Flexible Data-Driven Security for Android

Denis Feth
Fraunhofer Institute for Experimental Software Engineering IESE
Kaiserslautern, Germany
denis.feth@iese.fraunhofer.de

Alexander Pretschner
Karlsruhe Institute of Technology
Karlsruhe, Germany
pretschner@kit.edu

***Abstract*—Android allows users to cancel the installation of apps whenever requested permissions to resources seem inappropriate from their point of view. Since permissions can neither be granted individually nor changed after installation, this results in rather coarse, and often too liberal, access rules. We propose a more fine-grained security system beyond the standard permission system. With our system, it is possible to enforce complex policies that are built on temporal, cardinality, and spatial conditions ("notify if data is used after thirty days", "blur data outside company's premises", etc.). Enforcement can be done by means of modification or inhibition of certain events and the execution of additional actions. Leveraging recent advances in information flow tracking technology, our policies can also pertain to *data* rather than *single representations* of that data. For instance, we can prohibit a movie from being played more than twice even if several copies have been created. We present design and implementation of the system and provide a security and performance analysis.**

***Keywords*-Security, Android, Access Control, Usage Control, Information Flow.**

I. Introduction

With a market share of 52.5% [10] Android is the most popular smart phone platform. It is characterized by open concepts and its extensibility and individualization opportunities provided by a high number of low cost apps. On the downside, security becomes an issue. Many apps expose private data over the network, as recent studies have shown [7], [29]. Because third-party applications are untrusted per se, Android provides security mechanisms to protect the system and its data. By default, apps are sandboxed and therefore isolated from other system parts. They are not allowed to access data or functionalities outside the boundaries defined by the sandbox. At development time, developers *must* specify required permissions to perform desired actions (e.g., access to GPS or contact information) in a so-called manifest file. Prior to the installation of the app, the user has to accept this manifest.

While this approach guarantees a certain level of security, it lacks flexibility in practice. Permissions are assigned at installation time and cannot be changed afterwards. Also, it is not possible for a user to grant certain permissions while denying others—the entire manifest has to be accepted or rejected. In the latter case, the app cannot be installed. More importantly, users have no control over the runtime behavior of applications. Even if the requested permissions seem reasonable from the user's point of view, the app might abuse its rights (for example by sending private data to an advertisement server). The user has no possibility to restrict the usage of data once he has granted permissions.

Hence, the first attacker model we consider is that of malicious apps. As we will see, the security system that we describe also caters for a second attacker model, namely that of malicious users. In this case, it is not the phone's user who stipulates constraints on the use of resources but rather app providers or data owners. This is relevant in the context of digital rights management but also in contexts where, for instance, an employer provides phones to the employees but wants to make sure specific functionalities are not used on the premises of the company.

In this paper, we present a fine-grained security system that also encompasses data-driven usage control, a generalization of access control to the time after data has been accessed. Our objective is to enhance Android security in a way that the usage of resources can be observed and controlled through the specification of explicit, fine-grained security policies. To this end, we add a reference monitor to Android that can observe and control events that pertain to the behavior of applications and the usage of resources. In addition, our reference monitor is data-driven. This means that we can control not only one specific representation of some data (e.g., "play file movie.mpg at most twice"), but rather all representations of that data ("play the movie initially contained in file movie.mpg at most twice"). By the integration of these data-driven usage control concepts, data or resource usage can flexibly be restricted. Users can now accept manifests they wouldn't have accepted before since more detailed security requirements, defined by the users, can be enforced at runtime.

Example Scenarios. For illustration purposes, we will use the following scenarios that cannot immediately be enforced in the standard implementation of Android.

Imagine a malicious client app for a social network. At installation time, the user will be asked for several

Supported by Google's AndroidEDU program as well as by the Fraunhofer-Gesellschaft internal program Attract 692166.

permissions. For social networking apps, this usually includes Internet and location access, read and write access to contacts, sending text messages to friends and so on. However, once the user accepts the permissions, he has no control how these permissions will be used. The user might be interested in restricting the app by policies like “App X can track my position at most once per hour”, “App X is not allowed to send more than two text messages per day” or “App X can only access my favorite contacts”.

We have seen that besides malicious apps, malicious users are of interest as well. Imagine a company that provides smart phones to its employees. Currently, the company has no possibility to restrict the usage of this device. From their point of view, it would make sense to apply policies like “If at work, the social networking app X cannot be used” or “Pictures taken on the company’s premises, *including any copy of these pictures*, can only be displayed blurred after the employee left the building and must not be uploaded by the social network app”.

Problem. In a nutshell, the problem that we tackle is the lack of flexibility of Android’s security system which is based on rather coarse access permissions and requires users to either accept or reject complete manifests at installation time. Proposed extensions make it possible to specify complex conditions on permissions or certain events but do not allow to specify security policies for *all* representations of a data item; others focus on *tracking* these representations. Other extensions limit the security system’s reactions in terms of security violations to inhibition of the respective event, whereas the mere modification of this event (e.g., blurring) may be less impeding to the user.

Solution. Our solution consists of a reference monitor at the application framework level that monitors permission checks, intents, and queries to content providers, as well as some selected data sinks. For tracking data flows, it uses the TaintDroid [7] system. By means of security policies, the monitor can be configured at runtime and check complex conditions with temporal, spatial, and cardinality constraints. Policies are enforced by modifying or inhibiting the trigger events, or by executing further actions.

