locally generated and unique for each platform instead of being globally well-known like INTERNET or ACCESS_FINE_LOCATION; (iv) APs are stateful, they reflect a policy decision, i.e., allowed or denied.

Note that our current prototype does not include auto permissions. We are currently working on its implementation.

5. IMPLEMENTATION

In this section we present the implementation of our architecture and its main components, as presented in Section 4.1. In particular, we explain how we implemented the system graph (Section 5.1), how we store the policy decisions (Section 5.2), and our implementation of the system policies (Sections 5.3). We elaborate in more detail on the policy checking mechanism (Section 5.4) and how we realized the "virtual nodes" and their particularities for policy enforcement (Section 5.5). We will use the scenario depicted in Figure 7 as running example in order to explain our implementation. Our implementation is based on the Android 2.2.1 sources (denoted *Froyo*)[10].

5.1 System Graph Implementation

We implemented the system graph by means of the open-source *JGraphT*[11] library, version 0.7.3. Each vertex repre-

[9]As our test results show, it is particularly necessary to distinguish read/write access to Content and Service providers (cf. Section 6)

[10]`http://source.android.com`

[11]`http://www.jgrapht.org/`

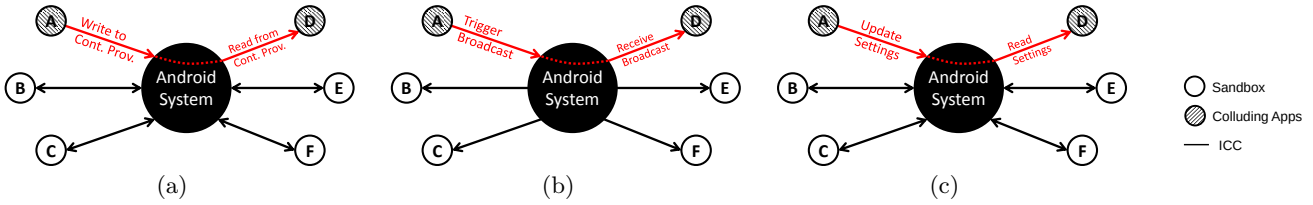

Figure 5: Communication channels via the Android system applications: (a) Overt channel through system content provider; (b) (Covert) channel via broadcasts; (c) Covert channel via system services

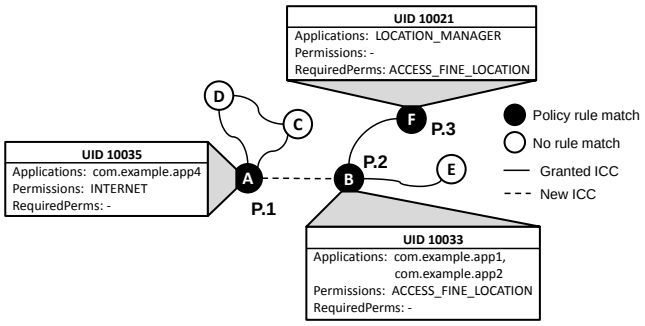

Figure 7: Example of a system graph

sents a UID in the system and contains the information for this UID which is retrievable from the Android PackageManager service, e.g., the package name or granted permissions. In case of shared UIDs, the information from all applications with the shared UID are merged in the corresponding vertex. Moreover, each vertex is tagged as trusted system component of Android or as untrusted, third party application. SystemView is persistent across reboots, by saving it upon shutdown and loading it upon system boot. If no saved SystemView can be found or during the very first boot of a system with *XManDroid*, an initial graph has to be build. We use the PackageManager to discover the system components and installed applications during this building process.

We further modified the PackageManager such that it uses an interface of the SystemView component to update the system graph upon installation or de-installation of an application package. The installation adds a new vertex for the new application UID into the graph. In case of shared UIDs, the information is again merged. Application un-installation causes the removal of the corresponding vertex. Un-installation in case of shared UID is slightly more complex, since only the information added by the de-installed application has to be removed. We implemented this by removing the corresponding vertex and reconstructing it afterwards with the updated application information.

A simplified example for a such graph representation is illustrated in Figure 7. The example involves six vertices, A to F, each representing a different UID in the system. Vertex F has a special role, since it represents a system service ($LOCATION\_MANAGER$) of the core application Android, which provides the APIs for location services, such as retrieving the fine grained GPS coordinates of the platform. Thus this node is essentially virtual. The applications are interconnected by a number of already established ICCs. Further, black vertices represent applications that form (in this constellation) a violation of the system policy (cf. Section 4.3), while white vertices represent non-violating applications. We will refer to this example in the subsequent Sections 5.3 and 5.4.

Figure 7 also shows the information stored in the vertices of our example. We only use a simplified example, e.g., the location system service holds and requires a higher number of permissions on a real-world system. Vertices A and F represent only one single application package, while vertex B represents two. B holds the permission to access the location service API in vertex F to retrieve the fine grained location data, while the application in vertex A has been granted the Internet permission.

5.2 Storing Decisions

The decisions made by DecisionMaker are stored in the component Decisions, which is implemented as a Java HashMap. It stores a Boolean for the decision result of each checked ICC, i.e. granted or denied. The index of the HashMap is the unordered tuple $\{uid\_caller, uid\_callee\}$ for each inserted decision. Based on the Decisions component, the DecisionMaker is able to efficiently look up previously made decisions and grant/deny an ICC that has occurred before. DecisionMaker is persistent across system reboots.

Moreover, granted and thus established ICC are reflected as edges in the system graph. Possible Intent information for the ICC are stored at the corresponding edge and updated in case of new Intents for this ICC pair.

The insertion or deletion of a vertex may affect previously made policy decisions. For instance, a removed vertex implicitly deletes paths in the system graph that contain the deleted vertex. Thus, previously denied ICCs might now be granted under the same system policy that was used during the first decision. Hence, the PackageManager triggers upon (un-)installation of a package a reset of the SystemView in order to enforce a re-evaluation of the previously made policy decisions. Reset in our implementation means the removal of the edges in the graph and the emptying of Decisions. For instance, if vertex B in Figure 7 were un-installed, the policy violating path from A to F would be resolved.

