Context-related Policy Enforcement for Android

Nguyen Vu Thien Nga
Context-related Policy Enforcement for Android

by

Nguyen Vu Thien Nga

A thesis submitted in partial fulfillment for the degree of Masters of Science in Grid Computing - track Computer Science

in the Faculty of Science Informatics Institute

July 13, 2010

Supervisors

Dr. Bruno Crispo
Dr. Mauro Conti
Dr. Alban Ponse

Examiners

Dr. Inge Bethke
Dr. Bruno Crispo
Dr. Mauro Conti
Dr. Clemens Greleck
Dr. Alban Ponse
"Be brave and kind! If your dream is Big enough, Facts don't count."
Abstract

Most of the research for enforcing security policies on smartphones considers coarse-grained policies, e.g. either allow an application to run or not. In this thesis we present CRePE, the first system that is able to enforce fine-grained and context-related policies on smartphones. These policies vary while an application is running and 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.
Acknowledgements

I would like to express my gratitude to my supervisor, Bruno Crispo, for offering me this challenging research project and for his valuable advices during this project. I am grateful to Mauro Conti for many enjoyable discussions. For continuously correcting my writings and giving me several research skills, I also would like to thank Mauro very much. I am very thankful to Alban Ponse for being my official supervisor and helping me a lot with graduation procedures. I would like to thank to Android engineers for their useful answers about the Android platform in Google Groups.

For advices, and interesting, often funny discussions I would like to thank Aram Visser, Irfan Uddin and Gökhan Ertaylan, and especially many thanks to Aram for proofreading this thesis. All these friends made my time here such an enjoyable time.

Furthermore, I would like to thank Dung Manh Chu for helping me with very first days in Amsterdam. For useful advices and various funny parties and chats, I would like to thank 'chi'¹ Thanh and 'anh'² Duc. Last but not at least, I would like to thank my family for being always besides me.

¹the Vietnamese way to call older female friends
²the Vietnamese way to call older male friends
# Contents

Abstract ii

Acknowledgements i

List of Figures iii

List of Tables iv

1 Introduction 1

1.1 Motivation ................................................................. 1
1.2 State of The Art ......................................................... 3
1.3 Thesis Contribution .................................................... 4
1.4 Thesis Organization ................................................... 5

2 Related Work 6

2.1 Java MIDP 2.0 ............................................................. 6
2.2 Symbian-signed Program .............................................. 8
2.3 Kirin ........................................................................ 8
2.4 Dynamic Mandatory Access Control for Multiple Stakeholders 8
2.5 Security-by-Contract ................................................... 9
2.6 Saint ........................................................................ 10
2.7 Summary .................................................................... 11

3 Android Overview 12

3.1 Android Architecture .................................................. 13
3.1.1 Linux Kernel Layer ................................................ 13
3.1.2 Libraries and Android Runtime ............................... 13
3.1.3 Application Framework .......................................... 14
3.1.4 Applications ........................................................ 15
3.2 Android Security .......................................................... 16
3.2.1 Security Mechanisms ............................................. 16
3.2.2 Security Enforcement ........................................... 17

4 CRePE Design 18

4.1 Definitions ................................................................. 18
4.2 CRePE Overview ........................................................ 20
4.3 CRePE Architecture .................................................... 20
List of Figures

3.1 Overview of the Android Stack. ........................................ 13
3.2 Typical Inter-Component Communication. .......................... 15
4.1 The CRePE architecture. ............................................. 22
5.1 Relational diagram in Policy Provider. .............................. 31
5.2 The user decides for optional rules. ................................. 33
5.3 Deny Access. ......................................................... 34
5.4 A running application is stopped because of breaking the active rules. ................................. 35
5.5 Switching off the flight mode. ...................................... 36
5.6 User Interactor Forms. ............................................. 41
5.7 Two examples of CRePE SMS message. ............................ 45
6.1 Time overhead for CRePE Permission Check. ..................... 48
6.2 Energy overhead for GPS locating. ................................ 49
List of Tables

4.1 Context Attributes. ................................................................. 24
4.2 Rule Attributes. ................................................................. 24
4.3 Conflict Resolution. ............................................................. 25
4.4 Location Determination. ....................................................... 27
4.5 Physical Domain. ............................................................... 27
4.6 State Changes of Contexts. .................................................... 28

5.1 Entrance and exit events. ....................................................... 37
5.2 CRRePE Message Syntax. ....................................................... 43

6.1 Average time for context activation and deactivation. .................... 50
6.2 Time overhead for trusted party interactions. ............................. 51
6.3 Response time for user interactions. ........................................ 52
Chapter 1

Introduction

As smartphones have become ubiquitous and people need to carry with them everyday, their security and privacy are challenged. Researchers have therefore focused on improving security models for smartphones. One significant aspect in security models for smartphones is to control the access to applications and phone resources such as WiFi or Bluetooth. One example is that the user can choose in which situations Bluetooth is used. This work aims to extend the control of the user and trusted parties over smartphones by using a particular type of fine-grained security policies. These policies depend on the current device context—a context being defined by the device surroundings; for example, location, time, noise, presence of the owner, or even a particular interaction between the user and the device.

Recently, Android [1] has emerged as an ideal platform for mobile phones. Android is tightly integrated up to even the operating system services and also how they interact with the Application Programming Interfaces (APIs). That helps to develop applications faster. Also, Android supports the developer with APIs related to low-level features. Moreover, the Android security architecture dominates other mobile security architectures with explicit fine-grained resource permissions. With these advantages, Android has lately prevailed for mobile platforms.

For these reasons, this work aims to create a new mobile platform Context-related Policy Enforcement (CRePE), which is an extension of Android. This new platform supports the security policies, which depend on the phone context.

1.1 Motivation

Recently, smartphones have become more and more popular. According to Applied market intelligence Company, iSuppli [2], smartphone sales increased 11.1% in 2009 though mobile
handsets had a big fall in overall sales. One reason for this trend is the fact that the computational power of mobile phone devices is continuously increasing. A smartphone is defined by its high-level operating system, including Apple’s iPhone OS, Microsoft’s Windows Mobile, Nokia’s Symbian, BlackBerry OS, Google’s Android, Palm OS and other Linux-based systems. Thanks to these operating systems, smartphones can actually run applications in such a way that is similar to how desktop or laptop computers do. Taking advantage of small size together with being well equipped with several functionalities (e.g. WiFi, Bluetooth, Camera) and applications (e.g. Calendar, Email), smartphones are now so essential that people want to carry with them everyday. Consequently, smartphones’ security and privacy are particularly challenged. Smartphones currently have to deal with various types of malwares [3]. These threats reduce the user confidence and put a barrier on the adoption of this technology in its full potential. As a result, several research projects have focused on improving security models for smartphones.

To improve security of smartphones, this work focuses on extending the control of the user and trusted parties over the access to applications and phone resources (e.g. WiFi or Bluetooth). These new controls are supported by context-related policies. In the mobile phone’s aspect, a security policy addresses constraints on functions over the phone and constraints on access by external systems or by the user. A context-related policy is a security policy which enforcing requires the awareness of the phone context. The context can be defined by the status of different variables like time, location, presence of other devices, presence of the user, 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 or bus).