Contribution. As far as we know, no implementation of data-driven reference monitors for complex conditions exists for Android (cf. Section III).

Organization. We present some background in Section II. We put our work in context in Section III. We then present the design and implementation of our system in Section IV. In Section V, we evaluate security and performance and discuss our system. We conclude in Section VI.

II. Background

Android—Architecture. The Android software stack consists of several layers. On the operating system level, a Linux kernel provides hardware abstractions and other basic functionalities needed by the upper layers like process, memory and power management, networking and basic security mechanisms. The Android runtime system is defined by the Dalvik Virtual Machine (DVM), a Java virtual machine for Android’s bytecode format ‘dex’. Hence, all Android applications are written in the Java programming language. As a middleware, the application framework builds the core of the Android system. It provides both APIs for application developers and several core services, including managers for accessing the location (LocationManager) or resources (ResourceManager, ContentProviders), controlling the application lifecycle (ActivityManager) and many more. Finally, Android comes with a set of core applications including phone, launcher or the home screen. Apps are composed of activities (UI screens), background services, receivers for broadcast messages and content providers. The former three component types are invoked via so-called intents. Content providers are used to share data between apps.

Android—Security. As apps are usually provided by third-party developers, it is important to protect the user and the system from malicious apps. On the operating system level, basic security mechanisms are provided by the Linux kernel. Every installed app has its own user ID assigned, so that access to the file system and privileged operations can be controlled. This user ID stays constant until the application is uninstalled. Further security is achieved by running each app in a separate instance of the DVM and therefore in its own process. In this way, apps are isolated from other apps, i.e., sandboxed. To perform actions outside the boundary defined by the sandbox, Android uses a permission system that regulates the access to resources. At development time, the author has to explicitly declare required permissions in the app’s manifest file. Once an app has been installed, these permissions cannot be changed. This is the starting point of the work presented in this paper.

Permissions are checked by the package manager in the application framework. Since every app runs in its own process, calls to other apps or the or the system process are remote calls. Those calls are realized by Binder, the inter-process communication mechanism of Android, which provides the possibility to securely retrieve the identity (UID and PID) of the calling process. This information it is then used to check whether the calling process has a specific permission.

As a distinguishing technical feature (cf. Section III), the work in this paper controls all introduced concepts: intents, queries to content providers, and permission checks.

Information Flow. Information, or data flow analyses, determine the flow of data through a system. This can be done both statically [11], [28] and dynamically [5], [20]. Flows can be explicit and implicit. Explicit flow is given via a chain of usages of and assignments to memory locations. Implicit flows occur if a condition is evaluated over a secret variable. In this case, information flows from the secret variable to all memory locations that are potentially assigned values in both the executed and the non-executed branches. Because the measurement of implicit flows is often impracticable in practice (because *some piece* of information almost always flows), in this paper, we make use of TaintDroid [7] that exclusively caters for explicit flows. While TaintDroid is about observing data flows until a specified sink is reached, our work allows it to specify complex conditions in the form of data-driven security policies.

III. Related Work

Overviews of Android security have recently been provided, including some criticism [27], [8], also of permission-based systems in general with a focus on Android [3].

Beresford et al. implemented MockDroid [4], a modified Android system that allows users to deny certain permissions after installation. It returns fake values if an app tries to access an API that has been protected in this way. TISSA [31] follows a similar approach. Selected "privacy-aware components" (such as the contacts provider) were modified to perform an additional permission check with the TISSA system and return faked values if desired. The APEX framework proposed by Nauman and Khan [19] enables the user to define runtime constraints for the permission system. In contrast to MockDroid, permissions are denied by the package manager if the policy is violated. This has the advantage that it is not necessary to modify each single security-related API. Ongtang et al. propose the Saint system [22] that extends the Android security system to application-level runtime checks of permissions that take into account the permissions of both caller and callee. This allows developers to protect interfaces in a more flexible way. Our work differs from these approaches (1) in that we not only cater for permissions or selected APIs, but also to intents and content provider queries (cf. Section IV), (2) in that we can specify various actions to be taken (inhibit, modify, execute), and (3) in that our solution makes it possible to enforce policies defined on complex conditions and the choice between specific and all representations of a data item via data flow detection.

Possibly closest to our work is the ConUCON system [2] which also allows for the specifiction of complex conditions before and during access to data. Our work differs in that (1) we can stipulate modifications of trigger events and executions of further actions in addition to blocking requests and in that (2) we cater to data flows and thus the possibility to specify representation-independent policies.

Porscha [21] enforces DRM policies specified in an extension of the OMA REL policy language, including location-based policies like ours. By the use of identity-based encryption and the binding of policies to data (e.g., to SMS), Porscha can control which applications can access protected data. Indirect receivers of protected data are covered by Porscha as well. Our work differs in that (1) we can express more powerful policies over cardinality, temporal, and spatial operators [14]; (2) we support the modification of trigger events as possible action, and (3) in that we cater to data flows. On the other hand, we do not consider protecting data in transit and do not bind the policy to the data item.

Portokalidis et al. propose to defer access decisions to a server [24], which is complementary to our work. Our decision point can be deployed both on the phone and remotely.

