CRePE: Context-Related Policy Enforcement for Android*

Mauro Conti[1], Vu Thien Nga Nguyen[2], and Bruno Crispo[3]

[1] Vrije Universiteit Amsterdam, NL - email: mconti@few.vu.nl
[2] Universiteit van Amsterdam, NL - email: V.T.N.Nguyen@student.uva.nl
[3] University of Trento, IT - email: crispo@dit.unitn.it

Abstract. Most of the research work for enforcing security policies on smartphones considered coarse-grained policies, e.g. either to allow an application to run or not. In this paper we present CRePE, the first system that is able to enforce fine-grained policies, e.g. that vary while an application is running, that also depend on the context of the smartphone. A context can be defined by the status of some variables (e.g. location, time, temperature, noise, and light), the presence of other devices, a particular interaction between the user and the smartphone, or a combination of these. CRePE allows context-related policies to be defined either by the user or by trusted third parties. Depending on the authorization, third parties can set a policy on a smartphone at any moment or just when the phone is within a particular context, e.g. within a building, or a plane.

Keywords Android Security, Context Policy, Policy Enforcement

1 Introduction

In the world there are almost four billion mobile phones. In developed countries there is almost one mobile telephone subscriber for each inhabitant—one every two inhabitants for developing countries. The computational power of mobile phone devices is continuously increasing leading to smartphones. Smartphones (also just "phones" in this paper) can actually run applications in such a way that is similar to how desktop computers do. However, because of the specific characteristics of smartphones, the security and privacy of these devices are particularly challenging [10]. These challenges reduce the users' confidence and make it more difficult to adopt this technology to its full potential. To remove this difficulty, researchers have focused on improving security models for smartphones.

One significant aspect in security for smartphones is to control the behaviour of applications and services (e.g. WiFi or Bluetooth). In current mobile systems, this control is mostly based on policies per application, and policies are set only

* The work of this paper is partly supported by the project S-MOBILE, contract VIT.7627 funded by STW - Sentinels, The Netherlands. The work of the third author is partially funded by the EU project MASTER contract no. FP7-216917.

at installation time. For instance, in J2ME each MIDlet suite uses a JAD (Java Application Descriptor) file to specify its dependence on requiring certain permissions [5]. The JAD file represents a MIDlet suite. It provides the device at installation time with access control information, for the particular operations required by the MIDlet suite. Similarly, in Android [2], the application's developer declares in a manifest file all the permissions that the application must have in order to access protected parts of the API, and to interact with other applications. At installation time, these permissions are granted to the application based on its signature and the interaction with the user [11]. While Android gives more flexibility than J2ME (the user is notified about resources that the application uses), granting permissions all-at-once and only at installation time is still a coarse-grained control: the user has no ability to govern how the permissions are exercised after the installation. As an example, Android does not allow policies that grant access to a resource only for a fixed number of times or only under some particular circumstances. Meanwhile, to protect users' privacy, the current security models restrict trusted third parties' control on mobile phones. Typically, only the device manufacturer and the telephone company have a small control on the smartphone. There are no mechanisms to allow other authorized parties (e.g. a government agency or a company that bought a smartphone for its employee) to have any direct control on the phone.

One way to extend the control of users and trusted third parties on smartphones is to use context-related policies. A security policy can be defined as a statement that partitions the states of the system into a set of authorized (secure) states and a set of unauthorized (insecure) states. A context-related policy is a security policy which enforcing requires the awareness of the context of the phone. The context can be defined by the status of different variables (e.g. location, time, temperature, noise, light), the presence of other devices, a particular interaction between the user and the phone, or a combination of these.

The following are some application examples of context-related policies:

- A user might want his Bluetooth interface to be discovered when he is at home or in his office, not otherwise (e.g. not when he is traveling by train).
- A user lends his phone to a friend while the user does not want his friend to be able to use some applications or to have certain data available (e.g. SMSs). An appropriate context, *friend-using*, could be set manually or transparently recognizing the actual user.
- A company might have the rule that employees' smartphones can run only a restricted set of applications while employees are working. The context could also be activated in a remote way through a message sent to the phone by the company.

We observe that these kind of policies require some features that are not available in current security mechanism for smartphones: fine-grained and dynamic context definition, platform and application independence, and trusted party verification.

