MOSES: Supporting Operation Modes on Smartphones

Giovanni Russello
Department of Computer Science
University of Auckland
Auckland, New Zealand
g.russello@auckland.ac.nz

Mauro Conti
Università di Padova,
Padova, Italy
conti@math.unipd.it

Bruno Crispo
Università di Trento
Trento, Italy
crispo@disi.unitn.it

Earlence Fernandes
Vrije Universiteit Amsterdam
The Netherlands
earlence@cs.vu.nl

ABSTRACT

Smartphones are very effective tools for increasing the productivity of business users. With their increasing computational power and storage capacity, smartphones allow end users to perform several tasks and be always updated while on the move. As a consequence, end users require that their personal smartphones are connected to their work IT infrastructure. Companies are willing to support employee-owned smartphones because of the increase in productivity of their employees. However, smartphone security mechanisms have been discovered to offer very limited protection against malicious applications that can leak data stored on them. This poses a serious threat to sensitive corporate data. In this paper we present MOSES, a policy-based framework for enforcing software isolation of applications and data on the Android platform. In MOSES, it is possible to define distinct *security profiles* within a single smartphone. Each security profile is associated with a set of policies that control the access to applications and data. One of the main characteristics of MOSES is the dynamic switching from one security profile to another.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*access controls, information flow controls*

Keywords

Android Security Extension, Separation of Modes, Light Virtualisation

1. INTRODUCTION

The total smartphone sales by the end of 2011 reached almost half a billion worldwide. Analysts expect that these figures will double by 2015 [7]. Of these, 100 million are sold only in the US, where smartphone penetration will overtake feature phone penetration by the end of 2011. Almost half of the new smartphones (43%) are equipped with Android OS [3]. These few statistics are enough to show how popular and pervasive smartphones are becoming and the important role of Android in this market.

Such rapid growth is mostly justified by the fact that these mobile platforms are open to any third party to develop new applications and services. Consumers can easily download and install applications via well known distribution points like the Android Market. This openness has thus created plenty of new business opportunity. At the same time, however, it has raised some new security concerns. Recently, several cases of privacy-abusing applications have hit the media [5, 1]. Given the popularity of platforms such as Android, it is not a surprise that this is a growing trend. Only in the first half of 2011, between half a million to a million Android users have installed malware-contaminated applications in their smartphones [4].

1.1 Motivations

With their increasing computational power and storage capacity, smartphones allow end users to perform several tasks while being on the move. As a consequence, end users require that their personal smartphones are connected to their work IT infrastructure. Companies are willing to support employee-owned smartphones because of the increase in productivity of their employees and avoiding the need for them to carry around several devices (i.e. at least one for work, and one for private computing). Several device manufacturers are even following this trend by producing smartphones able to handle two SIMs (Subscriber Identification Modules) at the same time.

However, because users can install third-party applications on their smartphones, several security concerns may arise. For instance, malicious applications may access emails, SMS and MMS messages stored in the smartphone containing company confidential data. This poses serious security concerns to sensitive corporate data, especially when the standard security mechanisms offered by the platform are not sufficient to protect the users from such attacks.

One possible solution to this problem is to compartmentalize the phone, by keeping applications and data related to work separated from recreational applications and private/personal data. Within the same device, separate *security environments* might exist: one security environment could be only restricted to sensitive/corporate data and trusted applications; a second security environment could be used for entertainment where third-party games and popular applications could be installed. As long as applications from the second environment are not able to access data of the first envi-

SACMAT'12, June 20–22, 2012, Newark, New Jersey, USA.

ronment the risk of leakage of sensitive information can be greatly reduced.

Such a solution could be implemented by means of virtualisation technologies where different instances of an OS can run separately on the same device. Although virtualisation is quite effective when deployed in full-fledged devices (PC and servers), it is still too resource demanding for embedded systems such as smartphones. Another approach that is less resource demanding is para-virtualisation. Unlikely full virtualisation where the guest OS is not aware of running in a virtualised environment, in para-virtualisation it is necessary to modify the guest OS to boost performance. Para-virtualisation for smartphones is currently in development and several solutions exist (e.g., Trango, VirtualLogix, L4 microkernel [30], L4Android [21, 16]). However, all the virtualisation solutions suffer from having a coarse grained approach (i.e. the virtualised environments are completely separated, even when this might be a limitation for interaction). Furthermore, the switch among the environments takes a significant amount time and battery.

1.2 Contributions

In this paper, we propose a *light virtualisation* solution for Android phones. We named our solution **MOSES** (MOde-of-uses SEparation for Smartphones). MOSES is a policy-based framework for enforcing software isolation of applications and data. In MOSES, it is possible to define distinct *security profiles* within a single smartphone. Each security profile is associated with a set of policies that control the access to applications and data. One of the main characteristics of MOSES is the dynamic switching from one security profile to another. Each profile is associated with a context as well. Through the smartphones sensors, MOSES is able to detect changes in context and to dynamically switch to the security profile associated with the current context. We have implemented MOSES and performed several performance tests. The results of our experiments show that MOSES overhead is minimal and not noticeable to the end user.

The rest of this paper is organised as follows. Section 2 provides an overview of the security framework of standard Android. In Section 3, we describe an application scenario to better illustrate the problem that we are addressing in this paper. Section 4 presents the architectural details of MOSES. Section 5 is focused on the main concept of our approach that is the separation of security profiles. The management of MOSES and security profiles are described in Section 6. To demonstrate the effectiveness of MOSES, we revisit our application scenario in Section 7. We have implemented MOSES and the evaluation of its performances is analysed in Section 8. In Section 9, we review existing approaches that aim at extending the security mechanism of the Android platform. Finally, Section 10 provides our concluding remarks and highlights future research directions.

2. ANDROID SECURITY

Google Android is a Linux-based mobile platform developed by the Open Handset Alliance (OHA) [2]. Most of the Android applications are programmed in Java and compiled into a custom bytecode that is run by the Dalvik Virtual Machine (DVM). In particular, each Android application is executed in its own address space and in a separate DVM. Android applications are built combining any of the following four basic components. *Activities* represent a user interface; *Services* execute background processes; *Broadcast Receivers* are mailboxes for communications within components of the same application or belonging to different applications; *Content Providers* store and share application's data. Application components communicate through messages called *Intents*.