Our work is closely related to usage control which extends access control to what happens to data once access has been granted [23], [26]. Typical technologies for implementing enforcement mechanisms include ad-hoc solutions, runtime verification, and complex event processing. In this paper, we use runtime verification technology to automatically synthesize the monitors from policies. Enforcement has been studied for many different layers of abstraction, including the operating system OpenBSD [12], the X11 level [25], Java [15], the .NET CIL [6], machine languages [9], [30], social networks [16] and in the context of digital rights management [1], [18]. The reason for this variety is that *data* that has to be protected comes in different *representations*: as network packets, as attributes in an object, as window content, etc. Our work differs from all the cited references in that we specialize on the Android platform and in that we combine the enforcement of complex usage control requirements with the possibility to distinguish between one specific or all representations of a data item.

IV. Design

A. Layers of Enforcement

Flexible security checks can be implemented at different system layers. The choice of the level considerably influences the architecture and the kind of policies that can be enforced. For our purposes, enforcement at the application framework level appears as the best solution. It serves as middleware, provides hardware abstraction and system services for multiple purposes. Therefore it is the ideal place to monitor events and to enforce security policies. Other possibilities like system call interposition, hooking into the DVM interpreter or static byte code manipulation were rejected since they are too low-level for our purpose and, given the higher number of events to be monitored at these layers, are likely to yield significantly higher performance overheads (cf. Section V-B). Furthermore, the level of the application framework is the only option that allows the proper use of TaintDroid for data flow tracking.

We monitor four types of events: permission checks, queries to content providers, intents and certain data sinks like the network, file system and IPC.

By monitoring **permission requests**, the current permission framework gets considerably more flexible. With our system, it is possible to control permissions given to apps on the grounds of complex conditions (see below) that are checked at runtime rather than at deployment time. For example, the user can choose to give a certain app the permission to access the Internet only once per hour. Once the permission is granted by our reference monitor, the decision is then up to the package manager, the normal decision point.

Queries to **content providers** are monitored to protect private data. Content providers encapsulate an application's data, for instance, contact data of the contact app. The author of a content provider can achieve basic access control by defining permissions which are necessary to access its data. However, it is not possible for the user to limit access based on self-defined rules. By monitoring queries to content providers—that syntactically resemble SQL queries—it can directly be controlled which data is read or written. For example, an app could be limited so that it only has access to favorite contacts.

Finally, **intents** are monitored. They are managed by the activity manager, which is part of the application framework and handles the lifecycle of apps. Via intents, activities, services and broadcast receivers are activated within and in-between applications. By monitoring them, the interaction between activities and services as well as broadcast messages can be controlled.

B. Information Flow Tracking

An important part of controlling data usage is the tracking of information flow. By the consideration of information flow, the user is able to specify policies about how data might flow through the system and to protect all representations, or copies, of a data item. To trace information flow, we use TaintDroid [7] which provides a framework for information flow tracking on Android. It works with taint tags that are persistently added to data at data sources. Whenever this data is copied, the taint tag is copied along with the data. With TaintDroid, it is hence possible to track data flows through the system and to consider specific representations of data. For example, it is possible to add a special taint marking to a picture that has been taken at a specific location and prevent any usage of this picture by applications. Since the taint tag is persistently added to the picture, all (also non-verbatim) copies of this picture would be covered by this policy. Instead of declaring "DSC0123.jpg cannot be used by app X", the policy would be "The picture with taint marking XY cannot be used by app X" Currently, TaintDroid supports 32 individual taint markings which can be combined as desired. We monitor information flow by checking taints tags passed in intents and the taint tags of data that is read or written via content provider queries. Additionally, events are triggered if tainted data is read from or written to the file system or if it passes the network sink. Note that our work extends TaintDroid in that TaintDroid *detects* data flows whereas our framework makes it possible to *control* data flows with respect to user-defined policies.

C. Policy Language

Policies consist of two parts: A set of event declarations (specifying events that may occur in a concrete system) and a set of mechanism descriptions, consisting of preventive and detective event-condition-action (or ECA) rules. Preventive mechanisms can block or modify the event, while detective mechanisms can only observe that an event happened under the condition specified in the rule. Both strategies can execute additional actions. Listing 1 provides the abstract syntax where $\mathbb{S}$ denotes strings and $\mathbb{N}$ denotes natural numbers.

Listing 1: Abstract Policy Syntax

```
Policy::= eventDeclaration, {PreventiveMechanism | DetectiveMechanism}+;
PreventiveMechanism::= Event, Condition, AuthorizationAction,
                       {ExecuteAction};
DetectiveMechanism::= Event, Condition, {ExecuteAction};
Event::= actionName, {paramMatch | Location};
Location::= latitude, longitude, tolerance, defaultValue, [negate];
Condition ::= PL | TL;
PL ::= true | false | xPathEval(𝕊) | eventMatch |
       not(PL) | and(PL,PL) | or(PL,PL) | implies(PL,PL);
TL ::= PL | not(TL) | and(TL,TL) | or (TL,TL) | implies(TL,TL) |
       since(TL,PL) | always(TL) | before(ℕ,TL) | during(ℕ,TL) |
       within(ℕ,TL) | replim(ℕ,ℕ,ℕ,PL) | repmax(PL,ℕ) | repsince(PL,ℕ,TL);
AuthorizationAction::= allow | inhibit | {Modifier};
Modifier::= paramName, (value |  taint(ℕ) | replace(𝕊,𝕊) | append(𝕊) |
            blur(ℕ) | delete(𝕊) | add(𝕊));
ExecuteAction::= log | notify | startActivity | mockLocation;
```

To check if the trigger event (which can be any event happening in the system) is relevant for a particular policy, the event is first matched against the parameters specified in the policy. Besides that, it is possible to check for taint markings, in order to apply the policy to any representation of a data item that is to be protected. Additionally, a location condition can be added to the trigger. It consists of coordinates (latitude, longitude), a tolerance in meters and a default value that will be used if the location cannot be accessed (e.g. due to a bad signal). If the last optional flag is set to true, the evaluation will be negated. This allows us to define a trigger which matches if we are *not* at the given location.

