Towards Taming Privilege-Escalation Attacks on Android

Sven Bugiel[1], Lucas Davi[1], Alexandra Dmitrienko[3], Thomas Fischer[2],
Ahmad-Reza Sadeghi[1,3], Bhargava Shastry[3]

[1]CASED/Technische Universität Darmstadt, Germany
{*sven.bugiel,lucas.davi,ahmad.sadeghi*}*@trust.cased.de*

[2]Ruhr-Universität Bochum, Germany
thomas.fischer@rub.de

[3]Fraunhofer SIT, Darmstadt, Germany
{*alexandra.dmitrienko,ahmad.sadeghi,bhargava.shastry*}*@sit.fraunhofer.de*

Abstract

Android's security framework has been an appealing subject of research in the last few years. Android has been shown to be vulnerable to application-level privilege escalation attacks, such as confused deputy attacks, and more recently, attacks by colluding applications. While most of the proposed approaches aim at solving confused deputy attacks, there is still no solution that simultaneously addresses collusion attacks.

In this paper, we investigate the problem of designing and implementing a practical security framework for Android to protect against confused deputy and collusion attacks. We realize that defeating collusion attacks calls for a rather system-centric solution as opposed to application-dependent policy enforcement. To support our design decisions, we conduct a heuristic analysis of Android's system behavior (with popular apps) to identify attack patterns, classify different adversary models, and point out the challenges to be tackled. Then we propose a solution for a system-centric and policy-driven runtime monitoring of communication channels between applications at multiple layers: 1) at the middleware we control IPCs between applications and indirect communication via Android system components. Moreover, inspired by the approach in QUIRE, we establish semantic links between IPCs and enable the reference monitor to verify the call-chain; 2) at the kernel level we realize mandatory access control on the file system (including Unix domain sockets) and local Internet sockets. To allow for runtime, dynamic low-level policy enforcement, we provide a callback channel between the kernel and the middleware. Finally, we evaluate the efficiency and effectiveness of our framework on known confused deputy and collusion attacks, and discuss future directions.

1. Introduction

Google Android [1] has become one of the most popular operating systems for various mobile platforms [23, 3, 31] with a growing market share [21]. Concerning security and privacy aspects, Android deploys application sandboxing and a permission framework implemented as a reference monitor at the middleware layer to control access to system resources and mediate application communication.

The current Android business and usage model allows developers to upload arbitrary applications to the Android app market[1] and involves the end-user in granting permissions to applications at install-time. This, however, opens attack surfaces for malicious applications to be installed on users' devices (see, for instance, the recent DroidDream Trojan [6]).

Since its introduction, a variety of attacks have been reported on Android showing the deficiencies of its security framework. Of particular interest and importance in this context are the so-called *application-level privilege escalation attacks* which are the main focus of this paper.

Privilege escalation attacks at application-level. Android's security framework (enforcing sandboxing and permission checks) is not sufficient for transitive policy enforcement allowing privilege escalation attacks as shown by the recent attacks [16, 12, 20, 35]. Prominent examples are *confused deputy* and *collusion* attacks. Confused deputy attacks [26] concern scenarios where a malicious application exploits the vulnerable interfaces of another privileged (but confused) application[2]. On the other hand, collusion attacks

[1]One can register as an Android developer by only paying a fee of $25. Afterwards, developers are free to publish applications on the Android market.

[2]These attacks range from unauthorized phone calls [16] and text message sending [12] to illegal toggling of WiFi or GPS service state [20].

concern malicious applications that *collude* to combine their permissions, allowing them to perform actions beyond their individual privileges. Colluding applications can communicate directly [28], or exploit covert or overt channels in the Android core system components [35]. Moreover, applications can launch privilege escalation attacks by exploiting kernel-controlled channels and completely bypass the middleware reference monitor. Examples for such attacks are confused deputy attacks over a locally established Internet socket connection, or collusion attacks over the file system [12, 35]. Hence, one needs protection at both abstraction layers: namely the middleware and the Linux kernel.

Security extensions to Android and problems. The problem of application-level privilege escalation attacks has been investigated in the last few years, and various security extensions and enhancements to Android have been proposed such as Kirin [16, 17], TaintDroid [14], Saint [33], QUIRE [13], IPC Inspection [20] to name some. However, as we will discuss in further detail (cf. Section 8), none of the existing approaches satisfactorily addresses both confused deputy and collusion attacks. While the existing solutions are either static or mostly delegate the policy enforcement to applications, we realize that tackling collusion attacks calls for a system-centric solution, not dependent on applications. Moreover, previous solutions suffer from other deficiencies such as incompatibility to legacy applications (requiring to over-privilege some applications), or inefficiency.

Our goal and contributions. We investigate the problem of building a security framework for Android to protect it against confused deputy and collusion attacks. We aim for a general framework which can capture all variations of application-level privilege attacks, as opposite to previous works targeting attack subclasses. To support our design decisions, we conducted a heuristic analysis of the runtime behavior of Android while running many popular applications to observe and identify possible attack patterns. We point out the related challenges towards tackling this problem, and discuss the trade-offs with respect to other major existing solutions some of which could be integrated in our framework to improve our solution. In particular, our contributions are the following:

- *Security Framework.* We present the design and implementation of a security framework to detect and prevent confused deputy and collusion attacks. For this, we extend Android's reference monitor concept at both the middleware and the kernel level as follows: 1) *runtime* monitoring on direct IPC calls between applications and indirect communication through Android system components. Inspired by the approach in QUIRE [13] we establish a semantic link between IPC calls that are checked at runtime by our monitor to identify call-chains to protect against confused deputy attacks (caused by intents); 2) kernel-level Mandatory Access Control (MAC) on the file system (files, Unix domain sockets) and Internet sockets; 3) a runtime interaction between our security extensions to the Android middleware and the kernel-level MAC, allowing for dynamic runtime policy mapping from the middleware to the kernel.

- *Policy enforcement.* Our policy enforcement is system-centric and uses an appropriate high-level policy language inspired by VALID [4] at the middleware layer. At the kernel level, we have adapted TOMOYO Linux [25] to the *Nexus One* smartphone. Although TOMOYO can intercept system calls and enforce MAC at the kernel level, it isn't aware of contextual information available to the Android middleware (e.g., permissions that Android applications possess) in order to take the correct decision. To bridge the gap between policy enforcement at the middleware layer and TOMOYO, we dynamically map the policies of the middleware to TOMOYO.

- *Performance and effectiveness.* Our reference implementation has a negligible performance overhead not noticeable to the user. We evaluated our implementation on a *Nexus One* development phone with 50 (popular) applications from the Android Market and 25 users (students). Note that in contrast to [15, 19] we perform our evaluation manually at *runtime*[3], and hence, 50 applications are already sufficient for our purposes. We successfully evaluated our framework against application-level privilege escalation attacks presented in [16, 12, 20, 35]. In contrast to existing solutions, our implementation detects all of these attacks including the sophisticated attacks of Soundcomber [35] and an attack launched through locally established Internet connection [12]. Finally, we discuss and evaluate the possible problems, such as the rate of attack detection and falsely denied communication.

Outline. The remainder of this paper is organized as follows: After we recall the Android architecture in Section 2, we introduce the general problem of privilege escalation attacks, present our adversary model and assumptions in Section 3. In Section 4, we present the architecture of our security framework, describe the graph-based system representation used in our framework, and provide a graph-based definition of privilege escalation attacks. Section 5 is devoted to defining a security policy for our framework. In

[3]Note that automated testing of mobile phone applications has been shown to exhibit a very low execution path coverage and is thus not suitable for our purposes [22].

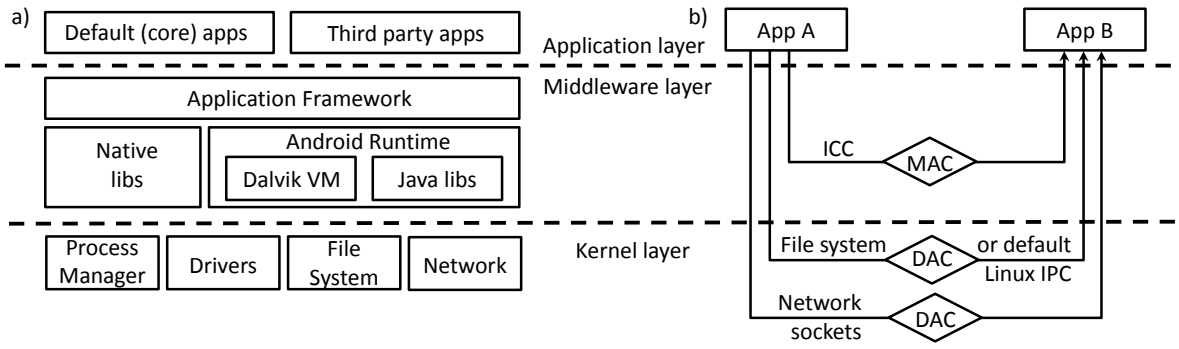

Figure 1. Android internals: (a) Android software stack; (b) Possible communication channels

Section 6, we present the implementation of our framework whose effectiveness and performance we evaluate in Section 7. Finally, we elaborate on related work in Section 8, and conclude in Section 9.