Focusing on security, Android combines two levels of enforcement [18, 29]: at the Linux system level and the application framework level. At the Linux system level Android is a multi-process system. During installation, an application is assigned with a unique Linux user identifier (`UID`) and a group identifier (`GID`). Thus, in the Android OS each application is executed as a different user process within its own, isolated, address space.

At the application framework level, Android provides access control through the Inter-Component Communication (ICC) reference monitor. The reference monitor provides Mandatory Access Control (MAC) enforcement on how applications access the components. In the simplest form, protected features are assigned with unique security labels—*permissions*. Protected features may include protected application components and system services (e.g. Bluetooth). To make the use of protected features, the developer of an application must declare the required permissions in its package manifest file: `AndroidManifest.xml`.

As an example, consider an application that needs to monitor incoming SMS messages, `AndroidManifest.xml` included in the application's package would specify: `<uses-permission android:name= "android.permission.RECEIVE_SMS"/>`. Permissions declared in the package manifest are granted at the installation time and can not be modified later. Each permission definition specifies a protection level which can be: `normal` (automatically granted), `dangerous` (requires the user confirmation), `signature` (requesting application must be signed with the same key as the application declaring the permission), or `signature or system` (granted to packages signed with the system key).

3. EXPLANATORY SCENARIO AND REQUIREMENTS

In this section, we present an application scenario that will be used throughout the rest of this paper to demonstrate the capabilities of MOSES. Moreover, we list a set of requirements drawn from the application scenario that will be used for comparing our approach with existing ones.

More and more companies nowadays provide mobile versions of their desktop applications. Studies have shown that allowing access to enterprise services with smartphones increase employees' productivity [25]. An increasing number of companies are even embracing the BYOD: Bring Your Own Device policy [6], leveraging the employee's smartphone to provide mobile access to company's applications.

Wise Inc. is one of such enterprises. Wise Inc.'s employees have to install on their smartphone GroupMoveApp, a document collaboration application for Android allowing employees to view, edit, and share company files from their smartphone. GroupMoveApp can store files on the local SD and it uses a remote repository for synchronising files. Wise Inc. decided to use a repository service from Smart Inc., a cloud-based company that provides a very reliable infrastructure for a fraction of the cost of developing its own solution.

From this simple scenario, we can identify the following security requirements.

- R1: All the company files stored on the smartphone have to be accessed only by the GroupMoveApp (or any other application allowed by Wise Inc.). Any applications installed by the employee and not authorised by Wise Inc. should not be able to access company files.

- R2: Company files can only be sent to the repository managed by Smart Inc. For instance, the user should not be able to use GroupMoveApp in a way such that the storage operation is hijacked to a destination different from Smart Inc. Similarly, if the employee uses DropBox (i.e. an application different from GroupMoveApp) for the backup of her own files, she should not be able to drop company files in DropBox.

- R3: At the same time, to protect the employee's privacy from Wise Inc., any personal files stored in the smartphone should not be accessible to GroupMoveApp and/or stored in the repository of Smart Inc.

- R4: Applications should not be able to use permissions not granted to them by exploiting other application permissions. For instance, GroupMoveApp may get infected by malware that tries to send company files to a malicious server by using the internet permission of the GroupMoveApp. The malware should not be able to send the company files to another server on the internet.

- R5: Finally, all the isolation features should be enforced on a context-based mode. As an example, the phone might not be allowed to run gaming applications during working hours, while it could be allowed to do so in other contexts. Similarly, an application should be allowed to access some specific data only under specific circumstances. For instance, when on the train the employee should not access very sensitive company data. This is to prevent other passengers from possibly reading it. As another example, the use of some applications (e.g. games) might be restricted under several circumstances (e.g. low battery).

The security mechanism offered by standard Android is not adequate to satisfy the requirements listed above. For instance, if the user grants an application the permission to access the local SD storage and internet then that application can read any file in the SD and send it to any server (thus violating requirements R1, R2, and R3). It is well-known that standard Android security is vulnerable to privilege spreading attacks [17], where an unprivileged application exploits the permissions of privileged applications (in clear violation of our requirement R4). Things in standard Android are even worse. Applications can export services that other applications can use without the user being aware of this. Given the open approach taken by Google that allows developers to create applications for the Android Market by just paying a very small fee ($25), designing colluding applications, that, on purpose provide to other applications their own permissions, is becoming increasingly popular [11, 28, 22, 14]. Finally, in Android there is no notion of dis/enabling applications or accessing data based on the notion of context. The user can start any application and accessing any files at any time and in any place (violating requirement R5)[1].

In the literature, several approaches have been proposed that satisfy some of the above requirements. However, to the best of our knowledge none is able to satisfy all of the requirements at once. Finally, it is important to realise that the aim of this work is not to protect the corporate data from an employee that is actively engaged in leaking sensitive data. In the rest of this paper, we assume the smartphone user is not willing to behave maliciously, for instance by installing on her smartphone a rogue application for intentionally leaking sensitive corporate data.

In the following section, we present MOSES, our Android security extension for data and application isolation which is able to satisfy all the above requirements.

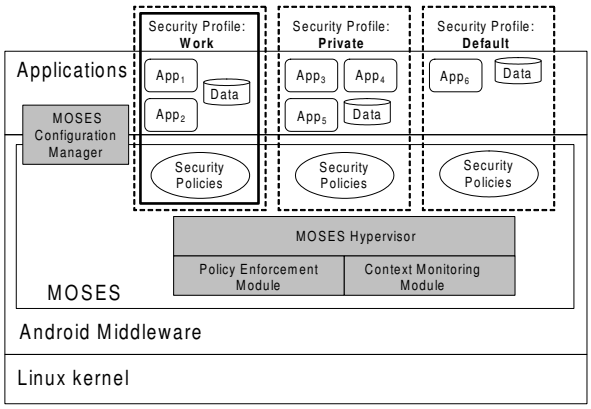

Figure 1: MOSES Overview.

4. MOSES

In this section, we provide details about MOSES system model and architecture.

4.1 System Model

Figure 1 provides an overview of MOSES. The MOSES framework is implemented within the Android middleware and rewrites/extends some of its modules. The main concept in MOSES is that of a **Security Profile** (SP). An SP represents an operation mode that can be used as a logical isolation unit that contains: applications, data and a set of *security policies*. Through the enforcement of the security policies associated with an SP, MOSES guarantees that applications within that SP can access only the data within the same SP. MOSES achieves this fine-grained level of enforcement by means of data tainting implemented in the **Policy Enforcement Module** (PEM). More details on this will be provided later. Here it suffices to say that the data within a given SP is tainted with the SP name. The security policies specified in that SP enforce the constraint that applications can only access data tainted with the label of the same SP name. For instance, in Figure 1 the data in the "Work" SP is tainted with the label "Work". The security policies of the "Work" SP grant access to the data only to applications contained in the same SP.