An alternative approach, which we call *AutoPermissions*, is explained in 4.5. To implement such a system, certain implementation challenges must be solved: (i) the ReferenceMonitor must be adapted to allow two permission labels per component, namely the default label defined by the developer plus the AP assigned by the PackageManager; (ii) to distinguish the case that an AP has been denied and that this ICC has never occurred before, the AP has to be "stateful" in a sense that the AP label indicates the policy decision, i.e. allowed or denied; (iii) AP labels have to be clearly distinguishable from all newly defined permissions by applications.

```
<?xml version="1.0" encoding="utf-8"?>
 <SystemPolicy>
  <PolicyRule name="Policy Rule 2" group="0" proceed="0">
    <Vertex>
     <Property type="RequestedPermissions" value="android.
          permission.ACCESS_(FINE|COARSE)_LOCATION"/>
     <Property type="RequestedPermissions" value="android.
          permission.ACCESS_INTERNET" negated="true"/>
    </Vertex>
    <Vertex>
     <Property type="RequestedPermissions" value="android.
          permission.ACCESS_INTERNET"/>
     <Property type="RequestedPermissions" value="android.
          permission.ACCESS_(FINE|COARSE)_LOCATION"
          negated="true"/>
    </Vertex>
    <Vertex>
     <Property type="PackageName" value="android"/>
    </Vertex>
  </PolicyRule>
 </SystemPolicy>
```

Figure 8: Example system policy with policy rule 2 (cf. Table 1)

5.3 Policy Implementation

The system policy is implemented in XML (the corresponding XML schema is presented in Appendix A), which is loaded upon system boot or after an update of the policy, respectively. SystemPolicy in our implementation is installed or updated via PolicyInstaller component, which is implemented as a service running in the middleware as part of the ActivityManager and provides an authenticated channel in order to externally update the policy.

Appendix A presents the XML schema of our system policy. The system policy consists of a number of policy rules. Each rule describes a path in the system graph by defining the vertices and edges that constitute this path. The vertex or edge definition is based on the information the PackageManager can provide for an UID or the information an Intent can contain, respectively. To ease the definition of more complex and powerful rules, the definitions can be described as regular expressions, also allowing to define negated properties, i.e., properties that a matching vertex must not have. Further, each rule is marked with an attribute that determines how to proceed in the policy checking (cf. 5.4). Based on optional vertices, policy rules are flexible regarding their path length. A rule is matched if all non-optional and any number of the optional vertices and all edge definitions of the rule could be assigned to distinct vertices along a path in the system graph.

To allow the definition of (fine-grained) exceptions from generic policy rules, the rules are checked in a top-down manner. If policy is matched and it would grant the ICC, all policy rules with the identical group ID as the matched rule are skipped. This enables the definition of more generic rules at the bottom inside a group of policy rules and more specific rules at the top, respectively.

Example.

Figure 8 shows the XML implementation of the policy rule 2 in Table 1, which prevents the leakage of the location data via the Internet by means of transition of permissions over two nodes, i.e., two colluding applications. The rule consists of three vertex descriptions. A vertex in the system graph that matches the top-most of these three description must contain the permission to access the location service, but must not contain the Internet permission. Vertex A in Figure 7, for instance, matches this description. Similarly, vertex B matches the second description. The third description applies to vertex F of the Figure 8, since it contains the package name "android". Consequently, the path A-B-F matches the policy rule 2 and the value zero of the *proceed* attribute of the rule indicates that the ICC between A and B has to be denied.

5.4 Policy Checking

5.4.1 Direct ICC

In the Android sources, the function *checkComponentPermission* of the ReferenceMonitor in the ActivityManager is responsible for verifying that the ICC caller has been granted the permission required by the callee (component) to establish the ICC (Cf. Figure 9(a)). If not, the ICC is denied. In our implementation we modified the ReferenceMonitor to redirect the control flow from this function to our DecisionMaker whenever an ICC occurs that would start an *Activity*, bind to a *Service*, connect to a *Content Provider*, or send a broadcast message to *Broadcast Receiver* (cf. Figure 9(b)). In particular, we wrapped the *checkComponentPermission* function with a new function, which first calls the default *checkComponentPermission* function and in case that it would allow the ICC, we verify in the DecisionMaker that the ICC is not violating the policy. In case the DecisionMaker detects a policy match, it overrules the ReferenceMonitor decision with the decision defined in the matched policy rule.

To enable the DecisionMaker to make the policy check, the wrapper function provides the UIDs of the caller and callee of the respective ICC as well as the Intent initiating the ICC to the DecisionMaker.

5.4.2 Broadcast filtering

In case of broadcast Intents one has to consider a possible one-to-many relationship between the sender and the receivers. To address this problem, we followed an implementation approach as presented in [35, Section 6]. Each sender-receiver pair is checked for a policy violation and the broadcast receiver list is adapted accordingly before sending the broadcast message.

5.4.3 Pending Intents

Pending intents allow an application A to delegate an Intent to another application B, where B inputs the Intent data. In this case the pending intent requires an ICC from A to B for the delegation and from A to the designated receiver. This can be easily misused for privilege escalation attacks. However, our framework can prevent this, since it monitors all ICCs and can apply the system policy on both ICCs accordingly.

5.4.4 Policy Check Algorithm

The policy check algorithm implemented in DecisionMaker determines if the edge for the new ICC would complete a path in SystemView, which matches a policy rule.

The algorithm is presented in Algorithm 1. It uses backtracking[12] to explore in a depth-first search fashion all paths,

[12]Backtracking algorithms incrementally build candidates to the solution for the targeted problem and abandon partial candidates, when they detect that this candidate does lead to a valid solution.

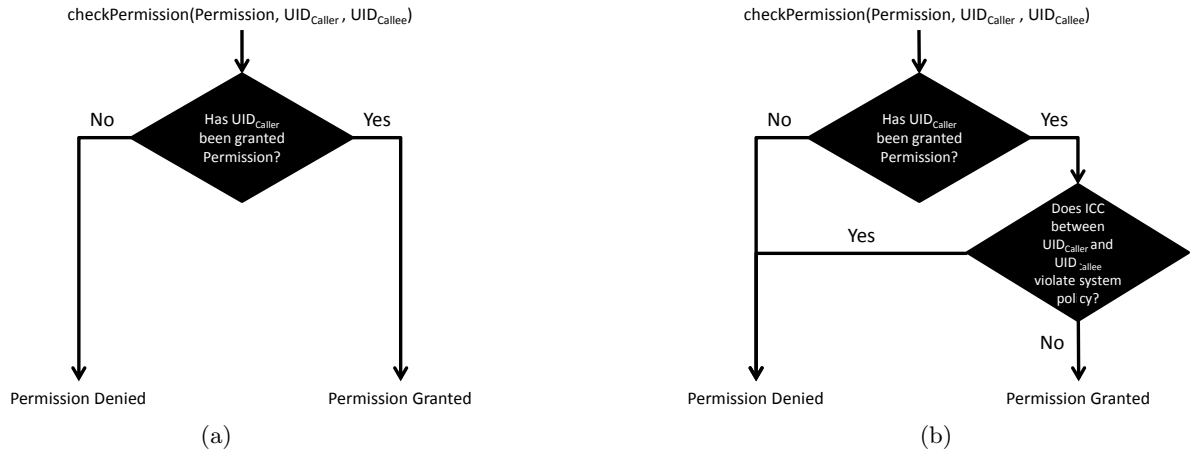

Figure 9: *checkPermission* process in (a) default Android and (b) with *XManDroid* extension