2. Android

In this section we briefly highlight the internals of the Android architecture and recall its major security mechanisms.

Android Software Stack. Android is an open source software stack for mobile devices. It builds on top of a Linux kernel and includes a middleware framework and an application layer (as depicted in Figure 1a). The Linux kernel provides basic system services to upper layers, such as process isolation and scheduling, file system support, device drivers and networking. The middleware layer includes the Dalvik Virtual Machine (DVM), Java and native libraries, and provides system services, such as the application life cycle management. Further, it provides a Mandatory Access Control system for application communication. On top, the Android application layer includes pre-installed and third party applications. Android applications are mainly written in Java, but may incorporate C/C++ code through the Java Native Interface (JNI). Further, applications basically consist of *components*, where Android provides four basic kinds of components: *Activities*, *Services*, *Content Providers* and *Broadcast Receivers*. *Activities* are associated with a user interface, *Services* implement functionalities of background processes, *Content Providers* are SQL-like databases, and *Broadcast Receivers* serve as application mailboxes for event notifications.

Application Communication Channels. Possible communication channels are shown in Figure 1b. Typically, Android applications communicate through a standard mechanism provided by the middleware, namely a Binder-based lightweight Inter-Process Communication (IPC) channel. Binder IPC calls occur at the granularity of application components. Thus, the resulting application communication is often referred to as inter-component communication (ICC) [18] to differentiate it from IPCs at the kernel level. To establish an ICC channel, application components send a special message called *Intent*. Intents typically encapsulate data that describe the task to perform (e.g., launch activity).

Besides ICC, applications may communicate over channels that bypass Android's middleware, but are controlled by the underlying Linux kernel. For instance, communication can be established via Linux's standard IPC mechanisms (e.g., Unix domain sockets, files) or via Internet sockets. Currently, only TrustDroid [9] is able to provide a means to enforce Mandatory Access Control to prevent communication between applications over these channels. However, TrustDroid attempts to solve the problem of domain isolation in a corporate context and is not intended as a solution against privilege escalation attacks.

Android Sandboxing. The underlying Linux kernel enforces process isolation and discretionary access control to resources (files, devices) by user ownership. To sandbox applications, every application instance in Android is assigned a unique user identifier (UID), while system resources are owned by either the *system* or *root* user. Applications can only access their own files, or files that are explicitly defined as world-wide readable.

Android Permission Framework. Security sensitive resources such as a phone call interface, Internet access or user contacts database are protected by permissions: namely security labels that are defined by the Android system. The list of standard Android permissions contains 110 items[4] [24]. Most security sensitive resources of Android are provided by a system application called Android. Application developers must declare which permissions are required to properly

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

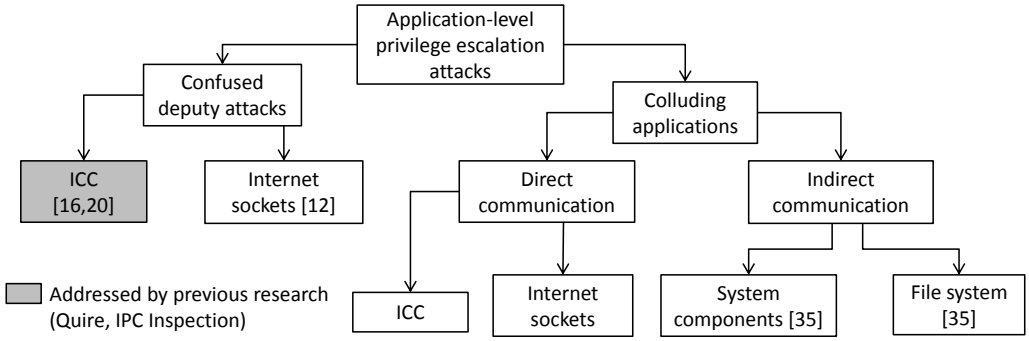

Figure 2. Classification of application-level privilege escalation attacks

execute the application. Moreover, they may define new permissions in order to protect access to sensitive application interfaces. Both newly defined and required permissions are included in a *Manifest* file, which is part of an application's installation package. At install time, applications request the necessary permissions from the user. The user can either grant all the requested permissions, or abort the installation process. Once granted, application permissions cannot be changed. Further, Android's middleware layer enforces Mandatory Access Control (MAC) on ICC calls. Android's reference monitor checks permission assignments at runtime and denies ICC calls in case the caller does not have the required permissions.

Exceptionally, several permissions (such as INTERNET, BLUETOOTH, WRITE_EXTERNAL_STORAGE) are not controlled by Android reference monitor, but are mapped onto Linux groups and are enforced by a low-level access control of Linux.

3. Problem Description

In the following, we propose a classification of application-level privilege escalation attacks and define our adversary model, requirements and assumptions.

3.1. Attack Classification and Adversary Model

We classify application-level privilege escalation attacks into two major classes as depicted in Figure 2: (i) confused deputy attacks and (ii) attacks by colluding applications.

The first class concerns malicious applications (under the adversary's control) leveraging unprotected interfaces of a benign application. Recent research results show that confused deputy vulnerabilities are common in both third party applications [12, 20] and Android default applications such as Phone [16], DeskClock, Music, and Settings [20]. Further, confused deputy attacks can be classified based on the channel used for privilege escalation i.e., either ICC-based [16, 20] or socket-based [12].

The second class concerns malicious applications that collude to merge their permissions and gain a permission set which has not been approved by the user. For instance, in the Soundcomber [35] attack, one application has the permission to record audio and monitor the call activity, while a second one owns the Internet permission. When both applications collude, they can capture the credit card number (spoken by the user during a call) and leak it to a remote adversary. In general, colluding applications can communicate either directly, e.g., by establishing direct ICC channels, or via a locally established socket connection, or indirectly, e.g., by sharing files or through overt/covert channels in system components of Android (as performed by Soundcomber [35]).

We consider a scalable adversary model with the following types of adversaries: WeakAdversary is able to launch *known*[5] confused deputy attacks over ICC channels. The BasicAdversary can launch all kinds of confused deputy attacks, including unknown attacks and attacks over Internet sockets. AdvancedAdversary can launch any confused deputy attack and also (unknown) collusion attacks that occur via direct ICC calls. Finally, StrongAdversary can launch any privilege escalation attack (at application-level), including collusion attacks that are performed via indirect communication, e.g., by sharing files, through Internet connection, or through channels established in system components such as Android.

Note that previous research works [33, 13, 20] aim at tackling only the problem of unknown confused deputy attacks over ICC channels, while we aim to work towards a framework that addresses all types of attacks mentioned above.

[5]Obviously, known attacks can be fixed, however, this adversary targets a policy-based approach offering hot fixes. Note that, remote app kill is not an appropriate solution against this adversary, as confused deputies are not malicious, and even system apps have been shown to suffer from confused deputy vulnerabilities [20].

3.2. Objectives and Requirements

Ideally, we aim to address the StrongAdversary model, and discuss in Section 7 that there are several challenges to tackle when designing and implementing a framework which is both general and fine-grained at the same time. Hence, there is usually a trade-off between attack coverage and granularity of system analysis and policy enforcement. Too coarse-grained analysis may result in negative effects, e.g., in false positives, thereby limiting the usability, whereas attack-specific solutions are obviously limited and may fail in defeating other attacks in realistic scenarios. Further, the intuitive idea of combining the existing attack-specific protection mechanisms may not suffice either, as those may not be compatible with each other and will likely induce significant cumulative performance overhead.

To address this challenge, we require a configurable framework which is flexible and can be adjusted to meet different trade-offs between security and usability. In particular, we require a solution which can be configured to be effective in WeakAdversary, BasicAdversary, AdvancedAdversary and StrongAdversary models depending on the requirements imposed by the underlying scenario. We further require (i) a *system-centric protection* because delegating the policy enforcement to applications is risky since the applications themselves might be either vulnerable or even malicious (given the huge number of applications available on the market), (ii) *legacy compatibility* since recompilation of (many or even all) applications in the Android market would be impractical, and (iii) *low performance overhead* since in mobile usage scenarios, the imposed performance overhead due to security mechanisms should not affect usability.

3.3. Assumptions

We make the following assumptions with regard to our Trusted Computing Base (TCB): we assume that the Linux kernel and Android's middleware are not malicious. Moreover, we assume that default Android applications are not malicious[6], but they may suffer from confused deputy vulnerabilities. Further, the Android application may suffer from design deficiencies that allow malicious applications to establish indirect communication links (see e.g., [35]).

4. Framework Architecture

After providing a short overview of our security architecture, we will describe in detail the involved components in our architecture and how they interact with each other. Finally, we introduce the graph-based system representation of our architecture.

[6]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.