- A user lends his phone to a friend while the user does not want his friend to use some applications or to see some personal data (e.g. SMS messages, phone numbers and files). A particular context $friend$-$using$ could be used to protect these applications and data—the context could be set manually or transparently recognizing the actual owner.

- A company might have the rule that only a restricted set of applications can run during office hours within the company’s buildings.

- A building might have a policy that does not allow smartphones to take pictures inside it (without completely forbidding the use of smartphones), or that phones must turn off the sound inside meeting rooms.
1.2 State of The Art

As mentioned above, this work aims at extending the control of the user and trusted parties over the access to phone resources and applications. In current mobile platforms, access control is mostly based on policies per application, and policies are set only at installation time. For instance, in Java 2 Micro Edition (J2ME) [4], each application package, MIDlet suite, uses a JAD (Java Application Descriptor) file to specify its dependence on requiring certain permissions [5]. A 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 [1], the application developer declares in a manifest file all the permissions that the application must have in order to access protected APIs and interact with other applications. At installation time, these permissions are granted to the application based on its signature and the interaction with the user [6]. The user finally decides to allow the application to use each single required resource. Indeed, Android gives more flexibility than J2ME: the user is notified about resources that the application uses. However, granting permissions all-at-once and only at installation time is a coarse-grained control. Although the user can choose either to grant or not grant a specific permission, he still has no ability to govern how they are later exercised. As an example, Android does not support policies that grant access to a resource only for a fixed number of times or only under some particular circumstances.

Both J2ME and Android assume that the phone’s current user is always the phone’s owner. This prevents the real owner from protecting his privacy. With current mobile systems, the owner has only two choices: allow someone to access all phone resources, including all phone resources and applications; or disallow anyone to access the phone by locking the phone with password. That is again a coarse-grained policy.

Meanwhile, to protect the user’s 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 is no mechanism to allow other authorized parties (e.g. a government agency or a company that bought smartphones for its employees) to have any direct control on the phone.

We observe that enforcing context-related policies requires some features that are not available in current security models:

- Dynamic context definition. Current security models only provide some pre-defined contexts that depend only on the time (e.g. a user can set the phone to be powered off after 11PM). However, the user cannot define new contexts depending on several elements such as time, location, and the presence of the user or even virtual contexts which are manually set like the example of friend-using.
• Fine-grained policies. Current security models do not support fine-grained policies which allow access to a specific set of phone resources and applications. Usually, they support phone locking by password. However, as discussed above, that is coarse-grained policy: access to all phone resources and applications or not at all.

• Platform and application independence. Currently control over phone resources and applications depends much on the resources and applications themselves. That means, how much control an application (or a phone resource) provides the user depends on the application (or phone resource) designer. For example, if a company wants employees to use only some applications at work, it may ask the application providers to modify application control. However, this would be awkward and quite unpractical (e.g. too costly).

• Trusted party verification. Current models concentrate mostly on the user control. Other trusted third parties hardly have control over the mobile phones and they do not investigate the verification of trusted parties.

In summary, current mobile platforms, in which access control is defined for each application separately and is granted at installation time, cannot support context-related policies. This work therefore proposes a new Android-based platform which supports context-related policies. We focus our efforts on Android platform, because it: (i) supports developers with low-level features, (ii) supports fine-grained security permissions and bundles this information with the application, and (iii) is open source.

1.3 Thesis Contribution

This thesis presents a solution to enforce fine-grained context-related policies in smartphones. To the best of our knowledge, this is the first system that is able to enforce such a type of policies. In particular, we design and implement our solution for the Android operating system. The new platform, Context-Related Policy Enforcement (CRePE), can handle both of the following context-related policies: (i) policies depended on the context of the phone, and (ii) policies that come from a third-party when the phone is within a particular context. The implementation, together with the limited overhead observed, confirm the validity of the design and the effectiveness of the proposed solution.

In addition, the result of this work has been summarized in a scientific paper submitted to a security conference. A preliminary version of the paper is already available as a technical report in the Computer Systems group, Department of Computer Science, VU University Amsterdam [7]. Further information can be also found on the project website [8].
1.4 Thesis Organization

This section explains the organization of the rest of this thesis. Chapter 2 presents some research related to our work. That helps to provide an overview about security in mobile phone. Since our system is based on the Android platform, Chapter 3 gives some general ideas about Android and security in Android. This chapter focuses on security enforcement. All information in Chapter 3 supplies some backgrounds for the following chapters. Chapter 4 explains all the requirements of the CRePE system. Based on the requirement analysis, this chapter also exhibits the designs of all components in the CRePE system. After that, Chapter 5 presents our detailed implementation of all CRePE components. Chapter 6 shows the evaluation of CRePE. The evaluation includes the security discussion and the overhead discussion. Also, some experiment results are provided in this chapter. Finally, Chapter 7 presents some conclusions and discusses some future works.
Chapter 2

Related Work

The spreading diffusion of smartphones is raising attention to the lack of security in these systems. The research community has investigated secure solutions for smartphones. Most of the research effort spent so far focused on mechanisms to let the system only run certified applications, or to check the permissions an application requests only at installation time [3, 9, 10]. As an example, Java MIDP 2.0 security model restricts the use of sensitive permissions (e.g. network access) depending on the protection domain to which the application belongs [9]. Similarly, the Symbian system gives different permissions to Symbian-signed programs [10]. These types of approaches solve the problem they are thought for with few drawbacks, mainly in terms of overhead. Furthermore, Kirin security service [3] has recently been proposed for Android to perform lightweight mobile phone application certification at installation time. Hence, it seems there are now efficient and effective solutions to filter applications at installation time.

Meanwhile, research has been focused on security enforcement at application run time. Some solutions for Security-enhanced Linux (SELinux) phones have been proposed for the Linux Security Module (LSM). One example is Dynamic Mandatory Access Control for Multiple Stakeholders [11]. One solution for Windows Mobile has been proposed using the security-by-contract concept [12]. Recently, Android has emerged as a promising mobile platform and its openness inspires for several security research. One solution to improve Android security is Saint infrastructure [13], which enforces policies based on both permission declarations at installation time and the status of the system at run time.

2.1 Java MIDP 2.0

Mobile Information Device Profile (MIDP) [14] is a specification published for the use of Java on embedded devices such as mobile phones and PDAs. MIDP is called an open platform
enabling anybody to create software for MIDP devices. However, accessing to sensitive APIs are not open to all applications. In MIDP, each application is represented by a MIDlet suite. Each MIDlet suite includes a policy profile that declares all permissions needed for the application [9]. These permissions are considered at installation time.

MIDP introduces two concepts: trusted MIDlets and untrusted MIDlets. A MIDlet is trusted if the device can verify the origin and integrity of its JAR file. Otherwise it is untrusted. A trusted MIDlet suite can obtain permissions automatically, depending on the security declaration at installation time. On the other hand, an untrusted MIDlet suite requires user approval to access restricted APIs which are declared in policy profile at installation time.