Contribution. In this paper we propose, to the best of our knowledge, the first fine-grained context-related user's policy enforcement solution (architecture and

complete implementation) for Android Smartphones: CRePE. While the concept of context-related access control is not new, this is the first work that brings this concept into the smartphones environment, where the characteristics of these devices (e.g. high mobility of the device, energy and computation constraints) make it particularly challenging. We underline that our concept of context activation differs from the simple one already existing in smartphones, like the "flight mode" or the "silent mode". In fact, these existing contexts can only be manually activated by the user—we also consider and extend the set of such type contexts. However, the main characteristic that our contexts have is that they can also be automatically detected and activated by CRePE (e.g. a new context is detected and activated when the device enters into a given geographical region). Furthermore, CRePE not only enforce policies at run-time but also allows trusted third parties (e.g. authorized government agencies) to define and activate policies at run time (e.g. through a SMS with the appropriate semantic). The implemented solution enjoys a small overhead both in terms of time and energy consumption—thus showing the feasibility of CRePE.

Roadmap. In Section 2 we give an overview of the related work in the area. Section 3 gives an overview of the current Android system. In Section 4 we present CRePE, our solution for fine-grained context-related policies enforcing. Section 5 discusses the evaluation of the proposal. Finally, in Section 6 we give some concluding remarks.

2 Related Work

The diffusion of smartphones is raising attention to the lack of security in these systems. In the last few years, companies have joined together in alliances and projects that aim to produce secure and usable mobile devices. Examples of these projects are OpenMoko [20] and OMTP [18]. On the other hand, the research community has been called to investigate secure solutions for smartphones. Most of the research effort so far has focused on mechanisms to let the system run only certified applications, or to check the permissions an application requests at the installation time [10,17,24]. As an example, the Java MIDP 2.0 security model restricts the use of sensitive permission (e.g. network access) depending on the protection domain the application belongs to [17]. Similarly, the Symbian system gives different permissions to Symbian-signed programs [24]. These type of solutions solve the problem they are created for with mainly the drawback of the induced overhead, recently solved in [10]. In [10] the authors proposed a lightweight mobile phone application certification for Android [11]. Hence, it seems that the problem of filtering applications at installation time has been efficiently and effectively solved.

However, less conclusive results have been obtained for enforcing security at application run time. Some solutions for Linux phones have been proposed leveraging the Linux Security Module (LSM) [26]. Solutions for Windows Mobile has been proposed leveraging either the security-by-contract concept [8,21] or the system call interposition [4]. Unfortunately, none of these solutions al-

low the user to define run time fine-grained context-related security policies. Fine-grained policy enforcement have been investigated outside the domain of mobile phones [13,25]. However, because of the induced overhead, these solutions cannot be adopted for smartphones. In [14], the authors started investigating security policy management for handheld devices. Enforcement of policies related to digital right management has also been proposed [7,16].

A recent work that considered the enforcement of fine-grained policies of the smartphone is [19]. In [19], the authors propose Saint, an install and run-time application management system for Android [2]. The authors start from this observation: Android protects the phone from applications that act maliciously, but provides severely limited infrastructure for installed applications to protect themselves. Leveraging on this observation, the authors built Saint in order to allow Android to be able to enforce application policies that allow an application A: to define which application can access A's interfaces, to define how other application use A's interface, to select at run time if using interface of B or C. While Saint [19] focuses on application policies, CRePE aims to enforce fine-grained context-related policies defined by the user (or other parties). Some envisaged ideas on the possibility to enforce context-related user policy on smartphones were presented in [15], while a comprehensive security assessment for Android can be found in [23].

Researchers have already shown interest in access control depending on the context, even if the meaning of "context" can be very different among these researchers ([22,3,12], to cite a few). For example, in [22] an access control system for active spaces (e.g. a room with heterogeneous computing and communication devices) is proposed, considering the context as different modes of cooperation between groups of users (e.g. being in the room for a meeting). In other works, a context is intended to describe the expected ways the user uses a service [12,3] with the purpose of not granting access if non usual behaviour is shown. The concept of context that we consider in our work is similar to the one of "environment roles" used in [6] to extend the RBAC model to also include environment conditions. However, while these works aim to extend the RBAC model, our aim is to extend the permission checking of Android (that is a modified Mandatory Access Control, MAC, [11]), with the purpose of implementing context-related policy enforcement at run-time.

3 Android Overview

Android is a Linux-based mobile platform developed by the Open Handset Alliance (OHA) [2]. Its increasing adoption by manufacturers, developers, users, and researchers [10,11,19] lead us to select it as our reference platform. In this section we give an overview of the Android architecture and its security model. We refer the reader to [2,11] for further details.