4.1. Overview

Our security framework performs runtime monitoring and analysis of communication links across applications in order to prevent potentially malicious communication based on a defined system-centric security policy. We maintain a system state that includes the applications installed on the platform, files, Unix sockets and Internet sockets, as well as direct and indirect communication links established among applications. Direct communication links are ICC calls, while indirect channels are, e.g., shared files. Our extension is invoked when applications attempt to establish an ICC connection, access files or connect to sockets. The framework validates whether the requested operation can potentially be exploited for a privilege escalation attack (based on the underlying security policy).

4.2. Component Interaction

The architecture of our framework is shown in Figure 3. Generally, it builds on our previous work, a framework called XManDroid [8] that enhances Android's middleware. In this work, we extend XManDroid with a kernel-level module. In the following paragraphs, we shall describe the components in our architecture and their interaction in the following use cases: ICC call handling (steps 1-11), application (un)installation (steps a-b), file/socket creation (steps α - γ), file/socket read/write access (steps i-iv), and policy installation (steps I-III).

ICC call handling. At runtime, all ICC calls are intercepted by the Android ReferenceMonitor (step 1). It obtains information about permissions from the AndroidPermissions database (step 2) and validates permission assignments. If ReferenceMonitor allows an ICC call to proceed, it invokes DecisionEngine (step 3) to ensure that the communication additionally complies with the underlying system security policy. DecisionEngine first requests for a record corresponding to this particular ICC call from the PolicyDecisions database (step 4). If a matching record is found, it means that a previously made decision can be applied. Otherwise, DecisionEngine makes a fresh decision based on inputs from AndroidPermissions (step 5), SystemPolicy (step 6) and SystemView (step 7). In particular, the SystemView is represented as a *graph*, where entities such as applications are represented as nodes, and their established communication links as edges (more details are provided in Section 4.3). Subsequently, the resulting decision is stored in the PolicyDecisions database (step 8), and if it is positive, SystemView is updated to reflect the

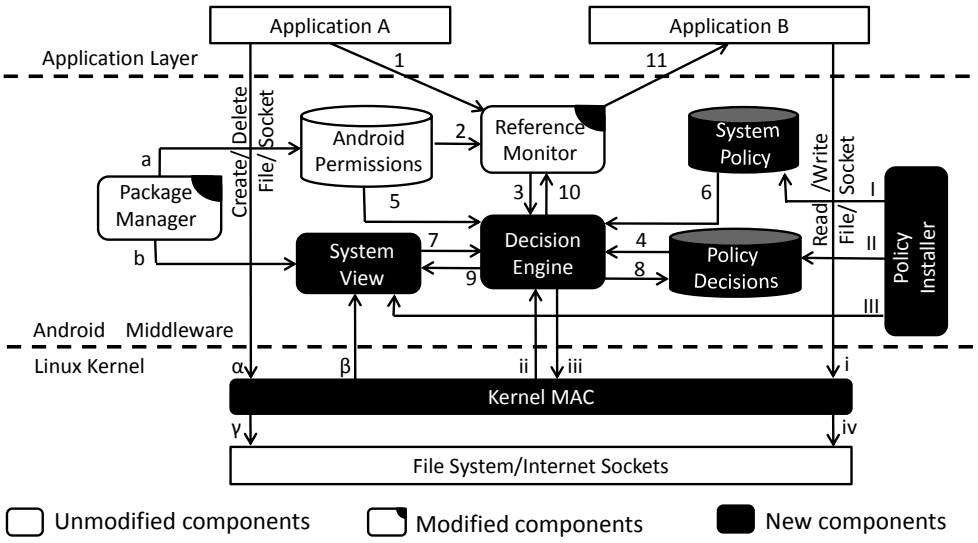

Figure 3. Framework Architecture

fact that a communication link exists among the components of applications A and B (step 9). Further, DecisionEngine informs ReferenceMonitor about the decision it has made (step 10), and ReferenceMonitor either allows (step 11) or denies the ICC call.

Application (un)installation. The installation procedure involves the standard PackageManager of Android. Typically, it extracts the application permissions from the *Manifest file* and stores them in the AndroidPermissions database (step a). In addition, PackageManager adds a new app into SystemView (step b). Upon application uninstallation, PackageManager removes all entries of the uninstalled application from the AndroidPermissions database (step a) and removes the app from SystemView (step b).

Operations on files, Unix domain sockets and Internet sockets. When an application performs a file or Unix domain socket operation (create, read/write, connect, etc.), or tries to establish a connection over an Internet socket, the operation is intercepted by KernelMAC: a Mandatory Access Control (MAC) module in the Linux kernel. When a file/socket is created (step α), KernelMAC updates SystemView accordingly (step β) and performs the requested action (step γ). Upon request for read access to files, connecting to Unix sockets, or connecting to local Internet sockets (step i), KernelMAC looks up its internal policy file to check if the requested operation can be allowed to proceed. If such a rule does not exist in its internal policy file, KernelMAC requests a policy decision from DecisionEngine (steps ii and iii), and subsequently denies/grants the operation accordingly (step iv). Furthermore, KernelMAC could be instructed to cache policy decisions (relayed by DecisionEngine) in its internal policy file for future use. Doing so reduces the performance overhead induced by context switches between the kernel and the middleware.

Policy installation. PolicyInstaller writes (updates) the system policy rules to the SystemPolicy database (step I). Next, it removes all decisions previously made by the DecisionEngine, as those may not comply with the new system policy (step II). Also, SystemView component is reset to a clean state (step III), i.e., all previously allowed communication links among applications are removed. Note that SystemView state is only reset upon update of SystemPolicy and persists across reboots.

Intent tagging. An important feature for fine-grained analysis of communication links that can lead to confused deputy attacks is to establish causal relations between different ICC calls and/or Intents. Inspired by the solution of QUIRE [13], our framework builds a call-chain of the UIDs in cohering ICCs. In contrast to QUIRE, we opted for a system-centric call-chain. We started to integrate such a mechanism into Android's Binder code to cover all kinds of IPC. Our first experiments demonstrate that such a mechanism is feasible and can be done efficiently. However, in our current design, we chose an Intent-based approach by automatically tagging newly created Intents with the UID of the calling application. This information is used by DecisionEngine to re-create the path in the graph (which is part of the SystemView) that lead to the current ICC.

4.3. System View Instantiation

Graph-based representation. We opted for a graph-based instantiation of the system state SystemView, where vertices represent entities such as application sandboxes, system components, files[7] and Internet sockets while edges represent the communication and access links (among them). In our design, we record file system access as well as access to Internet sockets in the graph, as edges inserted between, e.g., a file and an application writing or reading this file.[8] Generally, a graph-based representation allows different levels of system abstractions: for instance applications can be represented at the level of application sandboxes, application packages, or application components, while communication links can be seen as directed or undirected edges, thus either providing exact information about the direction of information or control flows or always assuming a bidirectional flow between graph vertices. Hence, we consider the Android system as a graph G consisting of a set $\mathcal{V}(G)$ of vertices and a set $\mathcal{E}(G)$ of direct edges. An edge $e \in \mathcal{E}(G)$ is an ordered pair $e = (i; j)$, where $i, j \in \mathcal{V}$: $i \neq j$.

Graph Vertices. We represent applications at the sandbox level because applications residing within the same sandbox have the same privileges, and thus cannot escalate privileges inside the sandbox[9]. However, the Android system application is an exceptional case and requires a more fine grained representation. It consists of system services and system content providers, which, by design, provide overt as well as covert channels between applications. For instance, applications can insert data into and read data from system content providers such as the contacts database (overt channel), or perform synchronized write-read operations on the settings of a system service such as the audio manager (covert channel). Thus, Android can mediate communication of other applications, and hence we take this into account in our analysis.

If represented as a single vertex in the graph, Android would cause transitive closure for many vertices because many applications access Android (cf. Figure 4a) to get system services. However, a finer-grained representation of Android, e.g., at the level of application components is challenging because Android system application is realized as a monolithic application. To resolve this issue, we introduced two important design extensions (discussed in more details in our technical report [8]): (i) extraction of the system content providers and services from the monolithic Android and representing them in the system graph as *virtual nodes*, because they are assigned virtual UIDs (cf. Figure 4b), and (ii) fine-grained policy-based filtering of data in those providers and services (cf. Section 6). Hence, we define four types of graph vertices $\mathcal{A}, \mathcal{S}, \mathcal{F}, \mathcal{I}$ representing the set of of application sandboxes, system components, files, and Internet sockets.