MOSES supports several SP instances within the same device. By default, the "Default" SP is always present in MOSES. This SP can be used for containing newly installed applications that are not associated with any SP, or for data that is not tainted with any label. A user can create new SPs and associate data and applications to the profile by means of the **MOSES Configuration Manager** (MCM). The user can use the MCM to edit the settings of existing ones. However, an SP can also require special credentials to be edited. For instance, the "Work" SP in Figure 1 is a special profile that the user owning the smartphone cannot edit. This profile has been created by the IT administrator of the company for which the user of the smartphone works (e.g., Wise Inc.). In this way, the company can make sure through MOSES that only the applications in the SP "Work" are allowed to access the company data. There is no limit

[1]R5 is partially addressable with Android 4.0, where it is possible to enable/disable the camera according to time and location through the Device Admin API.

to the number of different SPs that MOSES can support. However, for sake of simplicity, in the rest of this paper we consider only two profiles in the phone: "Work" and "Private". As we already said above, the "Work" SP is used for accessing work-related data through company-approved applications. The "Private" SP is used by the user for accessing private information such as emails and SMS messages from family and friends. Also, in "Private" SP the user can install her preferred applications and games.

Activations and deactivations of SP instances are executed by the **MOSES Hypervisor** (MH). When an SP is activated, the MH loads the security policies of the SP in the **Policy Enforcement Module** (PEM). When an application requests access to a piece of information, the PEM grants access only if a security policy in the SP grants such request. A user can switch manually from one SP to another. However, MOSES provides a more advanced mechanism where contextual information is used for automatically switching SP. In MOSES, SPs may be associated with context information (for instance location and time). When a given context is detected then the MH activates the respective SP. The context is detected through the **Context Monitor System** (CMS). For instance, the "Work" SP can be activated only during working hours and within the office facilities. Only outside the working environment, the employee is allowed to access applications and data within her private profile.

Context can be also used for automatically labelling data and applications. For instance, if a new contact is added to the phone contact list, the context and the current profile of the phone can be used to determine which label to use for tagging the new contact. Similarly for a new application that is installed to the phone.

On the other hand, a label associated with the data together with the current profile of the phone can be used to determine the behaviour of the phone. For instance, if the user receives a SMS from a private contact while the current SP is "Work" then instead of presenting directly the SMS to the user, MOSES can buffer the SMS and present the SMS only when the SP changes to "Private".

4.2 Architecture

In this section, we describe in more detail the internal components of each of the modules within the MOSES architecture. The components of each module are depicted in Figure 2.

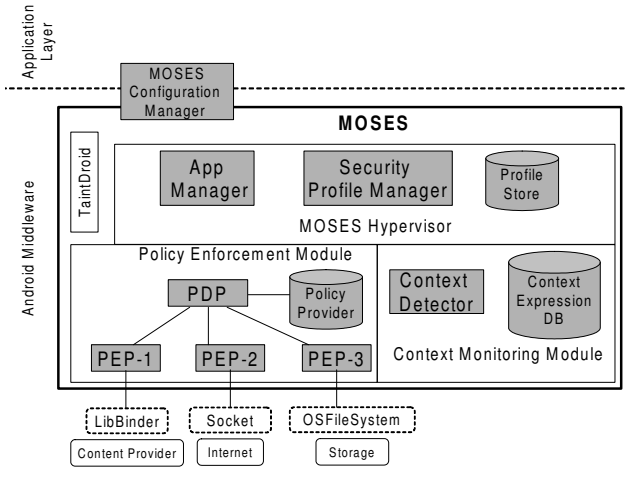

Figure 2: MOSES Architecture.

The MOSES Hypervisor (MH) represents the core module in MOSES. Within the MH, the **Security Profile Manager** is responsible for activating and deactivating the different SP instances defined in the **Profile Store**. The Profile Store obtains SP instances from the user either through the MOSES Configuration Manager or from authorised third-parties via SMS, MMS and Bluetooth. The SP switching can be done manually by the user or automatically by detecting the context in which the user is. In this latter case, the Security Profile Manager receives context information through the Context Monitoring Module (CMM). In the CMM, the **Context Detector** monitors the actual context by means of the smartphone sensors. Context in MOSES is defined as a boolean expression on the values generated by sensors. Context expressions associated with the SP instances contained in the Profile Store are stored in the **Context Expression DB**. Periodically, the Context Detector samples the different sensors and checks whether any context expression is satisfied. When a context expression is satisfied, the Context Detector notifies the Security Profile Manager that a new context has become active. If the new context is associated with an SP different from the one that is currently active, the Security Profile Manager makes active the corresponding SP. If no SP is associated with the new context, no changes are required.

The switching of SP consists in executing the following steps. Firstly, the Security Profile Manager notifies the **App Manager** to disable all the applications associated with the current SP. If applications are still active then the App Manager forces them to terminate. Secondly, the Security Profile Manger disables the set of security policies of the current SP that are stored in the **Policy Provider** (a component of the Policy Enforcement Module). Thirdly, the set of security policies associated with the new SP are enabled in the Policy Provider. Finally, the Security Profile Manager retrieves the list of applications of the new SP and notifies the App Manager to enable them.

The enforcement of the security policies happens within the Policy Enforcement Module (PEM). When an application requests access to a resource, the **Policy Enforcement Point** (PEP) intercepts such a request. The PEP collects information about application UID, the resource being accessed and the type of operation. The PEP forwards this information to the **Policy Decision Point** (PDP). The PDP uses the information received by the PEP to evaluate the security policies relevant to the request stored in the Policy Provider. Based on the evaluation of the policies, the PDP might decide either to allow or disallow the request. The PDP informs the PEP of the decision and then it is the responsibility of the PEP to take the necessary actions for the enforcement of such a decision.

In Android, several components are responsible for mediating access requests of applications to the device resources. Therefore, we need to connect several PEPs with these components within the Android Middleware to intercept such requests and to enforce the PDP decisions. The PEP-1 is connected with the `LibBinder` module for intercepting requests to access simple resources, such as device ID (IMEI), phone number and location data, as well as complex data such as user's calendar and contact entries.