Most of the Android applications are programmed in Java and compiled into a custom byte-code (DEX) 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 developed using pre-defined components: Activity, that represents a user interface; Service, that executes background processes; Broadcast Receiver, a mailbox for communication between application; Content Provider, to store and share application's data. Application components communicate through messages (Intents). Android Inter-Component Communication (ICC) is similar to the Inter-Process Communication (IPC) in Unix-based systems. However, ICC happens identically no matter whether the target component belongs to the same application or not [10].

Focusing on security, Android combines two levels of enforcement [11,23]: at Linux system level, and application framework level. At the Linux system level, Android is a multi-process system: each application runs in its own process and address space. The Android security model resembles a multi-user server, rather than the sandbox model found on J2ME or Blackberry platforms. Each application package, after installed, is assigned a unique Linux user ID which remains constant. That prevents other applications from intervening in its operation except by required permissions which are explicitly declared. At the application framework level, Android provides fine control through Inter-Component Communication (ICC) reference monitor. The reference monitor provides Mandatory Access Control (MAC) enforcement on how applications access the components. To make use of protected features, an application must declare the required permissions in its package manifest definition. As an example, if an application needs to monitor incoming SMSs, the 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. There are multiple ways in which a permission is granted on the caller side: (i) by the package installer; (ii) based on signature checks—the signature of the application that defines the permission should be the same as the application that asks for the permission; (iii) based on interaction with the user. Conversely, on the callee side, an application may restrict access to its components to protect its resources. In this case, the package manifest must define the permissions to protect application components. Each permission definition specifies a protection level which can be: normal (automatically granted), dangerous (requires 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). Protected features of the Android system includes protected application components and system services (e.g. the Bluetooth). Permission enforcement happens at different time during the program operation, i.e. when: invoking a system service; starting an Activity; starting or binding a Service; sending and receiving a message; accessing a Content Provider.

We observe that the current Android security model cannot serve our purpose of enforcing fine-grained context-related security policies. In fact, there are no mechanisms either to enforce or to change policies at application run-time.

4 The CRePE System

In this section we present CRePE (Context-Related Policy Enforcing), the first system that allows smartphones to enforce fine-grained context-related security policies.

4.1 Definitions

Here, we provide some definitions used in the remaining part of the paper.

Definition 1. ***Rule.*** *A rule R=`(<r>,<access>)` describes the access for a resource r, where `<access> ::=allowed|denied|any`. The value `any` is the default value, used if not otherwise specified. If all the policies specify `any` for a resource r, then accessing r is allowed. Otherwise, rules with `any` are not considered.*

For example, a rule R may allow access to the camera: R=`(CAMERA,allowed)`. The resources considered by CRePE are applications and system services (e.g. the camera). Hence, the type of access we consider is to execute an application or to use a system service.

Definition 2. ***Conflict.*** *Two rules R_1, R_2 are in conflict if they are defined on the same resource r_1 but with opposite access: R_1=(r_1,`allowed`), R_2=(r_1,`denied`). By Definition 1, conflicts never involve rules with access type `any`.*

Definition 3. ***Winning Rule.*** *Given two rules R_1 and R_2 that are in conflict, we say that R_1 wins over R_2 if the mechanism implemented for conflicts resolution decides to enforce R_1 (and not to enforce R_2).*

The specific mechanism that we implemented in CRePE to resolve conflicts between rules is described in Section 4.3.

Definition 4. ***Policy.*** *A policy P is a set of rules.*

Definition 5. ***Context.*** *A context can be defined by the status of some variables (e.g. location, time, temperature, noise, and light), the presence of other devices, a particular interaction between the user and the smartphone, or a combination of these. A context has one policy associated with it, and one policy can be associated to only one context (one-to-one relation).*

Definition 6. ***Active Context (Active Policy).*** *A context C is said to be active, at a given time t, if the circumstances that the context specifies are verified. (The policy P associated to C is then called active policy.) More than one context can be active at the same time.*

As an example, a context might be defined by a geographical region. Such a context is active when the phone is within that region.

Definition 7. ***Active Rule.*** *A rule R_1 is said to be active iff both of the following conditions are verified: i) $R_1 \in P_1$, where P_1 is an active policy; ii) if $R_1 \in P_1$ is in conflict with $R_2 \in P_2$ (where both P_1 and P_2 are active policies), R_1 wins over R_2, according to the conflict resolution implemented.*

We observe that, as a consequence of Definition 7, at any given time there can be rules that belong to active policies but that are non active.