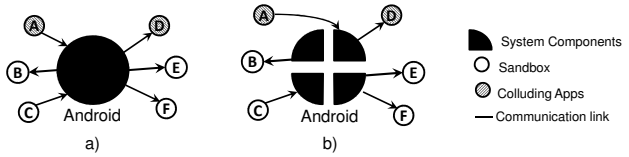

Figure 4. (a) Android system app provides transitive closure to other apps; (b) Android system app is split up into system components (System Services and System Content Providers)

Graph Edges. We define the following three types of edges: (i) ICC calls $\mathcal{M}$, (ii) file access $\mathcal{K}$, and (iii) Internet sockets/connections $\mathcal{H}$. Note that we represent ICC calls among two application sandboxes and established Internet connections as bidirectional edges in the graph.While representing ICC calls with bidirectional edges precludes certain legitimate communication, this over-approximation is necessary. For instance, ICC calls and Internet connections often result in bidirectional data-flows, not observable at our level of system abstraction. Similarly, pending Intents[10] obfuscate the actual data flow in the system graph, such that they are currently not representable with unidirectional edges. In contrast, the direction of data flow upon (read/write) access to files, as well as ICC calls to system components can be precisely distinguished. Edges in $\mathcal{M}$ can connect vertices representing application sandboxes $\mathcal{A}$ and/or system components $\mathcal{S}$: $\forall e \in \mathcal{M}$: $e = (i, j)$, $i, j \in \mathcal{A} \cup \mathcal{S}$, $i \neq j$. Edges in $\mathcal{K}$ can only connect pairs of vertices, where one vertex is an application sandbox in $\mathcal{A}$ and the other one is a file in $\mathcal{F}$ (ordered pair representing read/write): $\forall e \in \mathcal{K}$: $e = (i, j)$, $(i \in \mathcal{A}, j \in \mathcal{F}) \vee (i \in \mathcal{F}, j \in \mathcal{A})$. Edges that represent Internet connections can only connect pairs of vertices, where one vertex is an application sandbox and the other one is an Internet socket $\forall e \in \mathcal{H} : e = (i, j)$, $i \in \mathcal{A}$, $j \in \mathcal{I}$. Figure 5 depicts a graph snapshot with all defined types of vertices and edges.

[7] We do not distinguish files and Unix sockets because Unix domain sockets have much in common with files (e.g., file system address name space and access permissions) and can be treated in the same way.

[8] Alternative approaches may, e.g., rely on theorem solvers such as Prolog (see [16]). However, our early experiments (with tuProlog http://alice.unibo.it/xwiki/bin/view/Tuprolog/) showed that a Prolog-based approach suffers from manageability problems at runtime and induces user-notable performance overhead.

[9] The problem of malware comprising a single application or an application sandbox is orthogonal to the problem of privilege escalation and is addressed by previous research, e.g., by Kirin [17].

[10] The sender of an Intent delegates the insertion of payload to another app before sending the Intent.

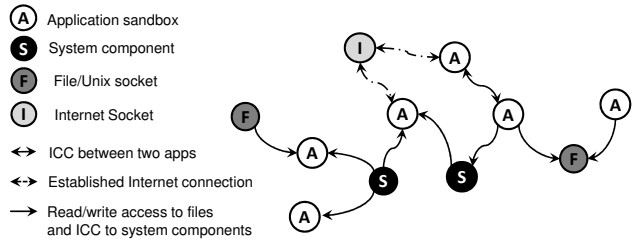

Figure 5. System graph representation

Vertex Properties. Each vertex V in the system graph G is assigned the corresponding properties: a unique identifier $U(V)$ (i.e., UID for a sandbox, virtual UID for a system component, path/file name for files and IP address and port for Internet sockets), and a trust level $T(V) = (true, false)$ where we currently distinguish two trust levels: "untrusted" are third party applications, while "trusted" are default Android apps and system components. Moreover, file and Internet socket vertices are also considered as trusted. This is because files or sockets themselves cannot perform a privilege escalation attack, but may mediate communication of malicious applications in the same way as trusted system components. Moreover, application sandboxes feature additional properties, particularly, names of applications included in a sandbox (e.g., package names) $\mathcal{N}(A)$, a list of application components for each application $\mathcal{C}(A)$, and the set of granted permissions $\mathcal{P}(A)$ to application A.

Path. A path $L(v_1, v_n) = (e_1, e_2, ..., e_{n-1})$ in a graph G between vertices v_1 and v_n is a sequence of edges $e_1 = (v_1, v_2), e_2 = (v_2, v_3), ..., e_{n-1} = (v_{n-1}, v_n)$, where $e_1, ...e_{n-1} \in \mathcal{E}$, $v_i \neq v_j$ for $i, j \in 1, ..., n$. In other words, a path is an acyclic sequence of edges connecting vertices. We define two path properties: (i) a path length, which is denoted as $|L(v_1, v_n)| = n - 1$, and (ii) a set of data $\mathcal{D}(\mathcal{L})$ transmitted through the path.

Privilege escalation on the graph. We define patterns of privilege escalation attacks by specifying a set of critical permissions in the system. For instance, gaining of privileges allowing both to access the Internet and location information can be defined as an attack pattern corresponding to a location tracker. We denote the *critical permissions* set by $\mathcal{Z}$. In a confused deputy attack, an unprivileged application $A \in \mathcal{A}$ can obtain a critical set of permissions $\mathcal{Z}$ by invoking a privileged application $B \in \mathcal{A}$, if there exists a communication path $L(A, B)$. The privilege escalation attack by colluding applications $A, B \in \mathcal{A}$ can occur, when there is a communication link $L(A, B) \vee L(B, A)$ between two applications, and each individual set of privileges $\mathcal{P}(A)$ and $\mathcal{P}(B)$ does not match the critical set of privileges, while a union of both does so, i.e., $\mathcal{Z} \nsubseteq \mathcal{P}(B) \wedge \mathcal{Z} \nsubseteq \mathcal{P}(A) \wedge (\mathcal{Z} \subseteq (\mathcal{P}(A) \cup \mathcal{P}(B)))$.

5. Policy

Different policy profiles can be created in order to cover different adversary models and user requirements. For instance, the following profiles can be specified: DefaultProfile, BasicProfile, AdvancedProfile and StrongProfile, that correspond to adversary models WeakAdversary, BasicAdversary, AdvancedAdversary and StrongAdversary (defined in Section 3) respectively. DefaultProfile can be activated by default, in the form of policy rules that help defend against known confused deputy attacks occurring via an ICC channel. Furthermore, policy rules could be defined such that they do not result in false positives (confirmed by our evaluation results, Section 7) thereby not affecting usability. This profile can be updated, e.g., by Google, in form of security patches upon discovery of new attacks. The profiles BasicProfile, AdvancedProfile and StrongProfile can be activated by users with higher security requirements who are willing to tolerate (a small number of) possible false positives in favor of increased security. For instance, these profiles can be useful in a corporate context, where an enterprise issues mobile devices to its employees and allows them to use their devices for both private and business purposes. The company's system administrators may then have the possibility of generating custom policy profiles that address the security requirements of the company.

For expressing policy rules, we define a policy language inspired by VALID [4], a formal security assurance language developed for virtualized infrastructure topologies. VALID has been shown to be effective when validating policies on graph based models of virtualized infrastructures [5]. It expresses high-level security goals (policy rules in our terminology) for such environments in the form of attack states. The language is very suitable for our purposes because it is capable of expressing operations on graphs, such as information flow in a graph model of the underlying topology. When mapped to our system representation graph (cf. Section 4.3), it is used to express system states that describe privilege escalation attacks. Security goals can then state the information flow through edges in the graph that could result in privilege escalation. VALID describes properties of graph vertices, but is limited in expressing path properties. For our purposes, we extended VALID to specify path properties, such as path source, path destination, data transmitted over the path and so on (cf. Section 4.3).

We illustrate a policy rule to preserve privacy of phone calls in Figure 6. For instance, we already explained the attack scenario of Soundcomber [35] in Section 3, which targets privacy of user phone calls. The rule defines an undesirable set of permissions that would allow malware to

```
Section types:
A,B : Application sandboxes
L(A,B) : Path

Section goals:
goal ProtectCallPrivacy(deny) := L.connects(A,B) ∧ L.type(any) ∧ A.trustLevel(
      untrusted) ∧ ¬(A.hasPermission(INTERNET) ∧ (A.hasPermission(
      PHONE_STATE) ∨ A.hasPermission(PROCESS_OUTGOING_CALL)) ∧ B.
      trustLevel(untrusted) ∧ ¬(B.hasPermission(PHONE_STATE) ∨ B.
      hasPermission(PROCESS_OUTGOING_CALL)) ∧ B.hasPermission(
      INTERNET)
```

Figure 6. A policy rule to defeat attacks of malware monitoring phone calls

record phone calls and transfer recorded data to the Internet. Further, it only restricts communication between two applications that do not individually possess the undesirable set of permissions, but would obtain it when their permission sets are combined. The applications are not allowed to communicate via any of possible communication channels (direct and indirect ICC, file system, or a local Internet connection).

Further examples of policies can be found in Appendix B.

6. Implementation

In this section we present the implementation of our architecture and its main components, as presented in Section 4.2. Our implementation is based on the Android 2.2.1 sources[11].