which only consists of vertices that match a different vertex description in the same policy rule until it finds a path that matches exactly the policy rule. The backtracking and depth-first search functionalities are implemented in the function *FindPath*, presented in Algorithm 2. For brevity, we omit the explicit checking for Intent definitions along the traversed edges in the graph.

Algorithm 1: Graph based algorithm for finding policy rule matching paths

input : Vertex *v1*, Vertex *v2*, Set *PolicyRule*
output: Boolean *policyRuleMatch*

```
1  if AreSystemPackages (v1, v2) == true then return false;
2  pat ← PatternMatch (v1, PolicyRule);
3  if (pat == null) then return false;
4  PolicyRule ← PolicyRule \ pat;
5  pat ← PatternMatch (v2, PolicyRule);
6  if (pat == null) then return false;
7  PolicyRule ← PolicyRule \ pat;
8  if PolicyRule == ∅ or OnlyOptionalLeft (PolicyRule) ==
   true then return true;
9  else return FindPath (v1, v2, PolicyRule);
```

Algorithm 2: `FindPath` function for finding paths that match the policy rule P

input : Vertex *root*, Vertex *otherRoot*, Set *PolicyRule*
output: Boolean *pathFound*

```
1  if IsSystemPackage (root) then return false;
2  Neighbors{} ← GetNeighborSet (root);
3  foreach n in Neighbors do
4  |  pat ← PatternMatch (n, PolicyRule);
5  |  if pat ≠ null then
6  |  |  PolicyRule ← PolicyRule \ pat;
7  |  |  if PolicyRule == ∅ or OnlyOptionalLeft
   |  |  (PolicyRule) == true then return true;
8  |  |  if FindPath (n, otherRoot, PolicyRule) == true
   |  |  then return true;
9  |  |  PolicyRule ← PolicyRule ∪ pat;
10 |  end
11 end
12 if otherRoot ≠ null then return FindPath (otherRoot, null,
   PolicyRule);
13 else return false
```

Example.

We explain the Algorithms 1 and 2 based on the example Figure 7. A number of ICCs have already been established, however the ICC between two application components in vertex A and in vertex B has yet to be established. The Android ReferenceMonitor allowed this ICC and our DecisionMaker evaluates if the ICC would lead to a policy violation by checking if the policy rule in 8 matches a path in the system graph. If a matching path can be found, it returns *true*, otherwise *false*.

The algorithm takes both vertices between which the ICC shall be established as inputs, here A and B, as well as the policy rule to check for, consisting of a set of vertex descriptions. Here we denote the three vertex description from Figure 8 top-down as $\{P.1, P.2, P.3\}$, where $P.1$ is the vertex with Internet access but no location data access, $P.2$ the vertex with location data access but no Internet access, and $P.3$ is the location manager system service.

The algorithm first checks if both vertices are trusted system packages (line 1) and if so returns *false*. Since we trust system packages, an ICC in between them can by definition not be malicious. In our example, both A and B are not trusted. Afterwards, the algorithm checks, if the both vertices match a vertex description in the policy rule (lines 2-7), since both must be on the matching path, and if one does not, *false* is returned. To keep track of which vertex descriptions are missing to complete the path, already matched descriptions are removed from the policy rule set[13] (lines 4 and 7). A matches $P.1$, while B matches $P.2$. If the set would consist of only two descriptions or only optional vertices remain in the set, the algorithm would return *true* (line 8), since the new ICC constitutes the complete path. If the set contains more than two non-optional descriptions, the algorithm tries to complete the path among the neighbors of *v1* and *v2*, i.e., A and B, by means of the *FindPath* function.

[13]Of course the algorithm operates on a copy of the rule, such the policy matching does not modify the system policy

In our example, the algorithm calls *FindPath(A, B, {P.3})*. The neighbors of A are explored first. However, if A would be a trusted system package, its neighbors do not have to be explored (line 1), since trusted system packages can only be end-vertices of a policy violating path (unless checked transitively, cf. Section 5.5). Each neighbor of A is checked for a match with a remaining vertex description of the policy rule (lines 3-4) and if a neighbor matches, the matched description is removed from the set (line 6). In our example, no neighbor matches and the function tries to complete the path among the neighbors of B (line 12) by calling *FindPath(B, null, {P.3})*. The *null* parameter indicates that the search is already among the neighbors of the vertex *otherRoot* and hence has to be aborted in line 12. In our example, the neighbor F of B matches *P.3* and the path is completed, thus *true* is returned by the algorithm. Since the policy rule in Figure 8 is marked with *proceed="0"*, the ICC between A and B is denied by the DecisionMaker. However, if the policy rule would be more complex and consist of more than three vertices, *FindPath* would try to complete the path by means of a depth-first search (line 8) and backtracking (line 9).

Complexity of the algorithm.

If we assume a full meshed graph with $|V|$ vertices and a SystemPolicy with $|S|$ policies, where $|P|$ is the cardinality of each policy, the complexity of the algorithm is $O(|P|^2 * |S| * |V|)$.

The functions *IsSystemComponent* and *AreSystemComponent* run in $O(1)$. *PatternMatch* requires $O(|P|)$. Removal of an element from set S runs in $O(|S|)$. Set union runs in $O(1)$. Lines 1-8 of Algorithm 1 run, thus, in $O(4*|P|)$. The function *GetNeighborSet* runs in $O(1)$. The *for-each* loop in *FindPath* executes $O(|V-1|)$ times. The loop body requires (excluding the recursion) $O(2*|P|+1)$. We require $O(|P|)$ recursive calls to *FindPath*. Hence, lines 9-11 of Algorithm 1 have a complexity of $O(|P| * (1 + |V-1| * (2 * |P| + 1))) = O(|P|^2 * |V|)$. We have to execute Algorithm 1 once for each policy in SystemPolicy, hence $O(|SP|)$.