The condition part is expressed in past temporal logic that can be formulated by the use of logical, temporal, cardinal and event matching operators, as well as with XPath expressions. Because of space restrictions, we do not describe the language semantics for expressing conditions in detail [14], [17] but rather give the intuition only: since(a,b) is true if b has been true ever since a happened; the always operator is intuitive, before(n,a) is true if a was true n time steps ago; within and during are intuitive. The cardinal operator repmax(n,a) specifies that a has been true at most n times in the past; replim(l,m,n,a) specifies a lower (l) and upper limit (m) of allowed events a in the last n timesteps; and repsince(n,a,b) specifies that a has been true at most n times since b became true.

The action part describes if the event is allowed or if it has to be modified or blocked (only for preventive mechanisms) and possibly contains additional actions. For the modification of event parameters, we can use a set of standard operations (such as replace or append), and of course a complete replacement of the parameter value. It is also possible to add taint markings to data. The current implementation allows the additional execution of four action types, namely logging, user notification, starting an activity and faking the current location. In practice, policies are specified in a concrete XML syntax, as shown in Listing 2.

D. Architecture

Our high level architecture (Figure 1) consists of two components: a reference monitor in the Android system and the security manager app.

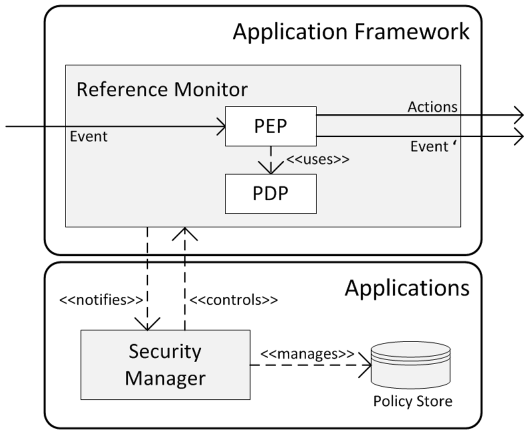

Figure 1: Conceptual view

The reference monitor is in charge of enforcing the ECA rules specified in the deployed policies. It consists of two components, namely the PEP and the PDP. The Policy Enforcement Point (PEP) receives events and enforces the policy by allowing, modifying or inhibiting them. It may also perform additional actions, as described above. The PEP is running as a system service, which is advantageous because system services have all permissions by default as they are running in the system process. Thus, all resources of the system can be accessed and used unrestrictedly (besides some smaller exceptions like SD card access).

The decision of whether or not an event is allowed is taken by the Policy Decision Point (PDP). It decides about the admission of incoming events based on the condition part of the ECA rule. The PDP is automatically synthesized from the policy condition [13]. We chose to run the PDP directly on the device rather than remotely. If necessary, the PDP could also run as a web service, with a tradeoff between global policy enforcement, availability requirements and expected performance drawbacks. For now, the PDP is a system service just as the PEP. However, the implementation itself is done natively. We chose to do so because in this way, we were able to re-use existing PDP synthesis implementations [17].

At runtime, the interaction with the monitor works as follows. The hooked component (e.g. the activity manager) synchronously sends the event to the PEP via Binder. According to the type of the event, the PEP transforms the event into a format that can be understood by the PDP. Via Binder, the PEP sends the event to the PDP, which decides whether the event is allowed or not and if modifications or additional actions are required. Based on this decision, the PEP will now execute desired actions or modifications. The hook will then replace the original event with the (possibly modified) event that has been returned by the monitor.

The Security-Manager app has two main purposes. First, it provides a user interface for our system. Regarding this, the user can deploy policies and change several settings of the monitor (such as status, logging level, etc.). Also, it is in charge of notifying the user about policy violations, if specified in the deployed policies. Second, the Security-Manager implements the policy store. The user can choose to import policies from the SD card or from a webserver or to create simple policies directly on the device with the built-in policy editor.

E. Scenarios Revisited

In Section I we introduced several scenarios. We now show how to define respective policies and explain how these policies are enforced in our system. Listing 2 shows some of the mechanisms for the scenarios mentioned in the beginning.

The policy contains four preventive mechanisms. The first mechanism deals with the first scenario and assures that app X with UID 10052 cannot send more than two text messages (SMS) per day. Each time the app tries to send an SMS an event is triggered and matched against the specified parameters. If the event matches, the condition is evaluated. If it evaluates to true—in this case, if at least two text messages have been sent during the day, and another attempt (trigger) to send a text message occurs—the user is notified and the permission will be denied. This is implemented in Android’s package manager, where a SecurityException is thrown.

The other mechanisms are used to implement the second scenario. To prevent employees from using a certain app at work, the policy contains a mechanism that blocks all intents that aim to start an activity of the app in question (e.g., all intents targeting a package starting with “com.socialnetwork”). In order to block the events only at the company, we added location information to the policy.