4.2 Architecture

The idea underlying CRePE is to put the policy enforcement before the Android permission check. Hence, we based the CRePE architecture on a hook before the Android permission check. The intercepted permission requests are handled by the six components of CRePE's architecture: *CRePE PermissionChecker*, *PolicyManager*, *PolicyProvider*, *UserInteractor*, *ContextInteractor*, and *ActionPerformer*. The architecture of CRePE is summarized in Figure 1. We present the role of the CRePE's components by describing the possible interactions between CRePE and the outside world (applications, user, contexts, and third parties).

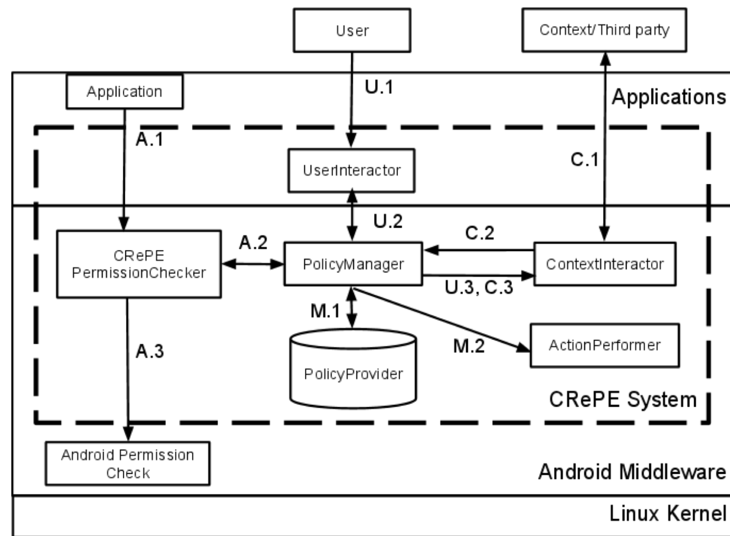

Fig. 1. The CRePE architecture.

When an application sends a request to start another application or to use a system service, the request is intercepted by *CRePE PermissionChecker* (arrow $A.1$, Figure 1). This component interacts with *PolicyManager* (arrow $A.2$) to check if the permission is allowed by the active policies. PolicyManager is the only component that has access (arrow $M.1$) to *PolicyProvider*, that is an archive

that stores all the defined contexts and their corresponding policies, each one defined by a set of rules. PolicyManager also holds the list of active contexts and their policies. If the request is not compliant with these policies, a negative response is returned to the caller. Otherwise, the request is passed on to the Android permission check (arrow $A.3$).

The user can interact with CRePE to create, update, and delete contexts together with their policies (arrow $U.1$). Also, the user can explicitly activate and deactivate a context. The CRePE component that allow this interaction from the user is *UserInteractor*, that is an Android application. When the user creates, deletes or modifies the context information, UserInteractor needs to retrieve and store the information via PolicyManager (arrow $U.2$). PolicyManager then updates the list or contexts to be detected when necessary (arrow $U.3$).

CRePE includes *ContextInteractor*, that is a context detector which informs CRePE when a context becomes active or inactive (arrow $C.1$). To do so, ContextDetector keeps the list of stored contexts needed to be detected. The detected information related to a context (i.e. activating or deactivating a context) is passed on to PolicyManager (arrow $C.2$). Based on this new information, the set of stored contexts can change. In that case, PolicyManager notifies ContextInteractor (arrow $C.3$). When a context is activated or deactivated, ContextInteractor informs the PolicyManager (arrow $C.2$) to update the set of active policies. In a similar way, CRePE also interacts with (authenticated) third parties to get contexts' information and their policies.

Finally, CRePE has a component, *ActionPerformer*, that handles the changes of the set of active policies at run time. We remind that these changes are due to contexts activation and deactivation. In these cases, PolicyManager asks ActionPerformer to perform the necessary actions (arrow $M.2$) to ensure that the policy is enforced also for the resources already assigned to applications (e.g. to stop a running application that is no more allowed).

4.3 Implementation

This section discusses the implementation of each component of CRePE. Being an Android extension, CRePE has been written in Java. Besides some minor changes to the Android code itself, the CRePE implementation consists of 268KB of code.

PolicyProvider. As mentioned in Section 4.2, all the information related to contexts and policies are stored in the PolicyProvider. Similar to the phone address book and the calendar provider, PolicyProvider is a database embedded inside the Android middleware. CRePE implementation uses the default database supported by Android: SQLite. Among the possible ways to define a context, in our prototype implementation we considered only *Time* and *Location*. Behind these, we implemented other attributes to handle the context: *Period*, *Owner*, *Activation*, and *Deactivation*. The meanings of these attributes are explained in Table 1. Conflicts can take place according to Definition 2. The conflict resolution is managed by PolicyManager.