In the `LibBinder`, we intercept the standard cursor from where we extract the `CursorWindow`. The `CursorWindow` provides methods that can be used for modifying the data contained in the cursor. Using the `CursorWindow` allows us to filter out from the cursor data only part of the information. In this way, our enforcement mechanism achieves a fine-grained filter capability. For instance, if a work application retrieves the contact entries from the contact provider, all the private contact entries can be filter out from the data contained in the `CursorWindow` before it is returned to the application.

Other PEPs are connected with some classes of the Java Framework Library (JFL) in the Dalvik Virtual Machine. In particular, the PEP-2 is connected with the `Socket` class for controlling network traffic even if sent over an encrypted socket (SSL). In the `Socket` class, we have modified the `socket.open(address)` method to inspect the address to where the data is sent. In this way, we can restrict the use of only authorised addresses or substitute the address specified by the application with an address defined by the user. By modifying the `sendStream()` method, we are able to intercept the data before it is sent and perform some actions, such as filtering or substitutions. Finally, for capturing operations on the file system, such as reading and writing on the local storage, the PEP-3 is connected with the `OSFileSystem` class.

5. REALISING SECURITY PROFILE ISOLATION

A central notion in MOSES is that of an SP (Security Profile) representing our unit of isolation to separate execution of applications and data accesses. This isolation is achieved by (i) separation of application executions, (ii) enforcement of security policies for accessing data, and (iii) dynamic adaptation through context. In the following, we provide details of each of these features.

5.1 Application Activity Separation

Separation of application executions is achieved by means of the App Manager component contained in the MH (shown in Figure 2). Each SP contains the list of applications that are allowed execute when the SP is active. To provide an immediate feedback to the user about which applications she is allowed to launch under a given SP, we have modified the Android `PackageManagerService` to display in the App Launcher only the applications defined in the active SP.

The App Manager is also responsible for terminating the applications associated with the SP that is being deactivated. If a process is not at the top of the Activity Stack, then the process will be just killed. Otherwise, if the process is the one in foreground, the App Manager launches a "decoy" activity which forces the previous activity to be pushed to run in background. This, in turn, forces the execution of `onPause()` in the Activity lifecycle, which gives developers a chance to gracefully save the process state. We then terminate the process and the "decoy" as well.

```
1 PolicyName: allow to Requester Operation on Target
2             with scope SP-Name
3             [perform Action(param-list)]
4             [while Condition]
```

Figure 3: The syntax of the MOSES policy language.

5.2 Security Policies

To constrain applications to access only data defined for the active SP, we leverage a data tainting mechanism. The main idea is that data within an SP is tainted with the SP name. For tracking the data, we use the TaintDroid labelling framework. We have extended TaintDroid to be able to use as labels the SP names.[2] Each taint is represented as a 32-bit value used to define the control group, the taint label, and some extra information used for history based inspection. The control group is used to specify whether the data is coming from a system resource such as the GPS provider by means of the `"SYSTEM_SENT"` tag. Also the control group can be used to specify that the label associated with a data can be set as a consequence of a policy evaluation. This is particular useful if the taint of data needs to be augmented with labels to keep track of all the applications that have received the data.

We have developed the MOSES policy language for specifying security policies. Here we briefly introduce the syntax and semantics of the language. Afterwards, in Section 7 we will present more examples of security policies for our application scenario to demonstrate the power and flexibility of our approach.

Figure 3 shows the syntax of a MOSES policy. Policies are identified by a name and define what `Operation` a `Requester` application is allowed to execute on `Target` resources. In MOSES, a resource can represent system content providers, system service providers, and services exposed by other applications. The **`with scope`** clause controls whether the requesting application is accessing data within the scope of the given `SP-Name`. Finally, a policy can have two optional clauses: **`perform`** and **`while`**. The **`perform`** clause specifies actions that have to be performed if this policy is enforced. MOSES provides a set of libraries that can perform actions on the data (such as, filtering, anonymisation, generation of random values, data encryption) and on the values of the parameters of the requested operation. Depending on the nature of the action, this clause can be performed before the right is granted (i.e., checks on the parameters of the requested operation) or after the operation is performed (i.e., data filtering). The **`while`** clause contains a condition that is a boolean expression. For the system to grant the access right to the requester, the condition needs to be true at the time of policy evaluation. Moreover, if the operation is granted over a period of time it might be the case that over time the initial condition does not hold true. By means of the **`while`** clause, we can enforce that the access right will be valid while the condition holds true.

5.3 Dynamic Behaviour Through Context

One of the main contributions of MOSES compared to other similar approaches is the use of context for controlling the activation and deactivation of SPs. In MOSES, each SP is associated with one or several contexts. A context is defined as a boolean expression over data collected directly from the device physical sensors (such as GPS, clock, Bluetooth, etc.). A context expression can also be defined on *logical sensors*, that is functions that combine raw data from physical sensors to capture specific user behaviours, such as detecting when the user is running. For instance, a "Work" SP that should be activated when the user is performing job-related activities could be associated the following context expressions: *Work@Office*{`(Time>8) AND (Time<18) AND (Location=OFFICE)`} and *Work@Home*{`(Time>18) AND (Time<24) AND (Location=HomeOffice) AND NOT (isWatchingFootballMatch)`}.

When a new SP is stored in the Profile Store, the Security Profile Manager writes the SP's context expressions into the Context Expression DB. The context expressions are periodically evaluated by the Context Detector with data obtained by the different sensors. Whenever a context expression evaluates to true, the Context Detector retrieves the name of the SP associated with the context expression and notifies the Security Profile Manager for the SP activation.

[2] Actually with our modifications of TaintDroid any labels can be used to taint data. For instance, it is possible to specify different labels for tainting data with different levels of sensitivity. For sake of simplicity, in this work we require that data is tainted with at least an SP name.

6. SECURITY PROFILE MANAGEMENT

In this section, we describe how the SPs are managed in MOSES. The module responsible for SP management is the MOSES Configuration Manager. The internal components of this module are shown in Figure 4. The **Profile Manager App** is an application that allows the user to create an SP and modify existing ones. The application also allows the user to define and edit context expressions that later the user can associate with an SP. The Profile Manger App stores and retrieves the context expressions to and from the **ContextDef** content provider. When a new context definition is stored in ContextDef, a conflict check is performed to avoid that the new context definition is overlapping with the context definitions already stored in the ContextDef. As a matter of fact, if two or more context definitions overlap then it might be the case that in a given situation more than one SP needs to be activated. We decided to have here a very restrictive approach by avoiding that overlapping context definitions can be stored in the ContextDef. However, as part of our future research direction we will explore remediation strategies such as prioritising each SP to select the one with highest priority.