To blur pictures that have been taken at a company, as well as all copies of these pictures, we need two mechanisms. First, we have to taint pictures if they have been taken at the company and second, we have to blur read operations on this file outside the company. Accordingly, the third mechanism assures that pictures are tainted with a special taint marking (0x10000 = 65536) whenever they are taken

Listing 2: Example Policy (excerpt)

```xml
<preventiveMechanism name="LimitTextMsg">
 <description>App must not send more than two text msg a day</description>
 <trigger action="permission:check" />
 <condition>
  <not>
   <repLim lowerLimit="0" upperLimit="1" amount="1" unit="DAYS" >
    <eventMatch action="permission:check">
     <paramMatch name="perm" value="android.permission.SEND_SMS" />
     <paramMatch name="uid" value="10052" />
    </eventMatch>
   </repLim>
  </not>
 </condition>
 <authorizationAction>
  <inhibit />
 </authorizationAction>
 <action name="notify">
  <parameter name="msg" value="App tried to send more than 2 msg." />
 </action>
</preventiveMechanism>

<preventiveMechanism name="BlockSocialNetwork">
 <description>The app cannot be used at the given location</description>
 <trigger action="intent:startActivity">
  <paramMatch name="location" value="49.430086;7.753326;50;true;false"/>
 </trigger>
 <condition>
  <xPathEval>
   starts-with(//event/parameter[@name='component']/@value, 'com.socialnetwork')
  </xPathEval>
 </condition>
 <authorizationAction>
  <inhibit />
 </authorizationAction>
</preventiveMechanism>

<preventiveMechanism name="TaintPictures">
 <description> Taint pictures taken at given location </description>
 <trigger action="dataflow:write">
  <paramMatch name="taint" value="128" />
  <paramMatch name="location" value="49.445626;7.760339;50;true;false"/>
 </trigger>
 <condition><true /></condition>
 <authorizationAction>
  <allow>
   <modify>
    <parameter name="data" value="taint$65536" />
   </modify>
  </allow>
 </authorizationAction>
</preventiveMechanism>

<preventiveMechanism name="BlurPictures">
 <description>
   When not at the given location, files with taint marking 0x10000 = 65536 are
          blurred upon access and the user is notified.
 </description>
 <trigger action="dataflow:read">
  <paramMatch name="taint" value="65536" />
  <paramMatch name="location" value="49.445626;7.760339;50;true;true" />
 </trigger>
 <condition> <true /> </condition>
 <authorizationAction>
  <allow>
   <modify>
    <parameter name="data" value="blur$5" />
   </modify>
  </allow>
 </authorizationAction>
 <action name="notify">
  <parameter name="msg" value="A tainted picture has been blurred." />
 </action>
</preventiveMechanism>
```

at the given location. The trigger condition is explained by the fact that data from the camera is identified by the taint marking 0x00080 = 128 that TaintDroid adds to pictures taken by the camera. Subsequently, the TaintDroid system will make sure that any copy of the picture will be tainted with the marking 0x10080. The fourth mechanism observes read operations on this kind of tainted pictures, and blurs the picture if the phone is outside a diameter of 50 meters of that location. Note that the file itself is not modified but only the output stream of the reading operation. This allows us to blur the file content while keeping the file itself unchanged. Because the default value of the location trigger is set to true, the monitor will assume that the trigger matches in case it cannot access the location.

To assure that the evaluation of the location cannot be manipulated, an additional mechanism is needed to prevent spoofed locations (not shown in the listing). To realize this, the ACCESS_MOCK_LOCATION permission has to be denied for all apps. Also, a default value has to be provided that is used whenever the location is not accessible.

V. Evaluation

A. Security

The goal of this evaluation is to analyze our security system to find potential vulnerabilities and possibilities to exploit them. We perform the analysis from an attacker's point of view, analyzing different attack vectors and possible benefits. In our case, we have two different attacker models. First, we have to consider malicious apps, such as spyware. Second, the user himself might also be interested in circumventing the reference monitor like in our sample scenario. In the following, we point out different attack vectors and analyze them with respect to their practicability. Unless otherwise stated, our discussion is valid for both attacker models. The analysis will be made under the following assumptions:

1) The phone is not rooted. This is a necessary assumption, because system files (including our system) can be modified on a rooted phone. However, as rooting would break the whole Android security concept anyway, this can be considered a reasonable assumption.
2) Android itself is free of vulnerabilities. We need this assumption in order to confine our discussion.

Man-in-the-Middle Attack. One way to attack the monitor is to intercept messages to (the reported event) and from (the possibly modified event) the monitor. There are three kinds of messages an attacker might be interested in, namely the communication with the reference monitor, the communication between the PEP and PDP, and the communication with the native PDP library. All communication with the monitor and between the services is done via Binder IPC. The communication with the native library is done via JNI. Both concepts are well approved and heavily used in Android. Following assumption 2, it can be assumed that these mechanisms are secure and free of vulnerabilities.

Denial-of-Service Attack. An attacker could also try to attack the availability of our system, which could be realized in several ways.

First, the monitor process could simply be killed. However, this is not possible, since the monitor runs in the system process and the termination of the system process would cause a reboot of the device.