In MIDP 2.0, each application belongs to a protection domain that is defined as a set of permissions and interaction modes. A permission can be either allowed which is granted automatically or user which is deferred until user approval. The user can either deny or allow user permissions. For user permissions, there are three interaction modes that describe how they can be granted:

- Blanket. The MIDlet suite can acquire the permission automatically, unless the user explicitly revokes the permission.
- Session. The user needs to authorize for the first time the permission is invoked.
- Oneshot. The permission requests the user to approve every time it is invoked.

Using MIDP 2.0, the device includes some predefined protection domains and installing a MIDlet requires to assign it to one of these domains. As a result, the MIDlet’s permissions should be a subset of the protection domain’s permissions. Each protection domain, except for the untrusted domain, is associated to a set of root certificates. Signing a MIDlet suite requires a public key certificate that can be validated to one of those root certificates. This association is used to assign the MIDlet suite to a protection domain. Each protection domain can be associated to many root certificates while a root certificated is associated to only one domain.

There are four predefined protection domains:

- Manufacturer. This domain uses root certificates belonging to the device producer.
- Operator. This domain uses root certificates of the network operator. These root certificates are usually stored in SIM cards.
- Trusted third party. This domain covers root certificates of well-known Certificate Authorities (CA).
- Untrusted. This domain includes no root certificates and is used for unsigned MIDlet suites.
2.2 Symbian-signed Program

Symbian-signed is the process of generating a digital certificate for an application. The certificate proves the origin of the application by including the identification of the application publisher. Only signed applications can use sensitive features of the platform. An unsigned application is not able to access sensitive functionalities and may not even be installed on the device depending on the security settings defined by the manufacturer. To sign an application, the developer needs to register for a Publisher ID which costs some money. There are some signed options for either application testing (Open Signed Online, Open Signed Offline) or application distributing (Express Signed, Certified Signed) [10].

A Symbian-signed application must meet the criteria defined in the Symbian Signed Test Criteria [10]. The Test Criteria are a set of conditions agreed by the network operators, handset manufacturers, application developers and other industry stakeholders. The purpose of the Test Criteria is to confirm that an application works properly on the specified Symbian phones.

2.3 Kirin

Kirin [3] is a security service for Android and it provides practical lightweight certification of applications at install time. In the Android platform, an application package includes a security profile specifying permissions to access sensitive resources. At install time, the user can either reject the application or approve all permissions. It is often difficult for the user to relate permissions to sensitive resources and real risks. Kirin therefore aims to mitigate malware using a set of predefined security rules.

Kirin certifies an application based on its policy configuration. Certification is issued based on security rules. These rules represent templates of undesirable security properties. A combination of properties can indicate malicious potentials. For example, an application that can start on boot, read geographic location and access the Internet might be a tracker spyware.

At install time, Kirin extracts the application’s security profile and evaluates it against a set of security rules. In case of failure, Kirin has two options: (i) to reject the application; or (ii) to notify the user with a security warning and let the user to decide.

2.4 Dynamic Mandatory Access Control for Multiple Stakeholders

In the SELinux LSM, to administer access control policies, there are two basic approaches: Discretionary Access Control (DAC) and Mandatory Access Control (MAC). Since DAC permits
administration of the device user, important system permissions can be overwritten by the user or processes on behalf of the user. Thus, the system security can be compromised. MAC, in contrast, restricts system permissions for trusted subjects only. Although MAC appears to ensure system security, it is not flexible. MAC requires a single system administrator that knows all permissions for all possible legal runs. Meanwhile, in open cell phone system environments, an application may require access permissions depending on multiple stakeholders, such as the device owner, the device manufacturer, the network operator, etc. Therefore, MAC is too restrictive for dynamic environments like open cell phone with multiple administrators from multiple stakeholders.

For all of the above reasons, the work in [11] introduces a new approach for dynamic MAC supporting multiple stakeholders in a distributed environment. In this approach, each application process is provided a base policy. The base policy consists of all access permissions to those operations granted for all runs. When an access permission outside the base policy is requested, a policy server is invoked to decide whether the operation is permitted or not. This decision is based on the inputs of application’s stakeholders. The policy server is implemented to determine the application’s stakeholders and their administrative decisions regarding the request. The policy server implementation is hierarchical, i.e. the stakeholder’s decisions may be cached at the policy server on the device, a proxy policy server (for example, in the telecommunication networks) or on the stakeholders themselves. The work recommends accepting already known stakeholders and caching their administrative policies on the device. In this case, an updating mechanism should be provided.

2.5 Security-by-Contract

The current security model of MIDP 2.0 is based on trust relationships, i.e. the application can be run only if it is digitally signed by a trusted party. The work in [12] points out two problems of this security model:

- The security model is coarse-grained. The whole security policy can be rejected or accepted. For example, an application from an untrusted party can be denied access to the whole network but it cannot be denied access to some specific domains.

- There is no semantics attached to the signature. This is a problem for both the application producer and consumer. The consumer must accept the application as it is without the possibility of making informed decisions. For example, a game application cannot be set to be rejected when the battery falls below 20%. On the other hand, the producer is not able to declare which security actions the application will apply. By signing the
application, the producer declares what they did and has to convince the operators that the application will not do anything harmful.

The work in [12] then proposes the notion of security-by-contract: the digital signature is bound with not only the origin of the code but also a contract. A contract describes relevant features of the application. A relevant feature can be a fine-grained resource control (e.g. automatically send a SMS), memory usage, constraint on access from other applications, user privacy protection, etc. The work also provides a description of the overall life-cycle of application code in the setting of security-by-contract. With each step in the process, the work supplies appropriate implementation mechanisms.

### 2.6 Saint

Like other security models, the Android security model is system-centric. At install time, an application must declare which phone resources it needs and the permissions to access these resources are granted by the system itself or by the user. This feature aims to protect the phone resources. To protect the application, Android provides a static and limited mechanism. Applications statically identify the policy that governs the rights to their data and interfaces at install time. However, the application has limited ability to decide to whom those permissions will be granted and how they are later exercised. Based on this observation, Secure Application INTeraction (Saint) [13] is built as an extension of Android. Saint is able to enforce the following types of application policies:

- Permission assignment policy. This allows application A to define which application B can access A’s interfaces.

- Interface exposure policy. This allows application A to control how other applications use A’s interfaces.

- Interface use policy. This allows application A at run time to choose which application interfaces to use.

Saint infrastructure addresses two types of policy enforcement:

- Install-time policy enforcement. Applications provide installation-time policies that regulate the assignment of permissions to protect their interfaces. An application permission defines the conditions under which it is granted for other applications. The conditions are a set of properties that the requesting application must have (e.g. the requesting application must hold permission to access location information).
• Run-time policy enforcement. Access to or communication between applications is regulated by security policies of both the caller and being called applications. Run-time rules are checked against current state of the phone (e.g. location, time, network configuration, etc) and both applications’ status (e.g. version, accessing to the internet, etc).