The **Profile Register** component is responsible for storing and retrieving the SP definitions. When a new SP is created, the Profile Register stores the SP definition in the Profile Store and it also registers the context expressions associated with the SP in the Context Expression DB. In this way, the new SP can be activated if the Context Detector evaluates to true the context expressions associated with it. In MOSES, each SP has assigned an *owner* that is the entity authorised to define and modify the SP. The owner of an SP can be the user of the device that creates her own SP. However, a user can deploy on her device SPs defined by third-parties. To protect the SP from unauthorised modification, we support several mechanisms for authenticating the SP owners, such as passwords, certificate, and biometric authentication.

SPs can also be edited/updated remotely. In this case, the requests are handled by the **Remote Manager** component. Edit/update Requests can be sent through SMS/MMS and/or Bluetooth. The authentication of remote requests can be performed through the SP owner's certificate. When a remote request for an update is made, first Authenticator verifies the validity of the certificate of the owner: the certificate includes the identity and the owner's public key, all these signed with the key of the certification authority. The trust architecture for remote management of SPs (via messages sent to the device) is organised as a Public Key Infrastructure (PKI). An incoming message containing a new version of a SP has to come with the certificate of the sender. A certificate can be transmitted in-band or just as an ID corresponding to a cached certificate in the **CertificateCache**. All certificates should be in the X.509 format. We use standard Java APIs to manipulate and verify certificates. The CA certificate is embedded in the system image at build time. All other certificates are cached in the `/data/moses/certificates` directory. The algorithm used for signature is SHA1 with RSA and a 2048-bit RSA public key. For all the algorithms, we use the BouncyCastle APIs – as done by Android itself. After the authentication phase completes successfully, the Remote Manager uses the Profile Register component to store the SP definition in the Profile Store and register the context expressions in the Context Expression DB.

7. APPLICATION SCENARIO REVISITED

In this appendix, we present the MOSES policies used in application scenario presented in Section 3 when MOSES is used.

The listing in Figure 5 shows the MOSES policies defined for the

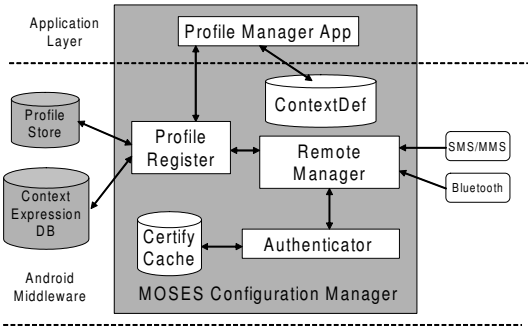

Figure 4: MOSES Configuration Manager.

```
1  WorkP1: allow to GMA ANY on ANY
2      with scope ''Work''
3      while context.isActual(''WorkOffice'')
4
5  WorkP2: allow to GMA Send on Internet
6      with scope ''Work''
7      perform sendOnlyTo(''www.smartinc.com'')
8
9  WorkP3: allow to GMA Read on ANY
10     with scope ''Work''
11     while location.isActual(''COMMUTING'') and
12           !ANY.level(''VerySensitive'')
```

Figure 5: The MOSES policies defined in the "Work" SP for the GroupMoveApp.

GroupMoveApp. In particular, the policy `WorkP1` specifies that the GroupMoveApp (identified in the policies as `GMA`) can perform any operations on any work data (line 2) while the user is in her office (captured in line 3 by the **`while`** clause). The policy `WorkP2` enforces that the application sends over the Internet work data and it can connect only to the url specified in the `sendOnlyTo` action in the **`perform`** clause (line 7). The policy `WorkP3` authorises the GroupMoveApp to read work data while the actual location of the user is on a train or a bus (line 11) as long as the sensitivity level of the data is not very high (line 12).

In the following, we discuss how MOSES addresses the requirements listed in Section 3. To guarantee that only applications authorised by Wise Inc. are authorised to access work data, as for requirement R1, the "Work" SP has to contain for each authorised application MOSES policies similar to `WorkP1`. MOSES implements by default a negative authorisation policy meaning that if no MOSES policy exists for a given application then the system does not authorise any operations on any resources coming from that application. If a MOSES policy exists then the **`with scope`** clause has to be satisfied. This clause makes sure that each authorised application accesses data associated with the same SP (by means of the tagging mechanism). As for the protection of the employee's privacy (requirement R3), MOSES policies defined in the "Work" SP will grant access to applications only to data tagged with the label `"Work"`. In this way, any employee's private data will be not accessible to any Wise Inc. applications. By means of the `sendOnlyTo` action in the **`perform`** clause, policies are able to enforce restrictions on where the data is being sent, thus satisfying requirement R2. This mechanism is also effective in the event the GroupMoveApp gets infected by a malware application that tries to exploit the GroupMoveApp permission to send data over the Internet. The malware could try to open a socket to send the work data to another server. However, policy `WorkP2` will prevent such an

action from happening because the action on the **perform** clause will not be satisfied and the operation will not be permitted (satisfying requirement R4). Finally, in MOSES contextual information plays a fundamental role. Context information is used for controlling the activation and deactivation of the SPs. Moreover, as shown in policies `WorkP1` and `WorkP3`, context can be used for granting access rights though the evaluation of MOSES policies.

8. MOSES PERFORMANCE EVALUATION

In this section, we will present the results of our testing to measure the overheads introduced by MOSES. Since the time overhead is a central concern of user experience, we evaluate the time overhead introduced by our security extensions compared to a standard Android system. At the same time, we understand that MOSES also brings overhead in terms of battery consumption. In the following, we concentrate on evaluating time overhead and battery consumption of the two main aspects introduced by MOSES: namely Security Profile switches and enforcement of MOSES policies.

All the experiments were run on the Samsung Nexus S phone with the 2.3.4 version of Android i.e. stock and modified platforms are based on the same version. To obtain time overheads we used a call to `System.nanoTime()` before and after measured event and compute the difference between the measured values.