Second, the monitor can be attacked with malicious input. This can be realized either by a flooding attack or with malformed messages. In the first case, the attacker aims to slow down or crash the monitor by producing artificial events which allocate resources and have to be processed by the monitor. However, we use a synchronous call model. This means that the system might slow down, but the attacker cannot gain benefit from it. In case of a malformed message, the monitor falls into a safe state and inhibits the event if an exception occurs on the Java level. If the monitor crashes on the C level (e.g., due to a segmentation fault) this will indeed cause a crash of the system process. In this case, the device reboots, which means that the attacker cannot gain any benefit.

Third, the native PDP implementation could be overwritten by prepending a malicious implementation. However, this scenario cannot be exploited if the device is not rooted (cf. assumption 1).

Fourth, since the library (libpdp) is prelinked, a return-to-libpdp attack, similar to a return-to-libc attack, might also be possible. To execute this attack, a buffer-overflow-attack is necessary, contradicting assumption 2 (presumed we did not introduce new vulnerabilities).

Fifth, the attacker could try to disable the monitor via the service's interface. However, the identity of the calling process is checked in advance. This means that the monitor can only be disabled if the call is coming from the Security-Manager. Following assumption 2, it is not possible for apps to forge their identity. Also, it is not possible to install a malicious Security-Manager with the name package name on a non-rooted device. To protect our system against unauthorized access (e.g., if the handset user is untrusted), the Security-Manager is password protected.

Finally, a DoS attack targeting the Security-Manager might also be beneficial for an attacker if he tries to hide notifications about policy violations. For this reason, notifications are persistently stored by the app so that they are not lost if the app is killed. If a new notification is pending, the app's broadcast listener will handle the notification, regardless of whether the app has been killed before. The second option—the installation of a malicious Security-Manager with the same package name—has been discussed above and shown to be not feasible on a non-rooted device. Attacks based on malicious messages (flooding as well as malformed messages) are not feasible either. The Security-Manager automatically rejects messages that do not originate from the system process.

Attacking Policy Evaluation. By modifying policies, the attacker could misuse the monitor in any desired way. In general, this is possible in the policy store before the policy got deployed or in the PDP after the policy deployment. The former possibility is prevented by the access control mechanisms provided by the Linux kernel as policies are centrally stored read-only in the private data folder of the Security-Manager. Manipulating the policies after they got installed is possible by deploying an own policy that overrides the previous one. However, this is not possible since the service interfaces are protected (see above).

In addition, a direct manipulation of the internal state of the PDP could be used to influence the evaluation. However, access to memory of other processes is not possible since the Linux kernel provides corresponding memory protection mechanisms. Illegal modification of internal states is usually achieved by vulnerability based attacks (such as buffer-overflow-attacks) which are covered by assumption 2.

Besides that, an attacker might manipulate external conditions that the PDP uses for the policy evaluation, namely time, location and taint tags. Time is supposed to be used relatively and via discrete time steps, so that an attacker cannot manipulate the evaluation in this way (e.g., by changing the system time). However, in the current implementation we frequently access the physical system time for evaluation. Fortunately, this poses no problem for the first attacker model, as user apps cannot directly change the system time on a non-rooted device. If we consider the second attacker model where the user is the attacker, we either have to change our current implementation or prevent the user from modifying the system time in the preferences. We currently work on that issue. Mock locations can be used to forge the device's position to circumvent the evaluation. If a location-based policy is installed, we hence remove existing Test-LocationProviders (location providers that provide a mock location). Additionally, the policy must contain a mechanism that rejects the ACCESS_MOCK_LOCATION permission for all applications. Since the position may not be accessible, e.g., due to a bad signal or disabled location services, the policy has to provide a default value for this case. In terms of information flow, there are some limitations regarding tagged files. Currently, taint tags are lost if files are read or written directly via the file system's address, which is a limitation of the current TaintDroid implementation. Also, a malicious user can access files on the SD card by simply inserting it in a desktop computer or connecting as USB mass storage. This could be prevented through an encrypted file system (which is supported on newer Android version) and by monitoring information flow over USB. However, both solutions are not implemented in our current proof-of-concept.

B. Performance

In the following sections, we describe the setup and the results of the performance evaluation of the reference monitor. We consider four aspects: overhead for monitored operations; influence on the overall system performance; performance-critical parts in the monitor; memory usage.

Influencing Factors. Performance is impacted by several influencing factors. First, devices differ in the integrated hardware which influences the performance not only of the monitor but of the whole system. Second, the number and usage of installed apps influence the performance. Simply speaking, the more happens on the device, the more events will be produced that cause a higher load on the monitor. Finally, a considerable influence stems from the deployed policies. If policies are more sophisticated (for example if they contain complex XPath expressions and time based operators), or there are many mechanisms installed, this will have a negative influence on the performance, simply because the evaluation takes more time.

Test Setup. For the evaluation, we used a Google Nexus S which is a middle class device, regarding its hardware capabilities. We randomly generated several thousand events (sending broadcast messages, read all contact data, checking a random permission for a random app, read a tainted file) and measured the time overhead for each operation. Additionally, we logged all events that occurred in a 65 hour time frame to get an idea about the total number and types of events in a realistic end-user environment.

For the identification of performance critical sections, we analyzed the runtime behavior of the reference monitor in order to get an impression of the (relative) time spent for several tasks like serialization and evaluation of the events and the enforcement of the policy.