System View. We implemented the system view graph by means of the open-source *JGraphT*[12] library, version 0.7.3. The initial system graph is built upon first boot of the device. We use the PackageManager to discover installed applications during the graph building process. In case of a sandbox that is shared by several applications, the information from all these applications with the shared UID is merged in the corresponding vertex. Vertices that represent files and Internet sockets are added to the system graph at runtime, at first access request to the file/socket. To keep the overhead low, we add only world-wide readable/writable files to the graph. This is sufficient for our analysis, because private files cannot be shared by different sandboxes (due to the Discretionary Access Control of Linux kernel).

To extract system content providers from the monolithic Android, we extended the ActivityManager of Android. Usually, the ActivityManager detects and registers all installed content providers in the system during boot-up. We extended this process such that the ActivityManager inserts a vertex for each of them in the graph via an interface of SystemView. We pre-install vertices that correspond to system services in the graph (as those are known and fixed). We opted not to use Android's Service Manager that maintains a list of the registered services, because it does not distinguish system and third party services. Further, we modified the content providers' interfaces to prevent colluding applications from exchanging data over content providers. Each data row in the database is tagged with the UIDs of the writers. Upon writing data, these tags are updated. Upon reading, our extension to the reading interface verifies for each read row if the corresponding reader-writer pairs constitute a policy violation. If so, the corresponding row is filtered from the response to the reader before return.

To enforce policy checks within system services, we tag each value of the service with the UID of the writer. Note, in contrast to content providers, values in system services can be only overwritten, thus we store the UID of the last writer. Tagging of values is implemented as a mapping from value to writer UID. Upon reading of values, we use a similar interface as for content providers to perform a policy check if the reading would trigger a policy violation. If so, a *null* value is returned.

Package Manager. We modified the PackageManager to update the system graph upon installation or uninstallation of an application package. The installation adds a new vertex for the new application UID to the graph. In case of shared UIDs, the information of all applications under this UID are merged. Application uninstallation causes the removal of the corresponding vertex. Uninstallation in the case of a shared UID is slightly more complex, since only the information added by the uninstalled application has to be removed. We implemented this by removing the corresponding vertex and reconstructing it afterwards with the updated application information.

Policy Decisions. PolicyDecisions stores a boolean for the decision result of each checked ICC, file access or socket connection, i.e., granted or denied. It is implemented as a mapping from distinct ICC to decision result. The index of the mapping is the unordered tuple $\{uid\_caller, uid\_callee\}$ for each inserted decision. PolicyDecisions is persistent across system reboots.

Decision Engine. DecisionEngine implements a policy checking algorithm, which determines if a new edge in the system graph would complete a path in SystemView, which matches a policy rule. The algorithm uses backtracking[13] to explore, in a depth-first search fashion, all paths consisting only of vertices that match a distinct vertex description in

[11] `http://source.android.com`
[12] `http://www.jgrapht.org/`

[13] Backtracking algorithms incrementally build candidates to the solution for the targeted problem and abandon partial candidates, when they detect that this candidate does not lead to a valid solution.

the same policy rule. This procedure is continued until it finds a path that matches the policy rule exactly. It further compiles the high-level policy rules (written in our adaption of VALID [4]) into XML (based on a custom XML schema), which is more efficiently parsed during policy checking.

Policy Installer and System Policy. The system policy is implemented in XML, which is loaded upon system boot or after an update of the policy, respectively. SystemPolicy is installed or updated via the 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.

Reference Monitor. We modified the default ReferenceMonitor of Android (which is a part of ActivityManager) to redirect the control flow to our DecisionEngine whenever an ICC occurs. 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, it invokes DecisionEngine. To enable the DecisionEngine to make the policy check on direct ICCs, 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 DecisionEngine. To handle broadcast Intents, we followed the approach presented in [33, 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.

Kernel MAC. To enable mandatory access control at the kernel level, our implementation employs *TOMOYO Linux*[14] v1.8, a path-based MAC implementation available as a kernel patch. An alternative to TOMOYO is SELinux [29], a type-based MAC implementation available as a Linux Security Module (LSM). However, SELinux enforces MAC by means of type enforcement which typically requires extended file attributes to be enabled in the filesystem. Since Android's flash file system (YAFFS2) does not support extended file attributes by default, using SELinux requires prior file system modifications. On the other hand, TOMOYO, being a path-based MAC implementation, does not require any file system modifications prior to usage. Furthermore, SELinux is harder to administer on a mobile device due to complex policy rules that need to be adapted to meet Android's security requirements.

Our choice of TOMOYO over SELinux was primarily motivated by the fact that the former has a readily available user-space interface that can be extended in order to provide a feedback channel between the kernel and the middleware; the latter lacks such an interface thereby precluding communication between the kernel and the middleware. To enable communication between the two layers, we wrote a native library with access to TOMOYO's interfaces and compiled it against the Java Native Interface (JNI). In order to enable runtime policy updates, we extended an interface of TOMOYO which allows it to seek a decision from a *supervisor* (e.g., our DecisionEngine). Further, to make the Android middleware understand what TOMOYO is asking for and vice-versa, we implemented a parser function in Java. To enable the middleware to make a decision at runtime, TOMOYO passes the UID of the process making a system call along with details pertaining to the call itself, e.g., path of file to be read. The middleware's DecisionEngine processes the request and conveys its decision to TOMOYO.

It is important to note that the TOMOYO kernel boots with a carefully written security policy catered for Android. This policy file is TOMOYO specific and is loaded in the kernel memory during device boot. On intercepting a system call post device boot, TOMOYO inspects its internal policy file to see if there is a policy rule that allows the system call to proceed normally. If yes, the request is granted *without* querying the DecisionEngine (in the middleware). If no, TOMOYO queries the DecisionEngine for a decision on the request made. If the DecisionEngine deems it safe to grant the request, it conveys the same to TOMOYO. There are two ways in which the DecisionEngine could relay a *grant* decision: (1) It could simply request TOMOYO to allow the specific system call to proceed normally, or (2) It could in addition to (1) request TOMOYO to add the decision to TOMOYO's policy file. Such a flexibility allows our framework to reduce the number of context switches between the middleware and the kernel which in turn reduces the performance overhead that could result from frequent context switches.

7. Evaluation

We begin this section with a heuristic study of communication patterns between third party applications that, in part, motivated our design decisions. Subsequently, we provide test results on effectiveness and performance of our framework and discuss challenges and problems therein.

Test methodology. As methodology for our evaluation, we opted for manual testing with a group of 25 test users, 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 [22]). With respect to this limitation, we argue that 50 selected applications from different market categories (e.g., games or social tools) form a representative testing set. The test users' task was to install and thoroughly use the provided apps, to

[14] http://tomoyo.sourceforge.jp/

trigger as much as possible of the apps' features and with interleaving installation, uninstallation, and usage.

7.1. Study of 3rd Party Application Communication

We performed a heuristic analysis of the communication patterns between third party applications from the Android Market. A graphical representation of our results is given in Appendix C, and at this point, we present our main observations and thereby motivate our design decisions (cf. Section 4).

File system and socket based communications. Figure 10 in Appendix C depicts the observation that the applications we tested neither share their data with other applications at the file system level, nor communicate with each other via Unix domain sockets. Consequently, an attack vector that uses files or Unix sockets as communication medium could be easily identified making it easier to prevent such an attack. To effectively prevent this attack vector, a kernel-level MAC is required, since applications can make use of native code which circumvents any security mechanisms in Android's middleware (cf. Section 2).

Moreover, since legitimate applications are far less likely to communicate this way, the rate of falsely denied communications is expected to be low. To illustrate this point, we developed sample apps that use a Unix socket or a file for communication and this pattern is clearly distinguishable from other applications.

ICC based communication. Our observation regarding ICC based communication shows that applications usually operate autonomously and do not have functional interdependencies with other applications. Exceptions are custom launcher applications which start the Activities of other apps and apps with a *"share with"* functionality that receive data from other apps for sharing (e.g., Facebook, Twitter etc.) The usual way for apps to share data is (System) Content Providers.

Our design accurately addresses this communication pattern, since (1) it implements a very fine-grained policy enforcement in the system components, and (2) direct communication between apps, which is the main target of generic system policy rules, occurs seldom and if so, with a very distinct pattern (*start Activity* or *share with*).

7.2. Effectiveness

We evaluated the effectiveness of our solution based on the detection rate of attacks specified in the system policy (i.e., the *false negative* rate) and the rate of falsely denied communications between applications (i.e., *false positive* rate). An ideal solution would provide both a zero false negative and zero false positive rate. Our current instantiation applies over-approximation of communication links, except for Intents (cf. Section 4.3). This means that it assumes a relation between communication channels where none might exist, e.g., it has no false negatives but tends to cause false positives.

Attack detection rate. To evaluate the detection rate of privilege escalation attacks, we developed sample applications that implement the attacks described in [35, 20, 16, 12] and deployed a system policy that contained rules targeting these attacks (see Table 3 in Appendix B). Our framework successfully detected all of the above mentioned attacks at ICC, socket, and file system level, including all the attack scenarios of Soundcomber [35] launched via covert channels in Android system components and a file based covert channel.