2.7 Summary

While it seems that solutions for security enforcement at install time are efficient and effective [3, 9, 10], less conclusive results have been obtained for enforcing security at run time. To improve security enforcement at run time, several approaches have been proposed. Previous sections have discussed some significant solutions, including Dynamic Mandatory Access Control for Multiple Stakeholders, Security-by-Contract and Saint. However, none of them is able to specify context-related policies which are proposed by this work. As for Saint [13], it proposes run-time rules which are checked against current context of the phone described by time, location, network configuration, etc. However, Saint is an application-centric system, i.e. each application has its own security policy. The purpose of our work, on the other hand, is system-centric: all security policies are applied for the whole system. In addition, although Saint supports context-based policy, it does not allow virtual-contexts which can be activated or deactivated manually. Also, Saint allows only the user to change application policies while our work supports trusted parties to define their own policies.
Chapter 3

Android Overview

Android is developed by Google and later by Open Handset Alliance (OHA) [15]. Android is built to be open: developers can take full advantage of all facilities a handset offers and extend Android to incorporate new and emerging technologies. Moreover, there is no difference between the phone’s core applications and third-party applications, and the user is able to customize the phone’s applications as he wishes.

Android is a software stack for mobile devices. The Linux kernel is the lowest level providing device drivers, memory management, etc. The next level is a set of native libraries for graphics, database and so on. Also, Android runtime includes core Java libraries and a virtual machine for running customized bytecode. Above this level is the application framework. Finally, the top layer contains some key applications.

From the security perspective, Android takes advantage of available mechanisms. For example, it uses Linux mechanisms: each application is associated with a different ID. This prevents one application from disturbing another. In addition, Android enhances security with its own mechanisms tuned into mobile devices. Android provides many built-in permission labels for phone resources such as taking pictures, using the Internet, etc. These built-in permissions controls applications accessing the phone system. These security controls are enforced at system level. Meanwhile, Android uses the permission label policy model to regulate communication between applications. An application might define its own permission label to control who can access its components. These controls are regulated by the Inter-Component Communication (ICC) mechanism.
3.1 Android Architecture

Android implements a complete software stack for running applications on mobile devices. An overview of the Android architecture is shown in Figure 3.1 [16]. Android includes four layers: Linux Kernel, Libraries and Android runtime, Application Framework and Applications.

![Android Architecture Diagram](image)

**Figure 3.1:** Overview of the Android Stack.

### 3.1.1 Linux Kernel Layer

Android is based on Linux version 2.6 for basic services such as memory management, process management and security. In addition, the kernel plays the role of communication with the hardware via drivers.

### 3.1.2 Libraries and Android Runtime

All libraries in Android are written in C/C++ but they are called through a Java interface. These includes System C library, Media Libraries, Surface Manager, LibWebCore, SGL, 3D libraries, FreeType and SQLite [16].

Android runtime environment includes two main components [17]:

---

[16]: Placeholder for reference

[17]: Placeholder for reference
• Core libraries. Android runtime environment provides base libraries with main features supporting Java applications. There are two kinds of Java packages used to create applications. First, Android provides a substantial subset of the Java Standard Edition 5 specification, including Collections, I/O, etc. Second, Google Android specific packages were created by Google, ranging from user interface construction to hardware specific functions. Currently, these packages are closed source but Google has committed to make them open source.

• Dalvik virtual machine. This component plays the role of the translator between the application and the operating system. Each Android application is executed in its own process and in a separate Dalvik virtual machine. The Dalvik virtual machine (VM) is not a Java virtual machine. It uses its own bytecode (DEX) instead of Java bytecode. Dalvik VM has been built with significant advantages over J2ME Java Virtual Machine [17]: low electric power consumption, rich libraries and optimizations for minimum memory.

3.1.3 Application Framework

This layer provides application developers access to what Android refers as "service” processes [17]. These services are invisible to the handset user. Application developers communicate with these services via a message bus. These services are built to help application developers. For example, Content Providers allow applications to access data from other applications or to share their own data and Notification Manager helps applications to display custom alerts in the status bar.

However, the Android application framework forces a structure on developers: an Android application must be constructed by means of multiple components [18]. Android defines four types of components with different purposes:

• Activity. This component presents an application’s user interface. Activity components interact with the user via the touchscreen and the keypad. An application can include several Activity components, one for each possible "display screen”. However, only one Activity at a time obtains the command from the keypad and has the focus, while the other Activities are suspended.

• Service. Services are background processes. When an Activity needs to perform some tasks that must continue after the user interface disappears such as playing music, it commonly uses the Services design for those tasks.

• Broadcast Receiver. Android uses the broadcast mechanism for the interaction between application components. Broadcast Receiver acts as a mailbox to receive and react to broadcast messages.
• Content Provider. This component is used to store and share application’s data. The data can be stored in the file system, in an SQLite database or any other manner.

An application can contain any number of combinations from these four components. To communicate with each other, application components send so called intents. An intent is a message object containing a destination component address and data. An intent can specify the component to call either by name (e.g. ”com.android.example.START”) or more abstractly by action (e.g. ”START”).

Figure 3.2 shows typical Inter-Component Communication (ICC) between application components. Android ICC is similar to the Inter-Process Communication (IPC) in Unix-based systems. However, ICC happens identically no matter if the target component belongs to the same application or a different one. Android provides APIs that accept intents and uses that information to start Activities (startActivity(Intent)), start Services (startService(Intent)) and broadcast messages (sendBroadcast(Intent)).

![Diagram of Inter-Component Communication](image)

**Figure 3.2:** Typical Inter-Component Communication. Inter-Component Communication (ICC) between Android application’s components: Activities, Services, Content Providers, and Broadcast Receivers [18].

### 3.1.4 Applications

The Android platform is equipped with a set of basic applications such as browser, email client, SMS program, calendar, etc. All applications are written in Java and can be run simultaneously.
3.2 Android Security

Android security focuses on putting the user in control of the device. Android combined some available security mechanisms (e.g. Linux security mechanisms) and its own security mechanisms tuned for a mobile environment.

3.2.1 Security Mechanisms

Android security mechanisms spread into three categories: Linux mechanisms, environmental features and Android-specific mechanisms [19]. First, from the Linux security’s prospective, Android is a multi-process system in which each application runs in its own process with unique ID. By default, no application has permission to perform any operation that impact other applications, the operating system or the user, for example, reading/writing user’s private data, reading/writing another applications’ files, performing network access, etc. Each application process is a secure sandbox and cannot interrupt other applications unless it explicitly declares the permissions needed by the target sandbox.

Second, Android takes advantage from the surrounding technological environment (e.g. the hardware and mobile carrier’s infrastructure) to enhance its security. Like most other operating systems, Android employs a Memory Management Unit (MMU) which is a hardware component that separates processes into different address spaces. Android uses this component to ensure that one process cannot read another’s memory page or corrupt it. In addition, Android utilizes some security functionalities from telephony systems. These include authentication, authorization and accounting.

Finally, Android dedicates some new security mechanisms. Android introduces the conception of permission label. A permission label is simply a unique text string and can be defined by both Android platform and applications. To control accesses to phone resources, Android provides a set of built-in permission labels. For example, label `android.permission.CAMERA` is used for access control to the camera. Any application can define its own permission labels to regulate accesses to its components. For simplicity, we later use permission to indicate the permission label.