We used a policy that covers all event types: (1) a specific app cannot be used at a specific location; (2) pictures that were taken at a specific location cannot be redistributed over the network and can only be displayed blurred outside that location; (3) only starred (favorite) contacts are displayed; (4) no app can access the location.

Results. First, we analyzed how the deployment of policies influences the overall performance. As a reference point, we used an unmodified Android system and an unmodified TaintDroid, respectively. For the evaluation, we randomly sent broadcast intents, checked certain permissions, read the contact database and a tainted sample file and called a remote test service via Binder IPC. Compared to an unmodified Android system, we found a computational overhead between factor two for more complex events (e.g., reading address book) and factor four for rather simple events (e.g., permission check). For simple events, the relative overhead is higher as they have a short runtime by default. Compared to TaintDroid, we found an overhead between factor 1.5 and 2.5. To evaluate the overall influence on performance, we performed a 65 hour test run under normal operating conditions. We intercepted 28,700 event, which is less than 8 events per minute. From a users point of view, the resulting overhead is, in realistic end-user setting, barely noticeable as the observed, absolute overhead was less than 50ms in average.

Second, we analyzed the relative runtime of certain tasks in the monitor (micro benchmark) in order to identify performance killers using method profiling. Although method profiling cannot be used to get realistic results about the absolute runtime, it is a good approach to compare the relative runtime distribution of subroutines and tasks performed by the monitored method. Based on the results, we changed the the XML-based communication we used in the first implementation, as XML processing consumed 38% of the total time spent in the monitor. We achieved an overall performance improvement of factor two.

In terms of RAM, overhead can hardly be measured since the reference monitor is a part of the system process. By observing the memory allocation (PSS), we estimate an overhead up to 10 percent for this process, depending on the deployed policies and event frequencies.

C. Discussion

In this section we discuss several design decisions and limitations of the current approach or implementation.

Design Decisions. We chose to add the reference monitor on a rather high level, namely at the application framework level. This is a fundamental decision which influences not only the implementation of the monitor, but also the monitored events and the policies that can be enforced. We chose it because for our purposes, this level is the most convenient, given that the application framework is responsible for the interaction of components (via intents), implements the permission system and provides APIs that can be used by the applications (for example to access content providers). Nevertheless, policy enforcement on other abstraction levels would also be beneficial, for example on the Dalvik interpreter or system call level. We rejected these options because of performance aspects and the fact that policies over high-level events (e.g., "granting a permission") are not possible at these abstraction layers.

As we chose TaintDroid for information flow tracking, we have to consider the limitations of the current TaintDroid implementation. From our point of view, the most critical limitation is the loss of taint tags on direct read or write operations of files. The other way around, TaintDroid produces false positives, among other things because taint tags are stored per array or IPC message. We did not evaluate the frequency or the impact to our solution, but have to keep this in mind for later improvements. Another TaintDroid-related problem is that the usage of tainted files (e.g., on the SD card) is not restricted on non-TaintDroid devices, such as desktop computers. This poses a problem for our second attacker model (malicious users), as users can just mount the SD card on an external device. A possible solution would be the encryption of the file system, which is supported on never Android versions.

Assumptions. Our security evaluation assumes a non-rooted and vulnerability-free device. Both assumptions are necessary to assure that system files are not tampered with. However, in practice both assumptions are questionable. Rooting an Android device is quite simple, even for unexperienced users. Also, vulnerability reports are quite frequent. A solution for this problem would be the use of a trusted computing platform module (TPM) which ensures that we are running an unmodified version of the system.

Practicability. Our—and likely any comparable—system may introduce problems it set out to resolve in the first place.

If a user decides, for privacy reasons, to disable location services, then a policy with location conditions will fall back to a default value that might cause the inhibition of certain events. This seems to be a general problem of implementing security policies rather than of our particular approach.

At runtime, policy enforcement might cause instability of apps, for example if permissions are denied (including ACCESS_MOCK_LOCATION which will be denied if location-based policies are deployed). To face this issue, app developers would have to consider permission denials by catching potential security exceptions, which is rarely done currently. An alternative would be a deep modification of the permission framework, which is certainly not desirable. The inhibition of the other event types should have no direct effect on the stability of the apps, but it may change their runtime behavior.

Usability is always a concern in specifying security policies. On the upside, the expressiveness of our language allows a user to clearly specify requirements that distinguish between different situations. For instance, we can concisely specify and enforce that location data may be sent to the network but not to an ad service without prompting the user every time location data is about to be sent. On the downside, the correct and adequate use of complex security policies like those we can enforce with our system likely is hard for non-expert users (this likely is more of a challenge when considering the attacker model of malicious apps rather than that of malicious users).

Several challenges coincide: understanding risk and security requirements; formalizing requirements in a language that does not hide its roots in temporal logic; properly understanding the concepts of data flow as measured by TaintDroid. This problem clearly transcends the research described in this paper.