8.1 Security Profile Switch Overhead

We recall here the steps executed during a Security Profile (SP) switch. Firstly, the Security Profile Manager notifies the App Manager to disable all the applications associated with the current SP. If applications are still active then the App Manager forces them to terminate. Secondly, the Security Profile Manger disables the set of MOSES policies of the current SP that are stored in the Policy Provider. Thirdly, the set of MOSES policies associated with the new SP are enabled in the Policy Provider. Finally, the Security Profile Manager retrieves the list of applications of the new SP and notifies the App Manager to enable them.

To measure the time overhead of an SP switch, we devised the following experiment. We created two SPs, namely "Work" and "Private". Each SP is associated with 100 MOSES policies and four applications (that are just dumb activities used to fire up a Linux process). The test forces the system to execute 100 SP switches. Between each switch, the four applications are started. We measured the time that MOSES requires for completing the switch, namely from the instant the Security Profile Manager notifies the App Manager to disable the applications till the App Manager enables the applications associated with the new SP. The results are shown in Figure 6 (where the x-axis represent the 100 switches). As we can see, except for few outliers, the switching time is less than one second.

To measure the power consumption for the SP switching, we executed the same experiment we performed for time measurement of SP switches. Only this time we executed the switches over a period of 1 hour (resulting in 2400 SP switches). At the start of the experiment the battery was at a full charge level. After the experiment was concluded, the level of the charge dropped to 77%. This means each SP switch consumes 0.009% of a full battery.

8.2 MOSES Policy Enforcement Overhead

The second set of experiments aim at measuring the overhead in terms of time and battery consumption of the policy enforcement in MOSES. MOSES policies are enforced when applications request access to data. To measure the time overhead, we run an application that performed 100 read operations on GPS data. We first execute the application on stock Android, to measure the average time of a single GPS read operation. Then we execute the same application, but this time MOSES is activated. To be able to access the GPS data, the MOSES policy `TestP1`, shown in Figure 7 is enabled in the Policy Provider in the "Private" SP.

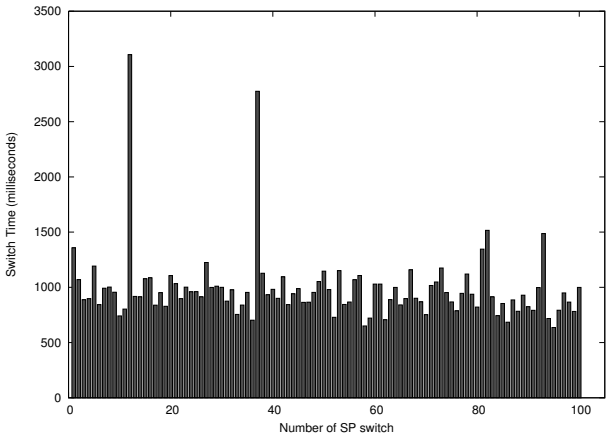

Figure 6: Time for Security Profile switching.

```
1 TestP1: allow to TestApp Read on GPS
2     with scope ''Private''
```

Figure 7: The MOSES policy defined for the TestApp in "Private" SP to read the GPS data.

We run several tests with MOSES enabled, each time increasing the number of MOSES policies present in the Policy Provider from 10 to 100. However, in each configuration we make sure that the `TestP1` is always the last to be evaluated resulting in the worst case scenario. The result are shown in Table 1. As we can see, the average time for accessing the data is around 1 millisecond in stock Android. When MOSES is enabled, the average time for the read operation increases from 3 (in the case of only 10 MOSES policies) to almost 10 milliseconds (in the case of 100 MOSES policies). We can conclude that the time overhead introduced by MOSES does not affect the user's experience.

Configuration	Average Time (ms)
Stock Android	1.071
MOSES-10	3.134
MOSES-20	3.813
MOSES-30	4.970
MOSES-40	5.615
MOSES-50	6.122
MOSES-60	7.205
MOSES-70	7.613
MOSES-80	7.621
MOSES-90	8.451
MOSES-100	9.658

Table 1: Performance of GPS reading operations with stock Android and with the "Private" SP activated.

The last set of experiments focus on the impact that the enforcement of MOSES policies has on the battery consumption. In order to have a tangible battery consumption we run the following experiment. We execute on stock Android the `TestApp` application to perform GPS read operations every 10 seconds over a period of 5 hours. At the start of the experiment, the battery was fully charged.

At the end of the 5 hours the percentage drop was 13%. After recharging the battery, we run again the same experiment, this time with the MOSES-100 configuration enabled (100 MOSES policies in the Policy Provider with the `TestP1` policy at the bottom). At the end of the 5 hours, the percentage drop was 17%. It should be noted that executing read operations every 10 second results in 1800 reads over a period of 5 hours. The percentage of battery consumption for a single operation in stock Android is 0.007% while with the MOSES enforcement mechanism it is around 0.010%.

From the above analysis, we can conclude that the overhead introduced in terms of time and battery consumption is negligible.

9. RELATED WORK

In this section, we provide an overview of the related work in the area, which is smartphone security: with focus on the Android system. In particular, in Section 9.1, we describe other research efforts in providing enhanced security mechanisms to the Android platform. In Section 9.2 we discuss solutions that could be used to solve (even though: only to some extent, partially, and in a non efficient way), the problem we address.

9.1 Security Approaches

In Android, at installation time users grant applications the permissions requested in the manifest file. Android supports an all-or-nothing approach, meaning that the user has to either grant all the permissions specified in the manifest or abort the installation of the application. Moreover, once an application is installed, the only way to revoke a permission is to completely uninstall the application.

To circumvent this coarse-grained approach, several solutions have been proposed to allow the user to manage in a more fine-grained way application permissions even during runtime. Saint [24] is a policy-based application management system that controls application permissions at install time and during runtime. Saint aims at controlling how applications interact with each other. Clearly, Saint is not aimed at solving the problems identified in our scenario. As a matter of fact, Saint policies can help in ensuring that applications authorised by the company are not invoked by user's applications (partially addressing requirement R1). However, Saint policies cannot prevent a user's application from accessing sensitive company data because there is no mechanism that facilitates distinguishing between private and work data. Finally in Saint, access rights cannot be granted on the basis of the actual context of the user since there is no way of defining context in the Saint policies.