Falsely denied communication rate. We evaluated with our user test the impact of our security framework on the usability of third party applications. During this test, in addition to the policy rules from the attack detection test, we deployed generic rules to prevent the leakage of sensitive information like the device location, contacts, or SMS via the Internet. Moreover, we deployed a singular rule that allows applications to launch other applications with an Intent if and only if the Intent does not contain any additional data/information (see rule 12 from Appendix B)[15].

To our surprise, the results of our tests showed no false positives. This is on the one hand counter-intuitive, since the policy rules of the AdvancedProfile and the StrongProfile are rather generic and are indicative of a higher rate of false positives. On the other hand, this result confirms our observations and conclusions on the communication patterns between third party applications on Android (cf. Section 7.1).

7.3. Performance

Our solution imposes only a negligible runtime overhead, not perceivable by the user. Tables 1 and 2 present our measurement results. In particular, as Table 1 shows, our framework performs quite steadily in terms of a very low standard deviation; the runtime system call latency is low considering the ratio of cached to uncached decisions. Only the overhead on intent messages cannot be optimized through caching. However, in our ongoing work we implemented a higher efficient and faster system-centric ICC call-chaining based on modifications to the Binder mechanism instead of only Intents.

[15]Note, data-less Intents can be used by the adversaries to establish covert communication. Thus, StrongProfile should include less generic exceptional rules which, e.g., additionally specify application names.

Type	Calls	Average (ms)	Std. dev. (ms)
Original Reference Monitor runtime for ICC			
system	11003	0.184	2.490
DecisionEngine overhead for ICC			
uncached	312	6.182	9.703
cached	10691	0.367	1.930
Intents	1821	8.621	29.011
DecisionEngine overhead for file read			
file read	389	3.320	4.088

Table 1. ICC timing results

Type	Average (ms)	Std. dev. (ms)
Read access to System Content Providers		
total number of accesses: 591		
read	10.317	41.224
overhead	4.983	36.441
Read access to System Services		
total number of accesses: 87		
read	8.578	20.241
overhead	0.307	0.4318

Table 2. Timing results for system components

On read access to System Content Providers (Table 2), the filtering of values conflicting with system policy imposes an overhead of approximately 48%. On read access to System Services, the overhead is merely about 2.4% on average. This discrepancy stems from the fact that, on read access to Content Providers, usually multiple reader-writer pairs have to be checked, while access to System Services involves only one reader-writer pair.

7.4. Impact on 3^{rd} party applications' usability

Although we did not observe any false positives in our tests, any falsely denied communications must be avoided, because they can have a severe impact on the usability of the smartphone. Denying applications ICC that was expected to be successful, most likely renders these application dysfunctional. Moreover, application developers do not anticipate this situation, since installed applications have been granted all the requisite permissions, and thus often omit exception handling code, causing applications to crash in case their ICC call is denied.

This problem applies to all approaches based on revoking permissions at post-install time or on denying ICC at runtime, as we will explain in Section 8 on related work. It is particularly hard to solve for situations where one cannot clearly distinguish between a confused deputy attack and a legitimate user action, e.g., when an app that provides a "share with" functionality receives an Intent to share data or when the browser is called to open a particular URL.

In our solution, we strive for minimizing the number of false positives by (1) decomposing the monolithic Android into distinct services and content providers in our system graph and (2) by performing a heuristic analysis of the communications patterns of applications. The former one facilitates a more fine-grained policy check on access to data shared via system components, while the latter is used to refine the policy design.

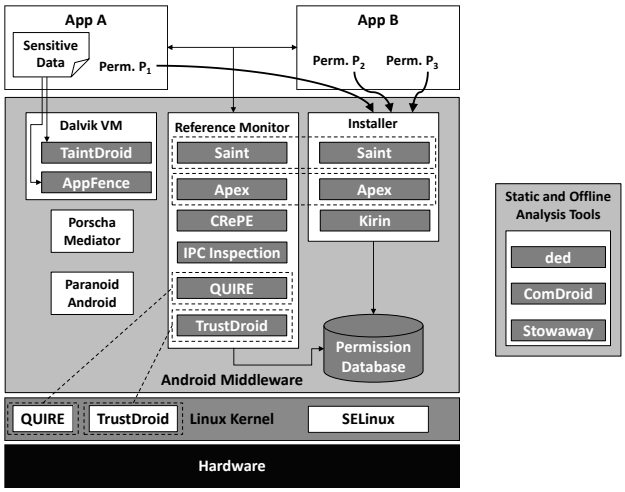

Figure 7. Security extensions for Android

8. Related Work

Figure 7 shows security extensions for Android that have been proposed in the past. Since most proposed solutions require changes to Android's middleware, we highlighted the main middleware components such as the application installer, the reference monitor, the permission database, and the Dalvik virtual machine. Each solution requires extension to one or more components, as we shall describe in the following paragraphs.

As Figure 7 shows, several security extensions to Android's security framework have been proposed over the last few years [16, 17, 33, 36, 30, 14, 11, 32, 34]. However, not all of them target privilege escalation attacks: Porscha [32] provides policy-oriented secure content handling; Apex [30] allows users to selectively grant and deny permissions at install time; CRePE [11] enables the enforcement of context-related policies; Paranoid Android (PA) [34] aims to detect viruses and runtime attacks. Finally, several static analysis tools have been proposed that mainly aim to detect vulnera-

ble application interfaces [10, 15] or analyze if applications follow the least-privilege principle [19].

Hence, in the following paragraphs, we shall discuss the strengths and shortcomings of security extensions that are closest to ours.

Kirin is an extension to Android's application installer [16, 17]. Kirin checks the permissions requested by applications at install-time. It denies the installation of an application if the permissions requested by the application encompass a set of permissions that violates a given system-centric policy. The main security goal of Kirin is to mitigate malware contained within a single application. In addition, the Kirin framework described in [16] also allows identification of security-critical communication links by analyzing which interfaces the new application is authorized to contact. However, due to its static nature, Kirin has to consider all potential communication links over the unprotected interfaces. This will stop any application from being installed, since applications can potentially establish arbitrary communication links over the unprotected interfaces. In contrast to Kirin, our framework (1) focuses on real (runtime) communication links rather than potential ones, (2) decides at runtime if the link (including communications over unprotected interfaces) to be established violates the system policy.

Saint [33] introduces a fine-grained access control model that allows application developers to attach security policies to their applications, in particular, to the application's interfaces. It enforces security decisions based on signatures, configurations and contexts (e.g., phone state or location), while security decisions themselves are enforced both at install-time and at runtime. In order to prevent confused deputy attacks, developers have to assign appropriate security policies on each interface, i.e., they have to specify which permissions/configuration/signature the caller is required to have in order to access an interface. However, if the incentives behind these policies protecting the callee conflict with the properties of the calling application, ICC is denied leading to the problem of caller malfunction or crash as explained in Section 7.4. Moreover, since application developers have to define these policies themselves, they might fail to consider all security threats. Finally, Saint does not address malicious developers, who will not deploy Saint policies for the obvious reason that they might want to mount a collusion attack. By contrast, our framework deploys a system-centric solution which is also applied to malicious colluding applications, and enforces its policies at the file-system and network layer as well.

QUIRE [13] is a recent Android security extension which provides a lightweight provenance system to prevent confused deputy attacks via Binder IPC. In order to determine the originator of a security-critical operation, QUIRE tracks and records the IPC call chain, and denies the request if the originating application has not been assigned the corresponding permission. Additionally, QUIRE extends the network module residing in the Android Linux kernel to verify the provenance of remote procedure calls (RPCs) when they aim to leave the device. However, similar to Saint, QUIRE is application-centric-applications forward and propagate the IPC call chain themselves. Due to its application-centric nature, QUIRE cannot prevent colluding applications, because they may drop the IPC call chain and act on their own behalf, and hence, circumvent QUIRE's defense mechanism. Furthermore, the unexpected denial of access by the receiver of the call chain might lead to application dysfunction/crash on the caller's side (cf. Section 7.4).

IPC Inspection [20] is a very recent work that is similar to QUIRE and tackles confused deputy attacks via Binder IPC. IPC Inspection reduces the permissions of an application when it receives a message from a less privileged one. In contrast to our framework, IPC Inspection does not require a policy framework, and hence, can prevent unknown attacks without the deployment of appropriate policies. However, IPC Inspection does not provide a solution against maliciously colluding applications. Although the receiver's permissions are reduced to the sender's permissions, the individual application instances at the receiver's side still reside in one sandbox and thus are not properly isolated from each other and can communicate freely. Moreover, IPC Inspection will induce a significant performance overhead, because it requires the maintenance of multiple application instances with different sets of privileges. An open question here is: how is permission reduction 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)? Further, applications executing with an unexpectedly reduced permission set are very likely to crash (cf. Section 7.4). This issue could be tackled over-privileging applications that crash. However, doing so makes the framework incompatible with legacy applications, while in our framework we only require policy adjustment.