Attribute	Values	Explanation
Time	a time value	This attribute indicates the period of time that the context is active. This can be either a single time interval or a period repeated by day, weekday, weekend, month and year. Note: this attribute is optional.
Location	a location value	This attribute indicates the location where the context is active. Note: this attribute is optional.
Period	{*short*, *long*}	If *long*, the context information and its policy are kept in PolicyProvider when the context is no longer active. If *short*, the context information and its policy are removed from PolicyProvider when the context is no longer active.
Owner	{*user*, *third-party*}	If *user*, the context and its policy are defined by the user. If *third-party*, the context and its policy are defined by a third party. PolicyProvider also stores the identity of the third party.
Activation	{*notified*, *auto*}	If *notified*, the context is activated by a notification from the owner. If *auto*, the context is activated automatically, when appropriate.
Deactivation	{*notified*, *auto*}	If *notified*, the context is deactivated by a notification from the owner. If *auto*, the context is deactivated automatically, when appropriate.

Table 1. Context Attributes.

PolicyManager. PolicyManager is the only component accessing PolicyProvider. Hence, it provides all the possible operations to manipulate stored contexts, policies, and rules. PolicyManager provides operations to check permission to access an application. PolicyManager is in charge to detect and resolve conflicts among policies. Conflict resolution works as follows. Each CRePE rule has a priority level assigned to it; we consider three possible priority levels (in decreasing order of priority): *government*, *trusted party*, and *user*. First, if the conflicting rules belong to different priority levels, the higher priority is the one enforced. In case conflicts exist at the same priority level, the more restrictive rule wins (we consider the access type `denied` as being more restrictive than `allowed`). Finally, note that the current version of CRePE does not handle automatic application restarting (not explicitly defined throughout policies).

UserInteractor. This is an application that allows the user to manage contexts, policies, and rules information. CRePE does not allow the user to define a context that has the values of all the variables (location and time) identical to the values of an already defined context. This restriction applies only to auto-activated contexts, while it does not apply to contexts activated by the user or third parties. In fact, while two auto activated contexts defined by the same variables would be necessarily both active or inactive—not justifying the existence of two contexts—this could not be the case for context explicitly activated by the user or third parties.

ContextInteractor. ContextInteractor plays the role of context detector. The current implementation of CRePE supports two physical attributes for automatic activation: time and location. For the time attribute, the context detector is the timer in Android. For the location attribute, ContextInteractor uses the GPS to detect the location of the phone. In case the context has the value *notified* for both activated and deactivated, only the context's owner can activate or deactivate the context. Context's owner can communicate to the phone via Blue-

tooth, WiFi or SMS. Third parties can also add, change and remove contexts and policies through these kinds of communication. While not yet implemented in the current implementation, the message can include a signature to authenticate the sender. In the current implementation we assume the network provider being trusted, hence authenticating the third party just by its phone number.

ActionPerformer. As mentioned above, this component performs necessary actions within the phone when the policy enforcement requires them. This component takes action in two cases: (i) when a policy that disallows access to a resource currently used becomes active; (ii) when a policy asks explicitly to take an action (e.g. turn off the phone). In the case of (i), if the resource is a system service, ActionPerformer disables that system service. If the resource is an application that is already running, PolicyManager stop the application.

The CRePE PermissionChecker. CRePE governs access to resources such as applications and system services. When an application invokes a system service, Android checks the access permission within the method `checkPermission` of the class `ActivityManagerService`. Hence, the CRePE PermissionChecker is invoked with a hook before the method `checkPermission`. Access to an application is mostly accessing its Activity components. Then, CRePE concentrates on controlling access to Activity. In Android, the security is enforced before starting an Activity. The caller component requests to start a new Activity through the method `startActivityForResult`. The class `ActivityManagerService` receives that request and calls its method `startActivity`. This method sets up the required environment parameters and checks if the caller is allowed to start the activity. Since the callee is the only one who knows who is allowed to start it, the permission check is performed at the callee site. However, in CRePE the policy information is stored in the database and can be retrieved at any time. Therefore, for efficiency reasons (avoiding computation and time overhead) CRePE can place a hook directly at the caller site (method `startActivityForResult` in the class `Activity`). PermissionChecker checks if the requesting application is allowed to access the resource based on the active policies. The set of active policies is provided by the PolicyManager.