Each permission definition specifies a protection level which can be: normal, dangerous, signature, or signature or system.

- A permission with normal level is automatically granted without asking for the user explicit approval.
- A permission request with dangerous level is granted only with the user confirmation.
• A permission request with signature level does not involve the user—it is granted only if the requesting application is signed with the same key as the application that defines the permission.

• A permission with signature or system is granted only to packages that are in the system image or are signed by the key used for the system image.

A basic Android application has no permissions associated with it. That means it cannot access protected features of the device. To make use of protected features, an application must explicitly declare all required permissions in its package manifest definition. Permissions declared in package manifest are granted at the installation time and cannot be modified later. An operation whose permission is not declared at install time will fail at runtime.

3.2.2 Security Enforcement

As mentioned in the previous section, each application must declare all permissions it wants to use and permission granting is performed at install time. A permission can be declared to access a phone resource (built-in permission) or an application component (permission defined by an application).

Android has two levels of security enforcement [18]: at Linux system level and at ICC level. Enforcement at Linux system level is applied to Android built-in permissions labels, i.e. security enforcement regulates the access to phone resources by applications. Each application package, at install time, is assigned a unique Linux user ID which remains constant for the whole duration of its life on that device. Whenever an application accesses any phone resource, its ID is used to check if the permission for accessing the resource is granted at install time.

On the other hand, security enforcement at ICC level controls defined permissions, regulating access to application components. As pointed out in the previous sections, an application component needs to obtain some specific permissions to access other components. To perform permission check, Android provides a finer grained control through Inter-Component Communication (ICC) reference monitor. A reference monitor provides mandatory access control enforcement of how applications access components [18]. In particular, the reference monitor grants the access depending on permission primitives. Permission enforcement happens in different places during the program operation: (i) when starting an Activity; (ii) when starting or binding a Service; (iii) when sending and receiving a message (the security enforcement controls who can send a broadcast to the associated receiver and who can receive the message); and (iv) when accessing and operating on a Content Provider.
Chapter 4

CRePE Design

To the best of our knowledge, CRePE is the first system that allows smartphones to enforce fine-grained policies depending on the phone’s contexts. Examples of such policies are given in Chapter 1. As discussed in Chapter 3, in the Android platform all permissions of an application are set at installation time and cannot change during its execution. On the other hand, context-related policies need to change during runtime. Consequently, the latter type of policies is not supported by Android. The CRePE system is then developed as an extension of Android supporting context-related policies.

The basic idea of CRePE is to place its own checks before Android permission checks. To perform permission checks, CRePE needs to manage the phone’s current contexts and their policies. A data storage is required to keep all contexts and policies. Since CRePE’s purpose is to handle context-related policies, it needs to be equipped with some context detectors. Also, CRePE allows two kinds of entities to define contexts: the user and trusted parties. Thus, a user interface must be provided and some specific ways of interaction must be supported for trusted parties.

4.1 Definitions

Before presenting the design of CRePE, we introduce some definitions.

**Definition 4.1. Context.** A context is a set of information about the circumstances under which the phone should act in some certain ways (defined by the associated policy).

As a simple example, a context might be defined as a geographical region.

**Definition 4.2. Resource.** A resource is an entity that accessing to it needs to be controlled. In CRePE, there are two kinds of resources: phone resources and applications. Phone resources
are common resources which can be accessed by any application (e.g. Camera, Bluetooth, WiFi, etc.).

**Definition 4.3. Rule.** A rule $R$ describes the control over a resource. In CRePE, the control is a permission to access or not.

For example, a rule may allow access to the camera.

In addition, CRePE supports two special rules: (i) the `AirplaneModeOn` rule forces the phone to turn to the flight mode, i.e., all wireless connections are disabled; (ii) the `SoundOn` rule controls the phone’s sound to be on or off. These rules are designed specially for trusted parties. Also, `AirplaneModeOne` is a special rule: it does not belong to any policy. It is like an operation more than a rule.

**Definition 4.4. Policy.** A policy $P$ defines how the phone acts in a certain context. One policy is defined for one context and one context is associated with one policy. In CRePE, a policy is represented by a set of rules: $P = (R_1, R_2, R_3, ...)$. 

**Definition 4.5. Active Context and Active Policy.** A context, which currently appears to the phone, is called an active context; otherwise it is inactive context. The policy associated to an active context is called an active policy. At a certain time, there may be more than one active context.

Hence, the context by a geographical region is active when the phone is within that geographical region.

**Definition 4.6. Conflict.** A conflict happens when two rules from different active policies are applied to the same resource; but they have different controls over the resource.

For example, one allows accessing the resource while the other denies.

**Definition 4.7. Set of Active Rules.** The set of active rules $\mathcal{R}$ is defined based on all active policies. It consists of all rules, each of which belongs to at least one active policy and there is no conflict in the set, i.e. there is no pair of rules defined by different permissions on a single resource. Basically, it is the union set of the rules of active policies after removing conflicts.

**Definition 4.8. Trusted Party.** A trusted party is a third party, not Android and not the user—whose identity is trusted by Android and by the user.

An example of a trusted party is the company who gives the phone to their employees. In this case, the employee is also the user.

In our implementation, we consider the government to be a trusted party with the highest priority.
4.2 CRePE Overview

Understanding that Android does not support dynamic policies, we aim to build CRePE as an extension of Android to address the problem we raised in Chapter 1. Basically, CRePE controls two kinds of resources: phone resources and applications. As mentioned in Section 3.2.2, Android performs security enforcement to regulate access to these two types of resources. Our solution is to place CRePE checks before Android checks. When the access to a resource is requested, a CRePE check is performed to decide to allow or deny the access based upon all the active policies. In case the request is permitted, the next Android check is then performed. Otherwise—if the CRePE check does not succeed—a denial message is sent to the caller.

In the current implementation, we identified two physical attributes used to switch a context’s state between active and inactive: location and time. Hence, CRePE requires some context detectors; for example, Global Positioning System (GPS) for location detection and the system timer for time schedule. Furthermore, CRePE also provides special ways to change a context’s state. A context might become active or inactive by an explicit notification. A notification can be an interaction from the user or a message from a trusted party. For example, the user can define the friend-using context to indicate the situation that a friend is using the phone. The aim of this context is to restrict the access to some phone resources or applications. Before lending the phone, the user can activate this context in order to protect his phone (restrict the usage or data access).

In CRePE, each context has its own policy which consists of a set of rules. CRePE has a storage to keep all defined contexts and policies. When a context appears or disappears to the phone, its policy becomes active or inactive.

Meanwhile, CRePE provides tools to edit contexts and policies. In CRePE, a context may come from two kinds of entities: the user and trusted parties. CRePE therefore provides a user interface (UI) to help the user operate on his contexts and policies. Also, some interaction mechanisms should be supported so that trusted parties can manipulate their own contexts and policies.