VI. Summary and Conclusions

We have extended Android Gingerbread 2.3.4 by flexible data-driven access and usage control mechanisms. We can express and enforce fine-grained policies with temporal, spatial, and cardinal conditions that refer to both single representations of data and, via taint tags, to all representations of a data item. Our system helps defend against two attacker models: malicious apps and malicious users. In the former case, the phone's user can stipulate policies for protecting against malicious apps (e.g., "Send at most two SMS per hour", or "Do not send any copy of my contacts"). In the latter case, an app or data provider can stipulate constraints on the usage of resources or data (e.g., "Play any copy of a movie at most twice"). In terms of enforcement, we chose the application framework level as a basis because user-relevant events happen at this level, hooks can be placed precisely, and TaintDroid can be used for information flow tracking. The monitor itself is implemented by two independent system services that run in the system process, the enforcement part (PEP) and the decision part (PDP). This has the advantage that the monitor is accessible from the entire system, while owning all security permissions of the system process. To process events and enforce the policy, the PEP queries the PDP that decides whether or not a specific event is allowed or should be blocked, modified or accompanied by other events. Different kinds of events are observed to control the behavior of applications. Permission checks are monitored in order to improve the current permission system and to control the usage of security relevant components. By monitoring intents, the interaction of components of different apps is controlled. Finally, the monitoring of content provider requests allows the observation of the usage of private data. By the usage of TaintDroid, tracking the flow of data through the system is possible. Therefore, our system considers the information flow in intents and content provider requests. Because this alone is not sufficient, additional hooks were placed to observe the information flow between apps and the file system, the network and remote services (IPC). As a user front-end, the Security-Manager app enables the user to securely manage policies on the device and control the monitor. The Security-Manager is deployed as an integral part of Android and cannot be uninstalled by the user. Authentication between the Security-Manager and the monitor is achieved by Android's IPC mechanism Binder.

We evaluated security and performance of our system. The security evaluation showed that the system can be considered as secure, in a sense that it is not possible for attackers to circumvent the monitor under the stated assumptions. The performance overhead was shown to be in an acceptable range for realistic end-user scenarios.

We did not discuss several challenges in this paper, including conflict detection and resolution among policies and the usability of the policy specification. These are the subject of current work as is the task of increasing security by also considering the SD card and the USB interface and the persistent storage and recovery of the monitor state after a system reboot.

References

[1] Adobe livecycle rights management es. http://www.adobe.com/products/livecycle/rightsmanagement/indepth.html, Aug. 2010.

[2] G. Bai, L. Gu, T. Feng, Y. Guo, and X. Chen. Context-aware usage control for android. In *SecureComm*, pages 326–343, 2010.

[3] D. Barrera, H. Kayacik, P. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM.

[4] A. Beresford, A. Rice, and N. Skehin. Mockdroid: trading privacy for application functionality on smartphones. In *Proc. 12th Workshop on Mobile Computing Systems and Applications*, 2011.

[5] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 196–206, 2007.

[6] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. The S3MS.NET Run Time Monitor: Tool Demonstration. *ENTCS*, 253(5):153–159, 2009.

[7] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010. To appear.

[8] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *Security Privacy, IEEE*, 7(1):50 –57, jan.-feb. 2009.

[9] U. Erlingsson and F. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. New Security Paradigms Workshop*, pages 87–95, 1999.

[10] Gartner. http://www.gartner.com/it/page.jsp?id=1848514, November 2011.

[11] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 8(6):399–422, 2009.

[12] M. Harvan and A. Pretschner. State-based Usage Control Enforcement with Data Flow Tracking using System Call Interposition. In *Proc. 3rd Intl. Conf. on Network and System Security*, pages 373–380, 2009.

[13] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6, Aug 2004.

[14] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In *Proc. ESORICS*, pages 531–546, 2007.

[15] I. Ion, B. Dragovic, and B. Crispo. Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices. In *Proc. Annual Computer Security Applications Conference*, pages 233–242. IEEE Computer Society, 2007.

[16] P. Kumari, A. Pretschner, J. Peschla, and J.-M. Kuhn. Distributed data usage control for web applications: a social network implementation. In *Proceedings of the first ACM conference on Data and application security and privacy*, CODASPY '11, pages 85–96, 2011.

[17] MASTER consortium. MASTER Deliverable 5.1.1: Security Enforcement Language. http://www.master-fp7.eu/, Apr. 2010.

[18] Microsoft. Windows Rights Management Services. http://www.microsoft.com/windowsserver2008/en/us/ad-rms-overview.aspx, 2010.

[19] M. Nauman and S. Khan. Design and implementation of a fine-grained resource usage model for the android platform. *Int. Arab J. Inf. Technol.*, 8(4):440–448, 2011.

[20] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.

[21] M. Ongtang, K. Butler, and P. McDaniel. Porscha: policy oriented secure content handling in android. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 221–230, New York, NY, USA, 2010. ACM.

[22] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 340 –349, dec. 2009.

[23] J. Park and R. Sandhu. The UCON ABC usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004.

[24] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[25] A. Pretschner, M. Buechler, M. Harvan, C. Schaefer, and T. Walter. Usage control enforcement with data flow tracking for x11. In *Proc. 5th Intl. Workshop on Security and Trust Management*, pages 124–137, 2009.

[26] A. Pretschner, M. Hilty, and D. Basin. Distributed usage control. *Commun. ACM*, 49(9):39–44, 2006.

[27] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *Security Privacy, IEEE*, 8(2):35 –44, march-april 2010.

[28] D. X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *ICISS*, pages 1–25, 2008.

[29] T. Vennon and D. Stroop. Threat analysis of the android market, 2010. http://www.globalthreatcenter.com/wp-content/uploads/2010/06/Android-Market-Threat-Analysis-6-22-10-v1.pdf.

[30] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proc IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

[31] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on android). In *4th International Conference on Trust and Trustworthy Computing, Pittsburgh, Pa.*, 2011.