5 System Evaluation

In this section, we describe how a practical application works in CRePE (Section 5.1). Hence, we discuss the CRePE security (Section 5.2) and its overhead (Section 5.3).

5.1 A Running Example

This section presents how CRePE works in a scenario cited in Section 1. We consider the example of a company that wants to restrict the set of applications that can run, during work activities, on the smartphones that the company has

given to its employees. In this example, the context `during-work` is defined by the location of the company's buildings and the time of working hours (e.g. from 9 to 17), while the Activation parameter is set to `auto`. Policy information includes rules that only allow access to the restricted set of applications. The company sets the context and the associated policy through a message sent to the employer's phone and signed with the company's private key. The context and policy information is passed on to PolicyManager and stored in PolicyProvider. When the context is detected ContextInteractor notifies PolicyManager to activate the context. PolicyManager checks if any application outside the restricted set is currently used and, if any, informs ActionPerformer to stop it. While the context remains active, any request to start applications outside the set of the allowed ones are checked by CRePEPermissionCheck and denied by PolicyManager. When the phone is out of the context, ContextInteraction informs PolicyManager. Requests to start other applications are then allowed.

We observe that the description of this example can be adapted to any similar policy that the company wants to enforce on the phones: (i) dynamically—not just all-at-once before the phone is given to the employee; (ii) and even remotely—through sending messages to the phone, without requiring the physical presence of the phone in a particular place.

5.2 Security

First, we observe that CRePE does not reduce the Android security. For each requested access to a application or system service, CRePE only introduces further checks—its own checks depending on the active policies. However, each access that is not denied by CRePE is passed on to the Android Permission Check and not influenced by CRePE anymore. As a result, CRePE can only reduce the number of accesses allowed, not reducing the security.

Furthermore, we observe that the current delegation mechanism of Android has a weakness that CRePE fixes to some extent. In particular we consider the following to be a weakness. An application *App* is allowed to access a resource (e.g. to use the Bluetooth service). *App* defines a permission *Perm* for its component App_1 (that actually uses the resource). *App* defines *Perm* with *normal* level which is automatically granted without asking for explicit approval from the user. This would imply that any other application can use the same referred resource, while the user is not actually aware of this. To some extent, CRePE helps to prevent this kind of compromise. A CRePE policy could be defined by the user to limit the access to resources in some necessary situations. For instance, the user can define a policy allowing to use Bluetooth only at home or the office which are trusted environments.

Finally, we underline that an adversary (either a user or an application) cannot skip the CRePE enforcement. We remind that CRePE is designed as an extension of Android, then it will run with the privileges of the Android Middleware. The only part of CRePE outside the middleware is UserInteractor. UserInteractor is the only application CRePE trusts and the user must be authenticated to use it. The adversary cannot influence the context activation

since the values considered for this operation (e.g. current time) are taken directly from CRePE using system service drivers. In order to avoid the adversary modifying the operating system of the phone (drivers and CRePE included) Trusted Computing mechanisms leveraging Trusted Platform Module (TPM) can be used. However, the discussion of these mechanisms is outside the scope of this paper.

5.3 Overhead

In this section, we report on experimental results we conducted in order to evaluate the performances of CRePE. In particular, we evaluated both time and energy overhead for the different features of CRePE—time performance and energy consumption are two main issues of smartphones. The experiments were conducted using a HTC Magic smartphone. In particular, we used its developers version (Android Dev Phone 2 [1]), featuring it an unlocked bootloader that we needed to install our custom system image of Android, that is CRePE. We identified two main characteristics of CRePE that induces overhead compared to Android: (i) calling the CRePE PermissionChecker before the Android Permission Check; (ii) determining context changes through ContextInteractor.

Let us start discussing the overhead induced by (i). We observe that this is generated for each request that is intercepted by CRePE. We ran experiments to understand the amount of overhead induced by the CRePE permission check in terms of both time and energy consumption. As for the time overhead we measured the time induced by the CRePE checks. As mentioned previously, the CRePE check is done through a hook before the Android permission check. We measured the time interval between the request of a resource (application or system service) and the moment that the request is starts being checked by Android. We considered request of access for both applications and system services: we did not experience differences between these two measures. In fact, both requests are processed in the same way by CRePE. Furthermore, during the experiments we noticed that the specific resource the rules are intended for, do not influence the results. Figure 2 shows the results we obtained. In particular, the graph shows the time overhead measured (y-axis) while varying the number of active rules (x-axis). We plotted the average obtained from 100 measurements. Overall, we observe that the time overhead of CRePE permission check is negligible—all the results are under 0.6 ms. Within this small time overhead, we observe that the time overhead increases while increasing the number of rules.