4.3 CRePE Architecture

The CRePE architecture is shown in Figure 4.1. Basically, CRePE includes seven components:

- Policy Provider. It is an archive storing all defined contexts and their corresponding policies, each of which consists of a set of rules.
• Policy Manager. This is the main component of CRePE. It interacts with all other components. Policy Manager manages all active contexts and the set of active rules $\mathcal{R}$ which is formed based on active policies. Depending on requests from other components (e.g. Context Detector, User Interactor and Trusted Party Interactor), Policy Manager might update $\mathcal{R}$ (e.g. activating and deactivating a context requires an update to $\mathcal{R}$). In addition, it is the only component which can access Policy Provider. Receiving requests from other components, it checks for the validity before updating data in Policy Provider.

• CRePE Permission Check. This component is responsible for policy enforcement. CRePE Permission Check catches requests invoking a phone resource or starting an application. Based on the set of active rules $\mathcal{R}$, this component decides to grant permission or not.

• Action Performer. When the set of active rules $\mathcal{R}$ changes, this component is invoked to change the phone’s state according to the new set of active rules. For example, it may stop some tasks which are currently running and not compliant to the new set of active rules.

• Context Detector. This component keeps a list of all contexts needed to listen to and decides when a context is active or inactive. When a context appears or disappears to the phone, it informs the Policy Manager to update the set of active rules $\mathcal{R}$.

• User Interactor. This is an Android application allowing the user to operate on his own contexts and policies. Via this application, the user can add, delete or edit his contexts and policies. Also, the user can interact with this application to activate or deactivate a context.

• Trusted Party Interactor. This component allows trusted parties to manage their own contexts and policies on the phone. Trusted parties may edit contexts and policies, and also activate or deactivate contexts. Only trusted parties’ requests will be exercised and each trusted party can manipulate only on its own contexts and policies. This component therefore must address the problem of authentication and authorization.

Interactions among CRePE components are described by arrows in Figure 4.1. These interactions originate from applications, contexts, the user and trusted parties.

• Interaction between applications and CRePE. When an application sends a request to access a phone resource or to start another application, the request is intercepted by CRePE (arrow A.1). At this point, the request is checked against the set of active rules $\mathcal{R}$ (arrow A.2). If the request is compliant with these rules, the request is passed to the Android permission check (arrow A.3). Android will continue or abort based on the Android decision mechanisms. Otherwise, the request is denied without being passed to Android permission check.
With $A.i$ ($i = 1, 2, 3$) we indicate actions that are consequences of applications interacting with CRePE. Actions $U.i$ ($i = 1, 2, 3$) indicates consequences of the user interaction with CRePE. $C.i$ ($i = 1, 2$) indicates actions happening when contexts appear or disappear to CRePE. $T.i$ ($i = 1, 2$) shows actions happening because of trusted parties communicating with CRePE. Finally, $M.i$ ($i = 1, 2, 3$) indicates actions always happening when the user, trusted parties or contexts interact with CRePE.

- Interaction between contexts and CRePE. When recognizing that a context becomes active or inactive (arrow $C.1$), Context Detector needs to inform Policy Manager to update the set of active rules $\Re$ (arrow $C.2$).

- Interaction between the user and CRePE. Through the application User Interactor, the user can interact with CRePE (arrow $U.1$). User Interactor can request to add, remove or modify the user’s contexts and policies. These requests are sent to Policy Manager (arrow $U.2$). Depending on the context information and the operation on context, Policy Manager may ask Context Detector to add/remove the context to/from the list to be listened to (arrow $M.3$). Also, through this application, the user can activate/deactivate a context manually. In that case, User Interactor then informs Policy Manager (arrow $U.2$) to update...
the set of active rules $\mathcal{R}$. Responses to the user are sent to User Interactor (arrow $U.1$) if necessary.

- Interaction between trusted parties and CRePE. Trusted parties can communicate with CRePE through Trusted Party Interactor. Trusted parties can request to operate on their contexts and policies. Possible operations include adding, removing, and modifying contexts and policies, and activating/deactivating contexts. Similar to the situation of User Interaction, Policy Manager might ask Context Interactor to add/remove the context to/from the list to be listened to (arrow $M.3$) if necessary. Requests to activate/deactivate contexts also necessitate Policy Manager to update $\mathcal{R}$. In case that a response is required, it is sent back to the trusted party (arrow $T.1$).

Besides, when Policy Manager is requested to change context and policy information, it communicates with Policy Provider (arrow $M.1$) to update information. When the set of active rules $\mathcal{R}$ is updated, Policy Manager informs Action Performer to change the phone’s state if necessary (arrows $M.2$).

### 4.3.1 Policy Provider

As mentioned above, all the contexts and policies are stored in Policy Provider. Similar as for the phone’s address book and the calendar provider, Policy Provider is a database embedded inside the Android middleware. We recall from Definition 4.1, that a context is the set of information about the circumstances under which the phone should act in some certain ways (defined by the associated policy). A context is identified by physical properties. Physical properties can be the time, location, noise around the phone, etc. The current implementation of CRePE supports time and location. These physical properties are used to determine when a context is active or inactive (e.g. a context is active only during a period of time). Furthermore, CRePE supports another way to switch context’s state from active to inactive or vice versa. A context can be activated or deactivated by an explicit notification from the user or a trusted party. Besides, a context has other attributes: Enabled, Period, Owner, Activation, Deactivation and Information. The meaning of all the attributes of a policy are defined in Table 4.1.

According to Definition 4.4, a policy is a set of rules, each of which describes the control over a resource. All attributes of a rule are shown in Table 4.2. A resource can be a phone resource or an application. Basically, Android provides each phone resource a permission label. A phone resource is therefore represented by its associated Android permission label.

The rule $\text{SoundOn}$ does not have the attribute $\text{Resource Name}$. Also, $\text{AirplaneModeOn}$ does not have any attribute, it is like an operation on the phone than a rule.
<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>boolean value</td>
<td>This attribute indicates if the context is enabled or not.</td>
</tr>
<tr>
<td>Time</td>
<td>time period value</td>
<td>This attribute shows the period of time in which the context is active or inactive. This can be either a one-time period or repeated period by weekday. Note: this attribute is optional.</td>
</tr>
<tr>
<td>Location</td>
<td>a location value</td>
<td>This attribute indicates the location where the context is active or inactive. Note: this attribute is optional.</td>
</tr>
<tr>
<td>Period</td>
<td>{short, long}</td>
<td>If long, the context information and its policy are kept in Policy Provider after the context becomes inactive. If short, the context information and its policy are removed from Policy Provider after the context becomes inactive.</td>
</tr>
<tr>
<td>Owner</td>
<td>{user, third-party}</td>
<td>If user, context and its policy are defined by the user. If third-party, the context and its policy are defined by a third-party. Policy Provider also stores the identity of the third-party.</td>
</tr>
<tr>
<td>Activation</td>
<td>{notified, auto}</td>
<td>If notified, the context becomes active by a notification from its owner. If auto, the context becomes active automatically, when appropriate. Note: if auto, at least one physical attribute of the context must be defined.</td>
</tr>
<tr>
<td>Deactivation</td>
<td>{notified, auto}</td>
<td>If notified, the context is deactivated by a notification from the owner. If auto, the context is deactivate automatically, when appropriate. Note: if auto at least one physical element of the context must be defined.</td>
</tr>
<tr>
<td>Information</td>
<td>text value</td>
<td>This attribute helps the owner to specify some meanings for the context.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Name</td>
<td>text value</td>
<td>This attribute indicates the resource on which the rule controls. If the resource is an application, it is the name of the application package. If the resource is a phone resource, it is the name of Android permission label required to access that resource.</td>
</tr>
<tr>
<td>Permission</td>
<td>{allowed, denied}</td>
<td>This attribute describes the control over the resource. If allowed, access to the resource is permitted by the rule. If denied, access to the resource is prohibited by the rule.</td>
</tr>
<tr>
<td>Level</td>
<td>{government, trusted-party, user}</td>
<td>This attribute indicates the rule's level.</td>
</tr>
<tr>
<td>Optional</td>
<td>boolean value</td>
<td>If true, the user can choose to applied the rule or not. If false, the rule must be applied.</td>
</tr>
</tbody>
</table>