5.5 Virtual Nodes and Data Filtering in System Components

In this section we briefly explain how we implemented the virtual nodes introduced in Section 4.4.2 and how we filter retrievable data from this nodes in order to mitigate overt and covert channels.

5.5.1 System Content Providers

To extract the system content providers from the monolithic Android (cf. 4.4.2) as virtual nodes we extended the ActivityManager of Android. During boot-up and normal operation of the system, the ActivityManager detects and registers all installed services and content providers in the system. We extended this process such that the ActivityManager inserts a virtual node for each system content provider it finds via an interfaces of SystemView.

Content providers are essentially databases used to store and exchange information between applications. The inner workings of content providers are abstracted through a well defined, unified interface. For example, all system content providers, such as the contacts database or system settings, are SQL database backed and provide an interface to insert or update content (write) and to (bulk-)query the data (read).

Figure 10(a) illustrates how we modified these interfaces in order to filter data upon reading in order to prevent potential policy violating, overt channels. Each data row in the database is tagged with the UIDs of the writers. Upon writing data, these tags are updated. We keep all writer UIDs in the tag, since simply storing the last writer is not sufficient to prevent colluding applications from using this channel, because writing does not necessarily overwrite existing data but can also append new data to existing ones. Upon reading, our extension to the reading interface verifies for each read row if the corresponding reader-writer pairs constitute a policy violation that would deny the ICC. If so, the corresponding row is filtered from the response to the reader before return.

To minimize the performance overhead, especially during bulk queries, DecisionMaker uses again the cache in Decisions. Moreover, the read interface extensions create a local cache for each read query in which the UIDs of writers are stored that have to be removed from the response, thus minimizing the necessary remote calls to DecisionMaker and speeding up the filtering. After each response the corresponding cache is deleted.

Since granted reading of a value forms a channel between the reader and the writer, this has to be reflected in SystemView by adding an edge between the reader's and the non-filtered writers' vertices.

5.5.2 System Services

System services implement certain phone functionality, e.g., the audio manager to control the volume or the location manager to retrieve the (fine-grained) location of the phone. Those services provide, similar to the content provider, an interfaced to read and to write values, for instance the concrete level of the volume.

Although Android's Service Manager maintains a list of the registered services, this list does not provide information (without intruding modifications) if a service is a system or third party service. Thus, we currently build our graph with pre-installed virtual nodes for system services.

Figure 10(b) illustrates how we enforce policy checks on the interfaces of system services. Similarly to system content providers we tag each value of the service with the UID of the writer. However, in case of services only the last writer, since no appending to values is possible but only overwriting of existing values. Since the services are not backed by a database, we maintain a HashMap from value identifier to writer UID in order to tag the values. Upon reading of values, we use a similar interface as for content providers to perform a policy check if the reading would establish a policy violating channel. If so, the response to the read is simply *null*. We are aware of the fact, that this forms essentially a covert-channel with one bit bandwidth, however, returning pseudo or default values on which falsely denied legitimate application would continue their operation might entail harm to the rest of the system, hardware, or even the user (e.g., falsely assumed screen brightness or volume level).

6. EVALUATION

In this section we evaluate the performance as well as the effectiveness of *XManDroid* by applying a malware test suite that we designed constituting the most recent privilege escalation attacks [13, 8, 26, 37]. Further, we select a representative set of 50 benign applications from the Android market and evaluate how *XManDroid* affects their performance and functionality.

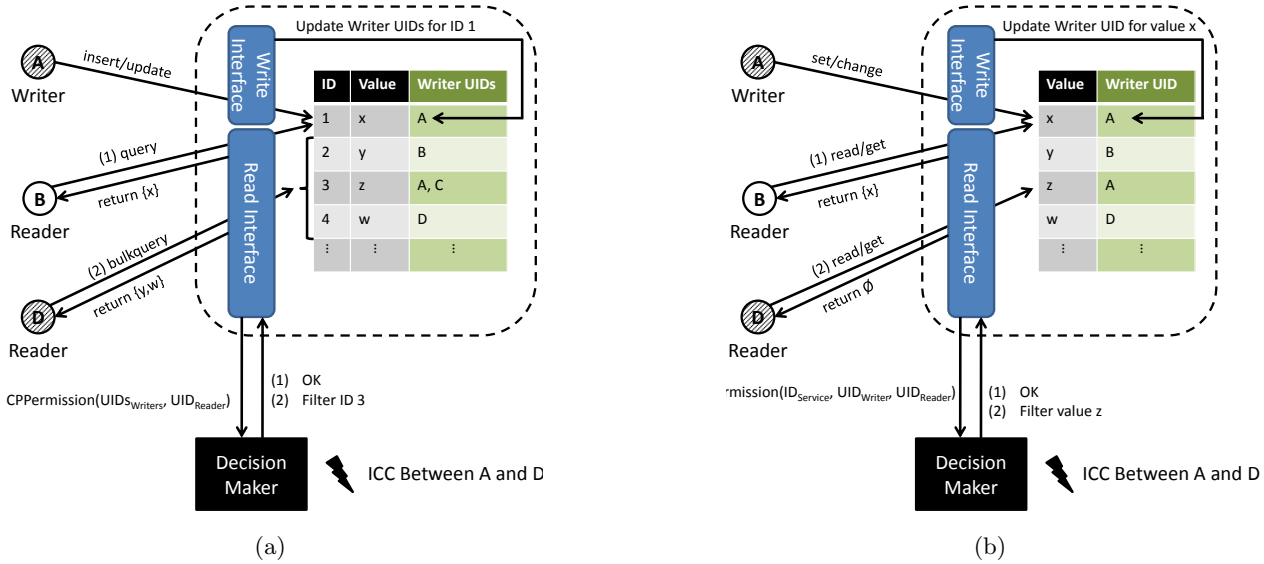

Figure 10: Filtering of returned data to a reader at (a) System Content Providers and (b) System Services

6.1 Malware Test Suite

Our malware test suite exploits transitive permission usage in order to perform attacks against user privacy or to gain unauthorized access to protected system interfaces. The collection includes 7 attack examples described in Table 2. Attack scenarios 2-4 are proof-of-concept examples of malware, while scenarios 1 and 5-7 emulate the attacks in [37, 26, 13, 8].