Since the permission check is the most common action of CRePE and energy consumption is one of the most important issues of today's smartphones, we also investigate the energy overhead induced by CRePE for its permission check. To investigate this aspect we ran an experiment with a high energy demand setting and high involvement of CRePE. In particular, we wrote an application that every ten minutes placed a phone call of 165 seconds (a free call to an automatic service). We let this application run for two hours starting the experiments with a fully charged battery. We ran the experiments 10 times for each of the two following cases: phone running Android; phone running CRePE. In the CRePE

case, we repeated the experiment for a different number of active rules. Again, we did not experience any difference for the specific resources the rules are for. The results of the experiments are shown in Figure 3.

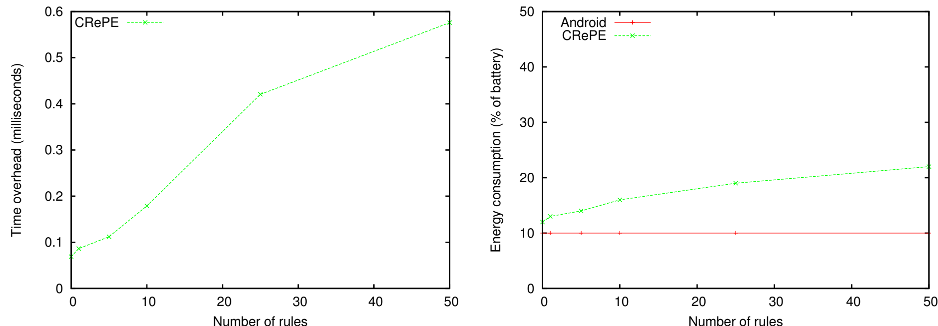

Fig. 2. Permission check: time overhead.

Fig. 3. Permission check: energy overhead.

From Figure 3 we observe that, as expected, the energy consumption of CRePE is higher than the original Android. This is due to the energy consumption due to the CRePE permission checking. In particular, during this experiment 2721 permission checks were called for 19 different resources (mostly for `DEVICE_POWER` permission, 1407 times, and `WAKE_LOCK` permission, 588 times). Furthermore, as also expected, the energy consumption of CRePE increases while the number of active rules increases. The energy overhead induced by CRePE is some 50% of the Android consumption when 10 rules are set, while it becomes more than 100% of the Android's one when the rules are 50. Interestingly, the energy consumption does not increase linearly with the number of active rules. This is due to some basic action of CRePE that does not depend on the number of rule set. During the experiment, we also varied the set of active rules for a given size of this set. We observed that the results do not depend on the specific rules that are active. In fact, for each invocation of CRePE, it checks the requested permission comparing it with each of the active rule—the overhead depends only on the number of rules, not on their specific nature. From this experiment, we can conclude that the energy consumption of the permission checking is not negligible. However, we underline that having 50 active rules might not be common and even in this case, while energy consuming, it shows that the CRePE solution is still feasible. We also underline that this is the first implementation that did not pay particular attention to optimizations.

As for (ii), the overhead depends mainly on how much the system is desired to be responsive to context changes. In fact, the sooner we want to detect a context change, the higher the context checking frequency will be—hence, higher the overhead. This is true for the context features that need to be actively checked by the phone, like the GPS coordinates, while this does not hold for a context explicitly set by the user (like the *friend-using* of Section 1) or by third parties

through messages. The simplest approach is to consider a continuous polling for detecting context changes. We ran experiments to evaluate our current implementation of CRePE—that only leverages GPS and polling. The experiments were conducted without any context active, just to measure the overhead of the GPS polling. We considered the energy consumption observed in 5 consecutive hours, starting with a fully charged battery. We observed that if CRePE requests the current position to the GPS device every 5 minutes, the battery level decreases by 48%, while checking the position every 15 minutes would consume 11% of the battery. The results underline how the energy consumption is one of the main issues in today's solutions for mobile devices. While these results are not negligible, the energy consumption for checking every 15 minutes is quite promising. In fact, in the problem addressed by CRePE, optimizations are possible from this point of view. As an example, the location checking frequency might be decreased while the value of a variable of interest (e.g. location) is far from the value for which the context is activated.