TaintDroid [14] is a framework which detects unauthorized leakage of sensitive data. 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 warn 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, TaintDroid mainly addresses data flows, whereas privilege escalation attacks also involve control flows. Its authors mention that tracking the control flow with TaintDroid will likely result in much higher performance penalties. Moreover, since TaintDroid only detects leakage, it would require some policy-based framework on top to distinguish between authorized and malicious data leakage, as done in AppFence

that is discussed below.

AppFence [27] builds upon and extends the current TaintDroid framework by allowing users to transparently enable privacy control mechanisms. In particular, AppFence tackles the usability problem when permissions are revoked from applications, e.g., it provides faked or blank data when applications access content providers they should not be allowed to access. Nevertheless, like TaintDroid, AppFence provides no means to detect privilege escalation attacks beyond data leakage attacks.

SELinux on Android [36] presents a prototype implementation of SELinux on an Android device. Although this work argues for an SELinux-based solution to Android's security vulnerabilities at the kernel level, it does not attempt to provide a solution for the same. Furthermore, the prototype presented in [36] lacks a coordination mechanism between the Linux kernel and the Android middleware. During the course of our work, we realized that such a coordination mechanism is crucial to bridging a semantic gap between the two layers: namely Linux kernel and Android middleware. Our TOMOYO-based solution not only proposes a solution against attacks employing kernel-level channels such as files and sockets, but also enables communication between the kernel and the middleware so that the former can query the latter in case the former does not have sufficient information to take a security decision at runtime.

TrustDroid [9] is a recent security extension to Android that aims to provide domain isolation, typically between a business domain and a private domain. Thus, TrustDroid builds on a pre-determined basis for classifying applications at install time. On the other hand, our work offers a more generic solution against privilege escalation attacks. Since there is no pre-determined basis to classify applications as in the case of TrustDroid, access control in our solution is more dynamic (requiring decisions to be made on the basis of application behaviour e.g., files read/written), and is carried out at run-time.

Orthogonal Security Frameworks. The framework we presented in this paper is built upon former research on operating system security. First, our work relates to stack inspection (e.g., [37]), a security mechanism that enforces access decisions by inspecting the call chain of a request. This is also along the lines of QUIRE [13]; our framework tracks the call chain in a system-centric way. Further, our framework is related to the chinese-wall (CW) security model [7]: If a subject aims to access an object then the CW model grants/deny the access based on what the subject has already accessed in the past. The overall goal is to prevent information flow between objects sharing the same conflict of interest class. Similarly, our framework enforces access decisions on what the IPC caller accessed in the past.

9. Conclusion

In this paper, we address the problem of confused deputy and collusion attacks on Android. We propose the design and implementation of a practical security framework for Android that monitors application communication channels in Android's middleware and in the underlying Linux kernel (namely, IPC, file system, Unix domain and Internet sockets) and ensures that they comply with a system-centric security policy. We propose several adversary models, define confused deputy and collusion attacks based on a graph-based representation of our security framework and analyze typical communication patterns of Android applications by means of a heuristic study. Our design accurately addresses our observations, as it implements a very fine-grained policy enforcement in the system components, which are the primary means for applications to share data. Inspired by QUIRE[13], we integrate Intent tagging techniques into our system (but in a system-centric way) in order to increase the precision of our analysis. Moreover, a novelty of our prototype is the runtime interaction between our security extensions to the Android middleware and TOMOYO Linux, allowing for dynamic runtime policy mapping from the semantically rich middleware to the kernel. Our evaluation results show that our framework is efficient, effective and usable. It can prevent recently published privilege escalation attacks [16, 12, 20, 35], including sophisticated attacks such as Soundcomber [35] launched via covert channels in the Android system components.

For our future work, we plan extensive user tests of our security framework using a large number of applications from the Android market. Furthermore, we aim to continue our current work on performing system-wide ICC call-chain verification at the Binder level. Finally, we would like to examine how we can integrate SELinux into our security framework.

Acknowledgments

The first author has been supported by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n°257243 (TClouds project: `http://www.tclouds-project.eu`).

References

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

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

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

[4] S. Bleikertz and T. Groß. A virtualization assurance language for isolation and deployment. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, 2011.

[5] S. Bleikertz, T. Groß, M. Schunter, and K. Eriksson. Automated information flow analysis of virtualized infrastructures. In *16th European Symposium on Research in Computer Security (ESORICS)*. Springer, 2011.

[6] T. Bradley. DroidDream becomes Android market nightmare. http://www.pcworld.com/businesscenter/article/221247/droiddream_becomes_android_market_nightmare.html, 2011.

[7] D. F. Brewer and M. J. Nash. The Chinese Wall security policy. *IEEE Symposium on Security and Privacy*, 1989.

[8] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, 2011.

[9] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Scalable and lightweight domain isolation on Android. In *Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[10] 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.

[11] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-related policy enforcement for Android. In *13th Information Security Conference (ISC)*, 2010.

[12] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *13th Information Security Conference (ISC)*, 2010.

[13] 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.

[14] 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 *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

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

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

[17] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *16th ACM conference on Computer and communications security (CCS)*, 2009.

[18] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security and Privacy*, 7, 2009.

[19] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *18th ACM Conference on Computer and Communication Security (CCS)*, 2011.

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

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

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

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

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

[25] T. Harada, T. Horie, and K. Tanaka. Task oriented management obviates your onus on Linux. In *Linux Conference*, 2004.

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

[27] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the Droids you're looking for: Retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS)*. ACM, 2011.

[28] C. Marforio, F. Aurélien, and S. Čapkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical Report 724, ETH Zurich, 2011.

[29] National Security Agency. Security-Enhanced Linux. http://www.nsa.gov/research/selinux.

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

[31] Notion Ink. Adam Tablet. http://www.notionink.in/.

[32] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy oriented secure content handling in Android. In *26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[33] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *25th Annual Computer Security Applications Conference (ACSAC)*. IEEE Computer Society, 2009.

[34] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile protection for smartphones. In *26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[35] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *18th Annual Network and Distributed System Security Symposium (NDSS)*, 2011.

[36] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-powered mobile devices using SELinux. *IEEE Security and Privacy*, 8(3), 2010.

[37] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *IEEE Symposium on Security and Privacy*, 1998.

Appendix A List of tested applications

The Android applications that we tested are the following: Android System Info, Angry Birds, Android Scripting Environment, Blast Monkeys, Bluetooth File Transfer, Bubbles, Cheech and Chong, Chess Free, ColorNote, Compass, Contact Adder, Daum Maps, Documents To Go, ES File Explorer, Facebook, Fede Launcher, First Aid, fring, GO Contacts, Google Chrome to Phone, Google Goggles, Google Plus, Google Sky Map, Google Talk, Google Translate, Hello Kitty, Hopstop, HowStuffWorks, Human Body Facts, Jewels, K9 Mail, last.fm, Meebo, Mobile Andrio, Musical Lite, Öffi, Opera Mini, PagesJaunes, Paper Toss, Qik Video, ESPN ScoreCenter, Talking Tom, Task Manager, The Three Stooges, Time2Hunt, Twitter and Urban Dictionary.

Appendix B System Policy

In the following we provide policy rules applied for testing. These rules are grouped into five categories: DefaultRules, BasicRules, AdvancedRules, StrongRules and ExceptionalRules. DefaultProfile includes policy rules of category DefaultRules, BasicProfile includes policy rules of category DefaultRules and BasicRules, AdvancedProfile includes rules of category DefaultRules, BasicRules and AdvancedRules, while StrongProfile includes policy rules of all categories. ExceptionalRules are exceptional cases from more general rules defined in the system.

We informally describe each rule in Table 3 and further provide definition of these rules in the policy language (Figure 8).

Appendix C Communication graphs

Figure 9 illustrates visualization of the ICC based communication among third party apps and the Android system during our user tests. Figure 10 visualizes the file system and socket accesses by third party apps during testing.