	1^{st} App	2^{nd} App
1	Malicious voice recorder RECORD_AUDIO and PHONE_STATE or PROCESS_OUTGOING_CALLS	Malicious wallpaper [37] INTERNET
2	Malicious step counter ACCESS_FINE_LOCATION	Malicious wallpaper INTERNET
3	Malicious contacts manager READ_CONTACTS	Malicious wallpaper INTERNET
4	Malicious SMS widget READ_SMS	Malicious wallpaper INTERNET
5	Malicious app no INTERNET	Vulnerable browser [26] INTERNET
6	Malicious app no CALL_PHONE	Vulnerable dialer [13] CALL_PHONE
7	Malicious app no SEND_SMS	Vulnerable SMS widget [8] SEND_SMS

Table 2: Malware test suite

Attacks 1-4 involve two colluding malicious applications, where one application has Internet access, while another one can obtain sensitive user data, such as user location, recorded audio, contact and SMS database. In the scenario 1, the malicious voice recorder additionally requires the PHONE_STATE or PROCCESS_OUTGOING_CALLS permission in order to be notified when the incoming or outgoing call begins. Applications collude to deliver private user data to the remote adversary. In scenarios 2-4, applications establish the standard ICC communication link, while in the scenario 1 they communicate by means of a covert channel (similarly to Soundcomber eavesdopping malware [37]). We validated three types of covert channels: (i) synchronized adjustment and reading of the voice volume; (ii) change of the screen state and (iii) change of the vibration settings.

In the attack scenarios 5-7 a malicious application misuses a vulnerable application with access to the Internet, voice call or SMS services in order to obtain transitive unauthorized access to these system interfaces. Scenario 5 emulates attacks shown in [26, 10] and implements unauthorized download of malicious files by exploiting an unprotected interface of the Android web-browser, while scenario 6 reproduces unauthorized phone call by misusing a vulnerability of the Android Phone application reported in [13]. Note that since this vulnerability of Android Phone application has been already fixed, we developed an own vulnerable dialer application for testing purposes. In scenario 7 the malicious application sends unauthorized text messages, similar to the attack reported in [8].

6.2 Effectiveness

We tested *XManDroid* to evaluate its effectiveness in detecting the malware presented in 6.1. All tests were performed on the Nexus One developer phone running Android 2.2.1 with our patches. The system policy of *XManDroid* included the fine-grained policy rules (1)-(7) shown in Table 1.

We installed the malicious applications from our test suite and performed the corresponding attacks. All attacks were successfully detected and prevented by *XManDroid*.

6.3 Performance

We used a clean Android system using automated testing scripts to test performance of *XManDroid* with 50 applications from Android market. The applications were installed, used by Monkey tool[14] and deinstalled.

Table 3 lists our performance results. In total 11970 ICC calls occurred during the testing and we observed 11592 cache hits (378 cache misses). The measured average runtime for the Android ReferenceMonitor was 0.184ms, to which *XManDroid* added in average 13.126ms in case of an uncached

[14]UI/Application Exerciser Monkey:
`http://developer.android.com/guide/developing/tools/monkey.html`

Type	Calls	Average (ms)	Min (ms)	Max (ms)	Std. dev. (ms)
Original Reference Monitor runtime					
system	11920	0.184	0.031	208.862	2.490
XManDroid DecisionMaker overhead					
uncached	378	13.126	0.305	779.510	43.783
cached	11592	0.105	0.031	23.712	0.636

Table 3: ICC timing results

ICC type	Policy checks	Denied ICC	Percentage
ICCs between apps	1746	33	1.9%
ICCs to System Service Providers	377	17	4.5%
ICC to System Content Providers	1701	48	2.8%
Total	3824	98	3.0%

Table 4: Aggregated number of occurred and denied ICCs during user tests

ICC call or 0.105ms in case of a cached call. The low standard deviation for our test cases indicates a consistent performance of our implementation. However, due to system load and hardware limitations such as I/O bandwidth, rare but strong fluctuations occurred, e.g., 779.510ms for uncached ICC calls.

The maximum memory usage of *XManDroid* during these tests was about 4MB, thus in an acceptable range.

Our tests show that the performance overhead imposed by our architecture is below human perception and the user will not notice any performance delays.

6.4 False Positives and Usability Test

To evaluate how *XManDroid* affects 3^{rd} party applications, we selected the representative set of 50 free well-known applications from the Android market. In particular, we included applications which properties match vertex descriptions (see details in Section 5.3) for tested policy rules, thus increasing probability for policy match.

To perform our usability test, we evaluate how many policy matches *XManDroid* produce in a user-study with a group of 20 students. We opted for manual testing, as automated testing of mobile phone applications has been shown to exhibit a very low execution path coverage, approximately 40% in average and only 1% in worst case [19]. Our test results comply to this observation, as during our performance test no policy matches occurred. Note that the performance overhead of our policy check is worst when no match is detected and, thus, the results of our performance test form an upper bound.

The students' task was to install and thoroughly use the 50 applications in our test set in an arbitrary order, with interleaving installation, uninstallation, and usage of the apps. The evaluation was conducted with an implementation, that did not yet provide the data filtering feature on system content providers and services, since its full implementation is ongoing work. Table 4 shows the number of policy checks that occurred during the user tests and how many ICCs were denied. The relative number of denied ICCs is very small, especially compared to a static system like Kirin [13]. If our policies would be enforced with Kirin, each of the 50 test applications would in average conflict with 27 other applications from the set. Several applications would even conflict with more than 40 other apps. However, the testers reported that the impact on the user experience with

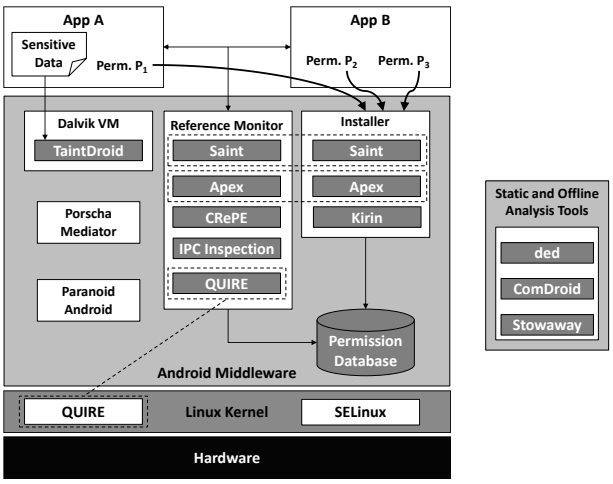

Figure 11: Main security extensions for Android

XManDroid was still noticeable and the the system has to be further optimized.