Context information plays a pivotal role in the approaches presented by Nauman et al. [23] and Conti et al. [13]. Here, context is used to trigger rules at runtime, that, to some extent can also be used to enforce security properties. Bai et al. [8] has further extended this approach to support a UCON security model. Although these approaches can satisfy our requirement R5, none of them are able to guarantee that only company-authorised applications access company data (R1), to control the dissemination of the data (R2), protect the privacy of the users (R3), and protect the system from malicious spreading of permissions (R4).

More recent papers [9, 31] concentrate on the protection of the user's private data (satisfying only R3). MockDroid [9] is a system which can limit the access of the installed applications to the data by filtering out information. For instance, an application querying the contacts' provider may receive no results even if the provider is not empty. This approach is more refined in TISSA [31] where users are able to define the accuracy level of the information revealed to the application by means of privacy levels. In TISSA, it is possible to define four privacy levels: *trusted, empty, anonymised or bogus*. For instance, *anonymised* means that the information is somehow anonymised while *Bogus* means that fake information is forged for the requesting application. Unfortunately, both the approaches do not solve the problem of privilege spreading. For instance, if an application that has a *trusted* privacy level (thus accessing real data) is infected with malware then the malware can access the real data as well, clearly violating requirement R4.

TaintDroid [17] proposes dynamic taint analysis to control how data flows between applications. TaintDroid is capable of tracking sources of specific tainted data. In TaintDroid, taints are statically associated with predefined data sources, such as the contact book, SMS messages, the phone number, the device identifier (IMEI), etc. TaintDroid limits the flow of tainted data by tracking the taints in the outbound network connections (satisfying requirements R2 and R4). However, TaintDroid is not capable of enforcing separation of operation modes. For instance, TaintDroid would treat private and work contacts as the same type (because they are tainted with the same taint) applying the same policy. Therefore it is not possible to have in TaintDroid corporate applications that can only access corporate data. The same holds true for private applications (thus violating requirements R1 and R3). Similar conclusions can be drawn for Paranoid Android [26]. Paranoid Android proposes tainting of data for runtime checks. In Paranoid Android security analysis is executed by a trusted remote server, which hosts the replicas of smart phones in virtual environments. However, this approach has a severe impact on the device performance since execution traces have to be continuously sent to the remote servers. Finally, both approaches do not consider contextual information for switching between different operation modes and for enforcing context-based security policies (violating requirement R5).

QUIRE [15] provides a lightweight provenance system that prevents the confused deputy attacks where a malicious application abuses the interfaces of a trusted application to perform an unauthorised operation (R4). QUIRE addresses the problem by tracing RPC chains to establish if all callers in the chain have the necessary privileges to execute the call. Tracing is realised by modifying the Android native RPCs. This however has the drawback that QUIRE's approach is not transparent to application developers. They need to rewrite their existing applications. Furthermore, QUIRE does not support separation of operation modes, meaning that applications can access both corporate and private data, in violation of requirements R1 and R3.

A solution similar to ours is AppFence [20]. By using TaintDroid's tainting capability, AppFence provides additional mechanisms to shadow sensitive data and to block exfiltration, that is the unauthorised leakage of data via network access (R2). Shadowing allows only data anonymisation and does not support other transformations over sensitive data. In principle, AppFence could be modified to support separation of operation modes as in our approach. However, there are no means for capturing context information to be used for enabling/disabling different operation modes (violating R5).

XManDroid [11] performs runtime monitoring and analysis of communications between applications by monitoring the ICC traffic and validates whether an ICC call can potentially lead to a spreading of privileges according to a desired system policy. This can be used to avoid that malware code exploits the privileges of other applications (satisfying R4) to perform unauthorised operations. The main limitation of this approach is that it cannot be used to control communication channels established outside the ICC framework, such as Internet communications (in violation of requirement R2). The main shortcoming of XManDroid is that it does not support

separation of operation modes and context information to drive the enforcement of policies (in contrast with requirements R1, R3, and R5).

YAASE [27] is an Android security extension aiming at protecting the Android users from both confuse deputy and privilege escalation attacks. YAASE uses the data tainting capability of TaintDroid to limit application access to the resources declared in the manifest file. Similarly to XManDroid, YAASE is able to avoid that malware exploit other applications' privileges to perform unauthorised access (satisfying R4). In addition to this, YAASE is also able to control data flow outside the ICC framework (satisfying R2). However, YAASE is not designed for separating operation modes and to use context information to adapt the enforcement of policies (violating requirements R1, R3, and R5).

Finally we come to TrustDroid [12]. TrustDroid is an Android security framework that most closely matches MOSES security enhancements. TrustDroid provides separation of operation modes by "colouring" applications and data. The underlying security policy is that applications can access only data of the same colours. The separation of operation modes is supported by representing an operation mode with a colour, satisfying requirements R1, R2, and R3. Applications are statically assigned to a colour at installation time. The assignment of colours to data is somehow very constrained: when an application writes data then the data is automatically assigned the same colour of the application. TrustDroid supports basic context-based policies, such as preventing Internet access by private applications while an employee is connected to the company's network (partially satisfying R5). One of the main limitations in TrustDroid is that the security policies are very coarse-grained. Applications can read and write data of the same colour. It is not possible to enforce more fine-grained policies where some applications can only read data, while others can have a full set of rights. For instance, if we consider a scenario of micro-payments: only one application should be able to both read and modify the actual balance, while all the other applications should only be able to read the balance. Finally, TrustDroid does not perform extra checks to avoid malware that is able to use legitimate applications' permissions to send data over the internet to an adversary server (in violation of requirement R4).

9.2 Heavy separation of Operation Modes

Virtualisation provides environments that are isolated from each other, and that are indistinguishable from the "bare" hardware, from the OS point of view. The hypervisor is responsible for guaranteeing such isolation and for coordinating the activities of the virtual machines. Hence, at the same time virtualisation can: (i) increase security, while (ii) reducing the cost of deployment of applications (the hardware is shared in a secure way).

Similar security motivations, together with a higher usability (see also the motivation of our work in Section 1.1) is pushing virtualisation techniques into the smartphone scenario. In fact, several virtualisation solutions have been already proposed for smartphones [19], and they have been also already considered from security point of view: e.g. with their proposal as a tool for rootkit detection [10]. However, virtualisation does not come for free, and it is a particular demanding task for resource-constrained devices like smartphones (e.g. in terms of battery) [30]. In particular, in [30] the authors evaluate the overhead due to the virtualisation on a smartphone by comparing (using typical smartphone apps): (i) L4Linux (a para-virtualised Linux on top of L4 microkernel) with (ii) the native Linux performance. The authors conclude that while in some specific cases the overhead might be acceptable in terms of delay, it is also "use case dependent" (system call triggering more kernel activities has worst performances). Furthermore, for some system calls it has been observed an execution time is 30 times slower than the one on native Linux.