Default rules		
(1)	A third party application that has no permission CALL_PHONE can invoke Phone system application only if data transmitted contains android.intent.action.DIAL parameter (that enforces user confirmation)	[16]
(2)	A third party application that has no WAKE_LOCK permission must not be able to invoke DeskClock system application to play an alarm	[20]
(3)	A third party application that has no WAKE_LOCK permission must not be able to invoke Music system application to play music	[20]
(4)	A third party application that has no CHANGE_WIFI_STATE permission must not be able to invoke Settings system application to toggle WiFi state	[20]
(5)	A third party application that has no ACCESS_FINE_LOCATION permission must not be able to invoke Settings system application to toggle GPS location state	[20]
(6)	A third party application that has no BLUETOOTH_ADMIN permission must not be able to invoke Settings system application to toggle Bluetooth state	[20]
Basic rules		
(7)	A third party application that has no SEND_SMS permission must not be able to contact Android Scripting Environment (ASE) application	[12]
Advanced rules		
(8)	A third party application with permission ACCESS_FINE_LOCATION must not communicate to a third party application that has permission INTERNET	
(9)	A third party application that has permission READ_CONTACTS must not communicate to a third party application that has permission INTERNET	
(10)	A third party application that has permission READ_SMS must not communicate to a third party application that has permission INTERNET	
Strong rules		
(11)	A third party application that has permissions RECORD_AUDIO and PHONE_STATE or PROCESS_OUTGOING_CALLS must not directly or indirectly communicate to a third party application with permission INTERNET	[35]
Exceptional rules		
(12)	A third party application is allowed to start a system or a third party applications by sending an Intent, if this Intent does not include any additional information	

Table 3. Policy rules applied during testing

Section types:
A,B: Application sandboxes
L: Path

Section goals:
goal ProtectDialer(deny) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC.direct) ∧ ¬(L.hasActionString(android.internet.action.DIAL)) ∧ A.trustLevel(untrusted) ∧ ¬(A.hasPermission(CALL_PHONE)) ∧ B.trustLevel(trusted) ∧ B.name(com.android.phone)

goal ProtectDeskClock(deny) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC.direct) ∧ L.hasActionString(com.android.deskclock.ALARM_ALERT) ∧ L.hasExtraData(intent.extra.alarm) ∧ A.trustLevel(untrusted) ∧ ¬(A.hasPermission(WAKE_LOCK)) ∧ B.trustLevel(trusted) ∧ B.name(com.android.deskclock)

goal ProtectMusic(deny) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC.direct) ∧ A.trustLevel(untrusted) ∧ ¬(A.hasPermission(WAKE_LOCK)) ∧ B.trustLevel(trusted) ∧ B.name(com.android.music) ∧ B.component(com.android.music.MediaPlaybackService)

goal ProtectSettingsWiFi(deny) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC.direct) ∧ L.hasCategory(Intent.CATEGORY_ALTERNATIVE) ∧ L.hasData(0:0#0) ∧ A.trustLevel(untrusted) ∧ A.hasPermission(CHANGE_WIFI_STATE) ∧ B.trustLevel(trusted) ∧ B.name(com.android.settings) ∧ B.component(com.android.settings.widget.SettingsAppWidgetprovider)

goal ProtectSettingsLocation(deny) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC.direct) ∧ L.hasCategory(Intent.CATEGORY_ALTERNATIVE) ∧ L.hasData(3:3#3) ∧ A.trustLevel(untrusted) ∧ ¬(A.hasPermission(ACCESS_FINE_LOCATION)) ∧ B.trustLevel(trusted) ∧ B.name(com.android.settings) ∧ B.component(com.android.settings.widget.SettingsAppWidgetprovider)

goal ProtectSettingsBluetooth(deny) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC.direct) ∧ L.hasCategory(Intent.CATEGORY_ALTERNATIVE) ∧ L.hasData(4:4#4) ∧ A.trustLevel(untrusted) ∧ ¬(A.hasPermission(BLUETOOTH_ADMIN)) ∧ B.trustLevel(trusted) ∧ B.name(com.android.settings) ∧ B.component(com.android.settings.widget.SettingsAppWidgetprovider)

goal ProtectASE(deny) := L.connects(A,B) ∧ L.type(Internet) ∧ A.trustLevel(untrusted) ∧ ¬(A.hasPermission(SEND_SMS)) ∧ B.trustLevel(untrusted) ∧ B.name(ASE)

goal PreventLocationLeackage(deny) := L.connects(A,B) ∧ L.type(ICC.direct) ∧ A.trustLevel(untrusted) ∧ A.hasPermission(ACCESS_FINE_LOCATION) ∧ B.trustLevel(untrusted) ∧ B.hasPermission(INTERNET)

goal PreventContactsLeackage(deny) := L.connects(A,B) ∧ L.type(ICC.direct) ∧ A.trustLevel(untrusted) ∧ A.hasPermission(READ_CONTACTS) ∧ B.trustLevel(untrusted) ∧ B.hasPermission(INTERNET)

goal PreventSMSLeackage(deny) := L.connects(A,B) ∧ L.type(ICC.direct) ∧ A.trustLevel(untrusted) ∧ A.hasPermission(READ_SMS) ∧ B.trustLevel(untrusted) ∧ B.hasPermission(INTERNET)

goal ProtectCallPrivacy(deny) := L.connects(A,B) ∧ L.type(any) ∧ A.trustLevel(untrusted) ∧ ¬(A.hasPermission(INTERNET) ∧ (A.hasPermission(PHONE_STATE) ∨ A.hasPermission(PROCESS_OUTGOING_CALL)) ∧ B.trustLevel(untrusted) ∧ ¬(B.hasPermission(PHONE_STATE) ∨ B.hasPermission(PROCESS_OUTGOING_CALL)) ∧ B.hasPermission(INTERNET)

goal AllowApplicationLaunch(allow) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC.direct) ∧ L.hasActionString(android.intent.action.MAIN) ∧ L.hasCategory(android.intent.category.LAUNCHER) ∧ A.trustLevel(untrusted) ∧ (B.trustLevel(trusted) ∨ B.trustLevel(untrusted))

Figure 8. Policy rules used for testing expressed in a policy language

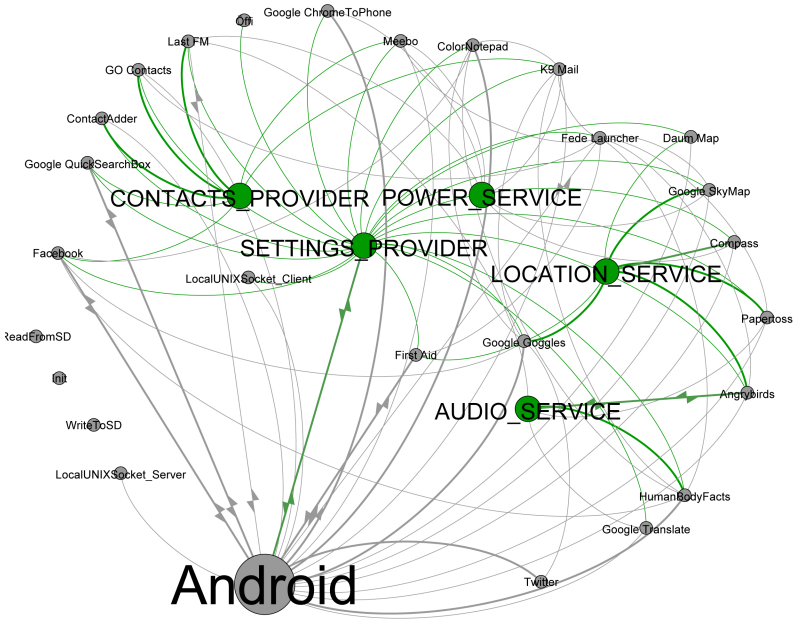

Figure 9. Visualization of the ICC based communication during user tests. Selected System Services and Content Providers are illustrated as separate nodes

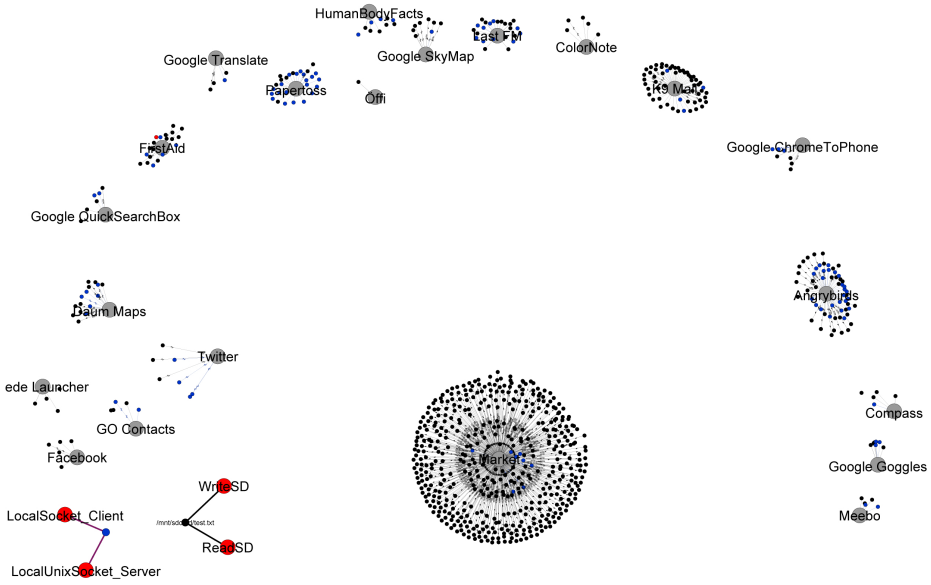

Figure 10. Visualization of file system and socket access by third party apps. Grey nodes represent benign applications, red nodes represent colluding applications, black nodes represent files and blue nodes represent sockets. On the bottom left are two examples of colluding applications: one pair of apps (WriteSD and ReadSD) collude by means of a shared file on the SDcard, while another pair (LocalSocket_Client and LocalUnixSocket_Server) collude using a socket in a client-server model