An evaluation of the denied ICCs revealed that all of them were false positives. Particular sources for false positives were (i) a customized application launcher app, causing 29 of the 33 false positives for direct ICC due to the high number of permissions this app holds (particularly, the CALL_PHONE permission caused the majority of the policy matches); (ii) the power system service provider, causing all false positives for the system service providers; and (ii) the system settings content provider, causing 45 of the 48 false positives. An examination of the ICCs to the settings content provider and power service showed, that more than 90% of the ICCs were read operations and roughly 10% write operations by trusted Android components. The false positives for the custom launcher application occurred when starting an application with the launcher, which requires direct ICC between those two applications.

While the false positives for the custom launcher can be prevented with tuning of the system policies, the false positives for the settings content provider and power service would have been prevented with enabled fine-grained filtering of data and values. Although filtering does not prevent false positives per se, it significantly reduces the rate of false positives induced by system providers as shown in our evaluation.

7. RELATED WORK

Figure 11 gives an high-level overview of the main relevant security extensions for Android. It depicts the main components of the middleware layer consisting of the application installer, the reference monitor, the permission database, and the Dalvik virtual machine. Each solution requires extension to one or more component. Figure 11 also shows several offline static analysis tools that aim to detect vulnerable or malicious application components and interfaces. In the following we will discuss the strengths and shortcomings of these extensions with respect to privilege escalation attacks.

Kirin is an extension to Android's application installer [13, 14]. Its model requires checking permissions at install-time. It denies the installation of applications that may encompass a set of permissions that violates a given system policy. The

Kirin framework described in [13] identifies security-critical communication links which may be used for a collusion attack. This is achieved by analyzing which interfaces the new application is authorized to contact. However, Kirin does not accurately address the fact that Android applications provide unprotected interfaces, by which applications can establish communication links without requiring any permission.[15] Due to its static nature, Kirin has to consider all potential communication links over the unprotected interfaces to address this problem. This will stop any application from being installed, since applications can potentially establish many communication links whereas most of them can be exploited for a privilege escalation attack. Moreover, Kirin focuses only on directly reachable interfaces [13, Section 5.2], whereas transitive permission attacks may involve multiple applications. In contrast to Kirin, *XManDroid* (1) focuses on real communication links rather than on potential ones, (2) decides at runtime if the link (including communications over unprotected interfaces) to be established violates the system policy, (3) detects transitive permission usage over any number of hops, and (4) handles exceptional cases (e.g., pending intents and dynamic broadcast receivers).

Saint introduces a fine-grained access control model [35] that allows applications to protect themselves from being misused. It requires application developers to add security features to their applications and extends the basic Android permission system by allowing the system to enforce security decisions based on signatures, configurations and contexts (e.g., phone state), whereas security decisions are enforced both at install-time and at runtime. This gives application developers the possibility to protect their applications, in particular the application's interfaces, from being misused by unauthorized or malicious applications. In order to prevent privilege escalation attacks, application developers have to assign appropriate security policies on each interface. For instance, the Browser attack [26] could be prevented by assigning Saint policies to the interfaces of the Android's web browser. However, since application developers have to define these policies themselves, they might fail to consider all security threats. Thus, developer-defined permission systems are more likely to be error-prone than system-centric approaches as we use in *XManDroid*. Finally, Saint does not address malicious developers, that will not deploy Saint policies for obvious reasons thereby allowing two malicious (colluding) applications to communicate with each other without being controlled by Saint policies.

Porscha provides policy-oriented secure content handling in Android [34]. The goal of Porscha is to bind any security-sensitive data or content to a certain phone and to a specific set of applications. Content sources such as devices transmitting SMS, MMS, or e-mails can attach a DRM (Digital Rights Management) policy to the message. Even though Porscha introduces a much more fine-grained permission model, it cannot prevent leakage of data not tagged with a security policy. Further, the main purpose of Porscha is to control data flows, whereas privilege escalation attacks based on control flows are not addressed by Porscha.

Recently, Enck et al. presented **TaintDroid**, a sophisticated framework which detects unauthorized leakage of sensitive data [11]. TaintDroid exploits dynamic taint analysis in order to label privately declared data with a taint mark, audit on track tainted data as it propagates through the system, and alerts the user if tainted data aims to leave the system at a taint sink (e.g., network interface). TaintDroid is able to detect data leakage attacks potentially initiated through a privilege escalation attack. However, similar to Porscha, TaintDroid mainly addresses data flows, whereas privilege escalation attacks also involve control flows. Enck et al. [11] mention that tracking the control flow with TaintDroid will likely result in much higher performance penalties. Moreover, in contrast to *XManDroid*, TaintDroid cannot detect attacks that exploit covert channels to leak sensitive information [37].

PiOS is a static analysis system to detect leakage of private data by applications for Apple's iOS operating system [10]. In contrast to TaintDroid, it does not perform a dynamic check at runtime, but analyzes the binary code of iOS applications statically. PiOS constructs a call graph from the binary and looks at all paths from a "privacy source" to a "sink" and checks whether private data (such as address book or GPS location) are transmitted to a sink without notifying the user. Hence, PiOS can only detect privacy leaks within a single application, but not if two or more applications are transitively chained together as in our attack scenario.

Apex is an extension framework of Android's permission model [30]. It allows the user to selectively grant and deny permissions requested by applications at install time. Moreover, the user has the possibility to define runtime constraints, e.g., limit the number of text messages to be send per day. Even though Apex makes Android much more flexible by allowing users to constraint certain functionalities, it unfortunately relies on the user to take security decisions. Moreover, privilege escalation attacks where permissions are split over multiple applications are not addressed by Apex.

CRePE (Context-Related Policy Enforcement for Android) enables the enforcement of context-related policies [7]. Hence, users may define policies which enable/disable certain functionalities (e.g., read SMS, bluetooth discovery, GPS) depending on the context of the phone (e.g., location, temperature, noise, user, etc.). Moreover, contexts can be also defined by a trusted third party facilitating, for instance, employers to enforce a company-wide policy for all employees owning Android smartphones. However, CRePE does not address privilege escalation attacks in general, since it mainly focuses on the enabling/disabling of certain functionalities rather than on the transitive permission usage across different applications.