Overall, we observed how the current implementation of CRePE has a reasonable amount of overhead from both the time and energy point of view. Furthermore, some optimizations are possible compared to the current implementation, e.g. the location checking could be optimized leveraging other information through the carrier provider or WiFi interfaces available, like the assisted-GPS works [9].

6 Conclusion

Smartphones are widely used. However, the lack of user possibility to specify fine-grained security policies make it more difficult to adopt this technology to its full potential. An an example, a user might avoid using an application if he does not really trust the application not to send out the user's private information. In this paper, we presented CRePE, a Context-Related Policy Enforcing solution, that is the first system that allows smartphones to enforce fine-grained context-related policies. CRePE allows both the user and authorized third parties to define a fine-grained security policy that depends on the context. Third parties can also ask CRePE to enforce policies remotely, through messages.

References

1. Android-Developers. Android dev phones. `http://developer.android.com/guide/developing/device.html`, retrieved June 30, 2010.
2. Android Project. Android. `http://www.android.com`, retrieved June 30, 2010.
3. Andromaly Project. Andromaly anomaly detaction in android platform. `http://andromaly.wordpress.com/`, retrieved June 30, 2010.
4. M. Becher and R. Hund. Kernel-level interception and applications on windows mobile devices. Technical Report TR-2008-003, Department for Mathematics and Computer Science, University of Mannheim, Germany, 2008.
5. R. C. Steel, R.Nagappan. *Core Security Patterns: Best Practices and Stategies for J2EE, Web Services, and Identity Management.* Prentice Hall, 2005.

6. M. L. Damiani, E. Bertino, B. Catania, and P. Perlasca. Geo-rbac: A spatially aware rbac. *ACM Trans. Inf. Syst. Secur.*, 10(1), 2007.
7. M. T. Dashti, S. K. Nair, and H. Jonker. Nuovo DRM paradiso: Designing a secure, verified, fair exchange drm scheme. *Fundam. Inf.*, 89(4):393–417, 2009.
8. L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. A flexible security architecture to support third-party applications on mobile devices. In *CSAW '07*, pages 19–28, 2007.
9. G. M. Djuknic and R. E. Richton. Geolocation and assisted gps. *Computer*, 34(2):123–125, 2001.
10. W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS '09*, pages 235–245, 2009.
11. W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security and Privacy*, 7(1):50–57, 2009.
12. W. Han, J. Zhang, and X. Yao. Context-sensitive access control model and implementation. In *CIT '05*, pages 757–763, 2005.
13. I. Ion, B. Dragovic, and B. Crispo. Extending the java virtual machine to enforce fine-grained security policies in mobile devices. In *ACSAC '07*, pages 233–242, 2007.
14. W. Jansen, T. Karygiannis, M. Iorga, S. Gravila, and V. Korolev. Security policy management for handheld devices. In *SAM'03*, pages 199–204, 2003.
15. A. Joshi. Providing security and privacy through context and policy driven device control. In *W3C Workshop on Security for Access to Device APIs from the Web*, 2008.
16. S. K. Nair, A. S. Tanenbaum, G. Gheorghe, and B. Crispo. Enforcing DRM policies across applications. In *DRM '08*, pages 87–94, 2008.
17. Nokia Forum. Signed MIDlet Developer's Guide. `http://www.forum.nokia.com`, retrieved June 30, 2010.
18. OMTP Project. OMTP: Open mobile terminal platform. `http://www.omtp.org`, retrieved June 30, 2010.
19. M. Ongtang, S. McLaughlin, W. Enck, , and P. McDaniel. Semantically rich application-centric security in android. In *ACSAC '09*, pages 73–82, 2009.
20. Openmoko Project. Openmoko. `http://www.openmoko.org`, retrieved June 30, 2010.
21. S3MS. Security of Software and Services for Mobile Systems. `http://www.s3ms.org`, retrieved June 30, 2010.
22. G. Sampemane, P. Naldurg, and R. H. Campbell. Access control for active spaces. In *ACSAC '02*, page 343, 2002.
23. A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *IEEE Security and Privacy*, 8:35–44, 2010.
24. Symbian Ltd. Simbian Signed. `https://www.symbiansigned.com`, retrieved June 30, 2010.
25. N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, and D. August. Rifle: An architectural framework for user-centri information-flow security. In *MICRO '04*, pages 243–254, 2004.
26. X. Zhang, O. Aciiçmez, and J.-P. Seifert. A trusted mobile phone reference architecturevia secure kernel. In *STC '07*, pages 7–14, 2007.