Virtualisation techniques have been recently also adapted to run a mainstream OS like Android. For example, the L4Android [21, 16] project combined L4Linux and Google modifications of the Linux kernel to enable a smartphone to run Android on top of a microkernel. However, even in this scenario, the pros and cons are inherited from the ones of virtualisation. In fact, while virtualisation is the perfect solution for our requirements R1 and R3, it cannot address requirement R2 (where isolation is not enough to describe the constraints of the operation mode of an application), it cannot address requirement R4 (isolation does not avoid confused deputy attack leveraged via applications belonging to the same environment), and it does not address requirement R5 (the actual environment running at a given time cannot be automatically defined via a context specification).

The proposal of systems like MOSES are hence motivated: from one side, by the need of virtualisation features on smartphones; from the other side, by the need to have a virtualisation that is efficient in terms of time, and energy overhead—which are still main issues for resource-constrained devices like smartphones.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented MOSES: a policy-based framework for Android that enables the separation and isolation of applications and data. Crucial in MOSES is the notion of security profiles. Each security profile represents a unit of isolation enforcing that applications can only access data of the same security profile. One of the innovative aspects introduced by MOSES is the dynamic switching from one security profile to another. Through the smartphones sensors, MOSES is able to detect changes in context and to dynamically switch to the security profile associated with the current context.

One of our main concerns was the impact on the smartphone user's experience when MOSES is used. In this respect, we implemented MOSES and analysed the overhead in terms of time and battery consumption introduced by MOSES. The results of our experiments show that MOSES overhead is minimal and not noticeable to the end user.

As future work, we are currently expanding the functionality of MOSES to enable the protection of data within a given security profile in case the user loses the smartphone. One possibility is to introduce encryption capabilities linked to the user's identity. Another option is to use the Mobile Trusted Module to validate the current context of the smartphone to decrypt the data only in a trusted environment. Another direction of future research is the distribution of security profiles and security policies. We are aware that the average smartphone user is not IT-minded. Specifying security profiles and policies could be a daunting task for most of the normal users. Our idea is to have third parties to create security profiles with different levels of security and make them available on the Android Market. Users can then install the security profile that matches their security needs and further customise it if needed.

11. REFERENCES

[1] Android malware steals info from one million phone owners. http://nakedsecurity.sophos.com/2010/07/29/android_malware_steals_info_million_phone_owners/.

[2] Android Project. http://www.android.com.

[3] Gartner says android to command nearly half of worldwide smartphone operating system market by year-end 2012.

http://www.gartner.com/it/page.jsp?id=1622614.

[4] Mobile app malware menace grows. http://www.theregister.co.uk/2011/08/04/mobile_malware_trends/.

[5] These 26 Android Apps Will Steal Your Phone's Information. http://www.businessinsider.com/up_to_120000_android_phones_have_been_infected_with_malware_2011_5.

[6] Unisys establishes a bring your own device (byod) policy. http://www.insecureaboutsecurity.com/2011/03/14/unisys_establishes_a_bring_your_own_device_byod_policy/.

[7] Worldwide smartphone market expected to grow 55of one billion in 2015. http://www.idc.com/getdoc.jsp?containerId=prUS22871611.

[8] Guangdong Bai, Liang Gu, Tao Feng, Yao Guo, and Xiangqun Chen. Context-aware usage control for android. In *Proc. SecureComm 2010*, pages 326–343, 2010.

[9] Alastair R Beresford, Andrew Rice, and Nicholas Skehin. MockDroid: trading privacy for application functionality on smartphones. In *Proc. HotMobile '11*, 2011.

[10] Jeffrey Bickford, Ryan O'Hare, Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Rootkits on smart phones: Attacks, implications and opportunities. In *Proceedings of HotMobile 2010*, 2010.

[11] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical report, Technische Universität Darmstadt, D-64293 Darmstadt, Germany, June 2011. Available at: http://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/PubsPDF/xmandroid.pdf.

[12] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 51–62, 2011.

[13] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: context-related policy enforcement for android. In *Proceedings of the 13th international conference on Information security*, ISC'10, pages 331–345, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th international conference on Information security*, ISC'10, pages 346–360, 2011.

[15] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, 2011.

[16] Technische Universitat Dresden and University of Technology Berlin. L4android.

[17] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of OSDI 2010*, October 2010.

[18] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[19] Nancy Gohring. VMWare Shows off Mobile Virtualization on Android. Internet Article, February 2011.

[20] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for": Retroffiting android to protect data from imperious applications. In *18th ACM Conference on Computer and Communications Security (CCS'11)*, CCS 2011, 2011.

[21] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 39–50, New York, NY, USA, 2011. ACM.

[22] Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These aren't the permissions you're looking, 2010. Available at: http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf.

[23] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proc. ASIACCS '10*, pages 328–332, 2010.

[24] Machigar Ongtang, Stephen McLaughlin, William Enck, , and Patrick McDaniel. Semantically rich application-centric security in android. In *Proc. ACSAC '09*, pages 73–82, 2009.

[25] SYBASE White Paper. Are Your Sales Reps Missing Important Sales Opportunities? http://m.sybase.com/files/White_Papers/Solutions_SAP_Reps.pdf.

[26] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, 2010.

[27] Giovanni Russello, Bruno Crispo, Earlence Fernandes, and Yuri Zhauniarovich. Yaase: Yet another android security extension. In *SocialCom/PASSAT*, pages 1033–1040. IEEE, 2011.

[28] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings of the 18th Annual Network & Distributed System Security Symposium*, NDSS '11, pages 17–33, 2011.

[29] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google android: A comprehensive security assessment. *IEEE Security and Privacy*, 8:35–44, 2010.

[30] Yang Xu, Felix Bruns, Elizabeth Gonzalez, Shadi Traboulsi, Klaus Mott, and Attila Bilgic. Performance evaluation of para-virtualization on modern mobile phone platform. In *Proceedings of the International Conference on Computer, Electrical, and Systems Science, and Engineering*, 2010.

[31] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and V.W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proc. TRUST 2011*, 2011.