**TABLE 4.1: Context Attributes.**

**TABLE 4.2: Rule Attributes.**

As defined in the section 4.1, a conflict happens when two active policies define conflicted rules over a resource: one rule with allowed and the other with denied. To avoid conflict, each rule is assigned a priority. In the CRePE system, there are three priority levels: government, trusted-party and user. Level government is the highest and level user is the lowest. Contexts from the user can define only rules with user level. Contexts from the government, however, can define rules at three levels. Contexts from other trusted party can define rules at trusted-party and user levels. Basically, level choice depends on the rule’s importance. For example, the government may decide that a rule is not very important and allows the user’s contexts to overwrite it, the government then defines it as user. However, another rule might be more important and the government gives only other trusted parties the ability to overwrite, the rule then should be defined at the trusted-party level. The conflict resolution is presented in Section 4.3.2.

A rule can be optional, which means the rule is not strict and the user can choose to apply it or not. This is reflected by the Optional attribute. In case that the rule is optional, the user is recommended to follow the rule but can also choose not do it. For example, a meeting room
recommends the phone user to turn off the sound. However if the user is waiting for an important call and does not want to miss it, he can choose not to apply the rule.

4.3.2 Policy Manager

Policy Manager is the only component accessing Policy Provider and it provides means for other CRePE components to edit context and policy information. Hence, Policy Manager should provide all possible operations on the contexts and policies, for example, adding a new context, editing a context’s attributes, adding a new rule to a policy, removing a rule form a policy, etc.

Policy Manager must ensure the validity of the data. For example, it is not allowed to define two different rules on the same resource for one policy. In addition, Policy Manager manages the set of active rules $\mathcal{R}$. To form $\mathcal{R}$, Policy Manager loads rules from all active policies, detects conflicts among them and resolves conflicts, if any. As explained in Definition 4.6, a conflict happens when two rules with different permissions are applied on a single resource. Table 4.3 shows how conflicts are resolved in CRePE. In general, the higher priority wins, i.e. the rule with higher priority is chosen to add to $\mathcal{R}$. In case conflicts happen at one level, the resolution is done based on the restriction: the more restrictive rule wins. i.e. the more restrictive rule is chosen for $\mathcal{R}$. Rules with denied permission are considered more restrictive than rules with allowed permission.

<table>
<thead>
<tr>
<th>User-level</th>
<th>Trusted-party-level</th>
<th>Government-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-level</td>
<td>More restrictive rule wins</td>
<td>Trusted party’s rule wins</td>
</tr>
<tr>
<td>Trusted-party-level</td>
<td>Trusted party’s rule wins</td>
<td>More restrictive rule wins</td>
</tr>
<tr>
<td>Government-level</td>
<td>Government’s rule wins</td>
<td>Government’s rule wins</td>
</tr>
</tbody>
</table>

| TABLE 4.3: Conflict Resolution.

When a context whose Period attribute is short is deactivated, the context and its policy are removed from Policy Provider. Also, when an optional rule becomes active, Policy Manager has to send a notification to the user. The user can then choose to apply the rule or not.

4.3.3 CRePE Permission Check

CRePE system defines a resource as (i) a phone resource or (ii) an application. To govern these two kinds of resources, policy enforcement must be performed while invoking them. Android protects these resources by using permission labels. At install time, the application must declare all permissions associated to the phone resources it wants to use. Permission granting happens at install time. Once installed, the application’s permissions cannot be changed.

As mentioned in Section 3.2.2, Security at Linux system level governs accesses to phone resources. At install time, the application is assigned to a unique ID. When the application accesses to a phone resource at run time, Android permission check is performed based on the
application ID and permissions it declared at install time. To control accessing to phone resources, CRePE Permission Check therefore places one hook before the Android permission check at Linux system level.

We recall that there are four kinds of application components: Activity, Service, Broadcast Receiver and Content Provider. We note that in most cases, an application is started with one of its Activity. Therefore, CRePE enforcement is designed focusing on applications’ Activity components. That means CRePE policies should be enforced when an Activity is requested to start. Android permission check for starting an Activity is enforced at ICC level as discussed in Section 3.2.2. Thus, CRePE Permission Check must place another hook before the Android permission check at ICC level.

At each CRePE hook, the Permission Check examines if the caller is allowed to access the resource based on $\mathcal{R}$ which is provided by the Policy Manager. If there is a rule in $\mathcal{R}$ that controls the requested resource, the access permission is decided by the rule. If there is no rule in $\mathcal{R}$ that gives control over the requested resource, the access is allowed by CRePE. When access is not allowed by CRePE, a denial message is sent to the caller. Otherwise, the access permission is then decided by Android. This ensures that CRePE does not reduce Android’s security.

4.3.4 Action Performer

As mentioned above, this component performs necessary actions within the phone when the set of active rules $\mathcal{R}$ changes. This component relates to rules in $\mathcal{R}$ which deny access to resources. Since permission is checked when any phone resource is invoked, Action Performer does not need to do anything in this case. In contrast, once an application is started, all its functions can be executed without checking the permission again. Action Performer therefore needs to stop running applications which are prohibited by $\mathcal{R}$.

There are some other cases in which Action Performer needs to change the phone’s state. When the rule AirplaneModeOn is active, this component turns the phone to the flight mode and disallows the user to turn it back to the normal mode. If a rule SoundOn with denied permission is added to $\mathcal{R}$, this component turns the phone to vibrate mode. In this case, the user can set the phone to silent mode or vibrate mode but not others. That means the sound must be off.

4.3.5 Context Detector

Context Detector is used to determine when a context becomes active and inactive by physical attributes. A context, by default, is inactive. A context’s state is changed later by either physical attributes or explicit notifications from the context’s owner.
Physical attributes include time and location. A time property defines a time period, including begin time and end time. The time period can be defined as repeated by a set of days in the week. A location attribute is an area characterized with a central point, a radius $r$, and a side value. The central point is defined by the latitude and longitude values. The central point together with the radius form a circle. One possible value of side is in indicating that the location is the geographical area inside the circle. The other possible value is out indicating the geographical location is the area outside the circle. The phone’s position is determined inside or outside a location based on the distance $D$ between the phone’s position to the central point of the location. Table 4.4 explains how to detect if the phone’s position is inside or outside the location.