Paranoid Android (PA) detects viruses and runtime attacks exploiting memory errors such as buffer overflows by deploying a virus scanner and a dynamic taint analysis system [36]. Potentially PA could be extended to check control flows thereby being able to address privilege escalation attacks. However, PA requires the execution trace to be stored in secure storage (in order to prevent malicious modification) and impacts the performance of the device, since the execution trace has to be continuously transmitted to a remote analysis server that replays and analyses the execution trace.

QUIRE [9] is a recent Android security extension. It provides a lightweight provenance system that prevents the so-called confused deputy attack. This attack is a special type of privilege escalation attacks where a malicious application abuses the interfaces of a trusted application to perform an unauthorized operation. In order to determine the originator of a security-critical operation, QUIRE tracks

[15] For instance, enforcing downloads of malicious files can be mounted over the unprotected interface of the Android Browser [26] without requiring the INTERNET permission.

and records the ICC call chain, and denies the request if the originating application has not been assigned the corresponding permission right. Additionally, QUIRE extends the network module residing in the Android Linux kernel to perform its analysis also on remote procedure calls (RPCs). In summary, QUIRE addresses privilege escalation attacks that exploit vulnerable interfaces of trusted applications. This approach is complementary to ours, however, QUIRE does not address privilege escalation that are based on maliciously colluding applications. Since the ICC call chain is forwarded and propagated by the applications themselves, colluding applications may forge the ICC call chain to obscure the originating application, and hence, circumvent QUIRE's defense mechanism. Moreover, the current scheme of QUIRE does not allow the prevention of privilege escalation attacks that exploit covert channels in the Android Core, and finally, their approach is in contrast to *XManDroid* not system-centric.

IPC Inspection [17] is a very recent work that similarly to QUIRE addresses confused deputy attacks. Remarkably, the authors discovered several severe attacks against Android's system applications and demonstrated that a number of pre-installed applications are vulnerable to confused deputy attacks. The key insight of IPC Inspection is to reduce the permissions of an application when it receives a message from a less-privileged one. In contrast to *XManDroid*, IPC inspection does not require a policy framework, and hence, can prevent attacks immediately, while in our system we require the deployment of appropriate policies. On the other hand, IPC inspection is less general than *XManDroid*, and does not cover many types of attacks that *XManDroid* prevents. First, IPC Inspection does not address the case of a malicious developer, and hence, it cannot prevent collusion attacks, mainly because the individual application instances still reside in one sandbox. Second, it provides no means to prevent attacks launched via covert or overt channels (like Soundcomber). Third, IPC inspection considers only control-flows, but does not tackle attacks that are performed through data channels. Fourth, it performs permission reduction for permissions classified as dangerous, but neglects permissions classified as normal, and does therefore not prevent privilege escalation attacks that exploit these permissions. Fifth, IPC Inspection does not consider confused deputy attacks that might happen as a result of a IPC callback. Finally, it remains unclear how permission reduction is performed for permissions that are controlled by the underlying Linux discretionary access control (DAC) system rather than Android's reference monitor (e.g., Internet or Bluetooth). Unfortunately, the authors do not mention this issue. Apart aforesaid, IPC Inspection is likely to impose significant usability drawbacks. Although the authors present no performance measurements for IPC inspection, it will likely induce higher performance penalties than *XManDroid*, because it requires the creation and maintenance of multiple application instances with different sets of privileges. Second, unexpectedly revoking permissions at runtime very likely causes applications to crash, as their developers do not anticipate this and omit corresponding error handling code. Moreover, the framework is not compatible with legacy Android applications, because it requires many applications to be recompiled with a higher permission set.

ComDroid [6] is a recent static analysis tool that detects application communication vulnerabilities. It analyzes a single application and detects vulnerable application interfaces and security-critical intent/broadcast transmissions. For instance, it warns the application developer from sending privacy-sensitive data with a public broadcast, because this would allow a malicious registered broadcast receiver to eavesdrop the message. Although ComDroid is able to detect vulnerable application communications, it cannot detect privilege escalation attacks that are based on multiple colluding applications, since it focus on a single application. Moreover, static analysis tools (like ComDroid) are likely to be incomplete, because they cannot completely predict the actual application communication that will occur at runtime.

Similar static analysis tools were presented very recently: **Stowaway** is built on top of ComDroid and checks if an application follows the least privilege principle [16]. In particular, the authors applied Stowaway to 940 Android applications and showed that one third of them are overprivileged. A similar application study has been performed on 1,100 popular Android applications by Enck et al. [12]. To perform the application study, the authors developed the sophisticated **ded** decompiler that decompiles Dalvik executables to Java source code. After obtaining the source code of the application, they perform static analysis on the source code to detect vulnerable interfaces and malicious components. The study shows that many applications leak sensitive information like phone identifiers and location. On the other hand, there has been no evidence for malware or exploitable vulnerabilities. However, similar to ComDroid, these static analysis tools do not in general target privilege escalation attacks. Instead they detect vulnerable interfaces and malicious components within a single application.

SELinux can be incorporated into the Android Linux kernel [38] to mitigate kernel-level privilege escalation attacks. Since SELinux operates at the kernel level, it cannot prevent application-level privilege escalation attacks which we mainly consider in this paper. However, we believe that defense mechanisms addressing application-level privilege escalation attacks can be implemented on top of a SELinux secured Linux kernel.

In order to complete the overview on Android's security extensions, we finally present the defense mechanism Schlegel et al. [37] propose against their own developed voice recording Android Trojan **Soundcomber** [37]. In general, Soundcomber exploits covert channels of the Android OS enabling two colluding applications to communicate. The defense technique is specific to Soundcomber, and not general, as provided by *XManDroid*: It maintains a list of critical numbers and disables audio recording during a sensitive phone call.

To summarize, Android's security extensions such as Kirin, Saint, TaintDroid, and QUIRE currently do not provide general solutions against privilege escalation attacks, in particular, none of them can detect recent attacks such as Soundcomber that escalates privileges over covert channels.

8. CONCLUSION AND FUTURE WORK

In this paper we address the problem of application-level privilege escalation attacks on Android as shown recently by several severe attacks [13, 8, 26, 37]. Existing security extensions to Android such as Kirin [13, 14], Saint [35], and TaintDroid [11] cannot adequately address these attacks. We present the design and implementation of *XManDroid* (eXtended Monitoring on Android) that extends the Android permission framework. *XManDroid* analyzes communication links among applications and ensures they comply to a desired system policy. Our reference implementation is very efficient and induces small performance overhead. *XManDroid* can

prevent recently published privilege escalation attacks [13, 8, 26, 37], including collusion attacks (e.g., Soundcomber) that exploit the covert channels provided by the Android's core application.

Further, we are integrating a new concept for storing the decisions made by *XManDroid* which can be directly integrated in the standard permission framework of Android.

9. REFERENCES

[1] Google Android. `http://www.android.com/`.

[2] D. Barrera, H. G. u. c. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *ACM CCS'10: Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[3] D. Bell. Samsung Galaxy Tab Android tablet goes official. `http://news.cnet.com/8301-17938_105-20015395-1.html`, Sept. 2010.

[4] T. Bradley. Droiddream becomes android market nightmare. `http://www.pcworld.com/businesscenter/article/221247/droiddream_becomes_android_market_nightmare.html`, 2011.

[5] M. Broersma. Serious security bugs found in Android kernel. `http://www.eweekeurope.co.uk/news/serious-security-bugs-found-in-android-kernel-11040`, Nov. 2010.

[6] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.

[7] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-related policy enforcement for Android. In *ISC 2010: Proceedings of the 13th Information Security Conference*, October 2010.

[8] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *ISC 2010: Proceedings of the 13th Information Security Conference*, October 2010.

[9] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smartphone operating systems. In *20th USENIX Security Symposium*, 2011.

[10] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *NDSS'11: 18th Annual Network and Distributed System Security Symposium*, 2011. To appear.

[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI'10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.

[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *20th USENIX Security Symposium*, 2011.

[13] W. Enck, M. Ongtang, and P. McDaniel. Mitigating Android software misuse before it happens. Technical Report NAS-TR-0094-2008, Pennsylvania State University, Sep 2008.

[14] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS'09: Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[15] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security and Privacy*, 7:50–57, January 2009.

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

[17] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium*, 2011.

[18] Gartner Inc. `http://www.gartner.com/it/page.jsp?id=1689814`, 2011.

[19] P. Gilbert, B.-G. Chun, L. Cox, and J. Jung. Automating privacy testing of smartphone applications. Technical Report CS-2011-02, Duke University, 2011.

[20] GIZMODO. `http://gizmodo.com/5568458/toshiba-ac100-netbook-runs-android-and-has-massive-seven-days-of-standby-battery-life`, 2010.

[21] D. Goodin. Android bugs let attackers install malware without warning. `http://www.theregister.co.uk/2010/11/10/android_malware_attacks/`, 2010.

[22] Google. The Android developer's guide - Android Manifest permissions. `http://developer.android.com/reference/android/Manifest.permission.html`, 2010.

[23] G. Halfacree. Android trojan captures credit card details. `http://www.thinq.co.uk/2011/1/20/android-trojan-captures-credit-card-details/`, 2011.

[24] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22:36–38, October 1988.

[25] J. Leyden. First SMS Trojan for Android is in the wild. `http://www.theregister.co.uk/2010/08/10/android_sms_trojan/`, 2010.

[26] A. Lineberry, D. L. Richardson, and T. Wyatt. These aren't the permissions you're looking for. BlackHat USA 2010. `http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf`, 2010.

[27] Lookout Mobile Security. Security alert: Geinimi, sophisticated new Android Trojan found in wild. `http://blog.mylookout.com/2010/12/geinimi_trojan/`, 2010.

[28] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[29] J. K. Millen. Finite-state noiseless covert channels. In *CSFW'89: Proceedings of the Computer Security Foundations Workshop II*, pages 81–86, 1989.

[30] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *ASIACCS '10: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[31] Nils. Building Android sandcastles in Android's sandbox. BlackHat Abu Dhabi 2010. `https://media.blackhat.com/bh-ad-10/Nils/Black-`

Hat-AD-2010-android-sandcastle-wp.pdf, 2010.
[32] Notion Ink. Adam Tablet. http://www.notionink.in/.
[33] J. Oberheide. Android Hax. SummerCon 2010. http://jon.oberheide.org/files/summercon10-androidhax-jonoberheide.pdf, June 2010.
[34] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy oriented secure content handling in Android. In *ACSAC'10: Proceedings of the 26th Annual Computer Security Applications Conference*, Dec. 2010.
[35] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *ACSAC '09*. IEEE Computer Society, 2009.
[36] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile protection for smartphones. In *ACSAC'10*, Dec. 2010.
[37] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, pages 17–33, Feb. 2011.
[38] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-powered mobile devices using SELinux. *IEEE Security and Privacy*, 8(3):36–44, 2010.
[39] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev. Google Android: A state-of-the-art review of security mechanisms. *CoRR*, abs/0912.5101, 2009.
[40] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A comprehensive security assessment. *IEEE Security and Privacy*, 8(2):35–44, 2010.

APPENDIX

A. SYSTEM POLICY XML SCHEMA

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
 <element name="SystemPolicy" type="SystemPolicyType" />
 <complexType name="SystemPolicyType">
  <sequence>
   <element name="PolicyRule" type="PolicyRuleType" />
  </sequence>
 </complexType>
 <complexType name="PolicyRuleType">
  <attribute name="name" type="string" use="required" />
  <attribute name="group" type="integer" use="required" />
  <attribute name="proceed" type="integer" use="required" />
  <attribute name="maxHops" type="integer" use="optional" />
  <sequence>
   <element name="Vertex" type="VertexType" minOccurs="2"
        maxOccurs="unbounded" />
   <element name="Edge" type="EdgeType" minOccurs="2"
        maxOccurs="unbounded" />
  </sequence>
 </complexType>
 <complexType name="VertexType">
  <sequence>
   <element name="Property">
    <complexType>
     <attribute name="type" type="VPropTypes" use="required"
          />
     <attribute name="value" type="string" use="required" />
     <attribute name="negated" type="boolean" use="optional"
          />
    </complexType>
   </element>
  </sequence>
 </complexType>
 <completeType name="EdgeType">
  <sequence>
   <element name="Property">
    <complexType>
     <attribute name="type" type="EPropType" use="required"
          />
     <attribute name="value" type="string" use="required" />
     <attribute name="negated" type="boolean" use="optional"
          />
    </complexType>
   </element>
  </sequence>
 </completeType>
 <simpleType name="VPropTypes">
  <restriction base="string">
   <pattern value="(PackageName|RequestedPermissions|
        RequiredPermissions|UID|SharedUID)" />
  </restriction>
 </simpleType>
 <simpleType name="EPropTypes">
  <restriction base="string">
   <pattern value="(Data|Action|Extras|Component|Package)" />
  </restriction>
 </simpleType>
</schema>
```