<table>
<thead>
<tr>
<th>side value</th>
<th>$r - D$</th>
<th>Inside/Outside</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>≥ 0</td>
<td>Inside</td>
</tr>
<tr>
<td>in</td>
<td>&lt; 0</td>
<td>Outside</td>
</tr>
<tr>
<td>out</td>
<td>≥ 0</td>
<td>Outside</td>
</tr>
<tr>
<td>out</td>
<td>&lt; 0</td>
<td>Inside</td>
</tr>
</tbody>
</table>

**Table 4.4: Location Determination.**

(r is the radius, D is the distance between the phone’s position and the central point).

Either a *Time* attribute or a *Location* attribute or a combination of them decides the physical domain of a context. Table 4.5 explains how the physical domain is determined by time and location.

<table>
<thead>
<tr>
<th>Time</th>
<th>Location</th>
<th>Physical Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defined</td>
<td>Not defined</td>
<td>The physical domain is the time period.</td>
</tr>
<tr>
<td>Not defined</td>
<td>Defined</td>
<td>The physical domain is the geographical area.</td>
</tr>
<tr>
<td>Defined</td>
<td>Defined</td>
<td>The physical domain is the geographical area during the time period.</td>
</tr>
<tr>
<td>Not defined</td>
<td>Not defined</td>
<td>The physical domain is empty.</td>
</tr>
</tbody>
</table>

**Table 4.5: Physical Domain.**

According to Table 4.1, *Activation* and *Deactivation* attributes define the way the context becomes active and inactive, respectively. There are two possible values for *Activation* and *Deactivation* attributes: auto and notified. If auto is used, the context’s state changes automatically when the phone gets in or out of its physical domain defined by time and location. In contrast, notified indicates that the context’s state changes by only explicit notifications from the context’s owner. Notifications to activate or deactivate contexts can be the user’s interactions through User Interactor or trusted parties’ messages received by Trusted Party Interactor. Context Detector therefore needs to handle only physical attributes. There are two kinds of notifications: active-notification used to activate a context and inactive-notification used to deactivate a context. Table 4.6 shows how a context’s state is changed according to *Activation* and *Deactivation* attributes.

To detect the time, the system timer can be used. The time values therefore come from the hardware and we assume that those time values are trustable. Global Position System (GPS)
Chapter 4. CRePE Design

### Activation and Deactivation

<table>
<thead>
<tr>
<th>Activation</th>
<th>Deactivation</th>
<th>Explanation for the Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>auto</td>
<td>The context’s state changes to active when the phone gets in its physical domain and changes to inactive when the phone gets out its physical domain.</td>
</tr>
<tr>
<td>auto</td>
<td>notified</td>
<td>The context becomes active when the phone gets in its physical domain and becomes inactive when an inactive-notification is received.</td>
</tr>
<tr>
<td>notified</td>
<td>auto</td>
<td>The context becomes active when an active-notification is received and becomes inactive when the phone gets in the physical domain.</td>
</tr>
<tr>
<td>notified</td>
<td>notified</td>
<td>The context becomes active when an active-notification comes and becomes inactive when an inactive-notification is received.</td>
</tr>
</tbody>
</table>

**TABLE 4.6: State Changes of Contexts.**

[20] or Assisted GPS (A-GPS) [21] can be used for location detection. We also assume that GPS and A-GPS signals are trustable.

### 4.3.6 User Interactor

This is an application that allows the user to operate on his contexts and policies. This application must not allow access to contexts and policies that do not belong to the user. User Interactor should provide all possible operations for the user and should facilitate with friendly user interfaces. For example, a form to edit contexts’ attributes should be included. Also, there should be a form to set rules for a policy. This form must show all available resources on the phone.

Depending on the user’s interactions, this CRePE component sends requests to Policy Manager to get or update context and policy information. In case that the request is to activate a context, or to deactivate a context or to operate on an active policy, Policy Manager has to update the set of active rules $\mathcal{R}$.

### 4.3.7 Trusted Party Interactor

Via some communication mechanisms like SMS, Bluetooth and Near Field Communication (NFC) [22], trusted parties can send requests to Trusted Party Interactor. Requests are represented by a string of text. A standard format is therefore required. This standard format is defined by CRePE and messages, which fail to match this format, will not be processed. In case of failure, Trusted Party Interactor sends a response back to the sender.

Since only requests from trusted parties can be executed and each trusted party can only exercise on its own contexts and policies, authentication and authorization must be considered. A possible solution is to use Public Key Infrastructure (PKI) [23].

After verifying the message, Trusted Party Interactor extracts operations from the message. Possible operations can be to add, remove and modify contexts and policies. Also, operations can be to activate or deactivate a context. These operations are then passed to Policy Manager.
to perform. Policy Manager has to update the set of active rules $\mathcal{R}$ if the operation is to activate or deactivate a context, or if the operation is performed on an active policy.
Chapter 5

CRePE Implementation

Chapter 4 addresses the requirements for the CRePE system as well as presented the design of the system. Basically, CRePE extends the Android platform by adding seven components: Policy Provider, Policy Manager, CRePE Permission Check, Action Performer, Context Detector, User Interactor and Trusted Party Interactor.

Policy Provider plays the role of a database storing all CRePE contexts and policies while Policy Manager governs active contexts and policies. Policy Manager also gets requests from other components to update the information in Policy Provider. As a very important component in the system, CRePE Permission Check catches requests to access resources and decides to grant permission or not based on the set of active rules. Meanwhile, when the set of active rules changes, Action Performer is invoked to execute appropriate tasks. Context Detector is designed to listen to physical attributes of contexts and activate or deactivate them when necessary. Finally, User Interactor and Trusted Party Interactor are two interfaces that allow the user and trusted parties to operate on their own contexts and policies.

5.1 Policy Provider

Policy Provider is a database that keeps the information of contexts and policies. According to definitions in Section 4.1, one context is associated with only one policy which is a set of rules. For simplicity, Policy Provider does not store policies but their rules instead. For each context, Policy Provider stores its information and all rules in its policy. Context and rule attributes are shown in Table 4.1 and 4.2 respectively. The Entity-Relationship of the database is shown in Figure 5.1. A context is associated with multiple rules while a rule belongs to only one context.

In addition, each context is assigned a unique ID. Context IDs are divided into several categories by using prefixes. These prefixes represent context owners. For example, context IDs with prefix
belong to the user. Similarly, each trusted party has its own prefix. These prefixes are also used to ensure each trusted party can operate only on its contexts. Since in one context’s policy there is only one rule for one resource, the rule’s ID is the combination of the context’s ID and the resource name.

5.2 Policy Manager

The design of this component is presented in Section 4.3.2. Policy Manager gives other components the access to Policy Provider. Therefore, all possible operations on contexts and policies are provided. These operations include getting a context’s information, getting all rules of a context, adding a new context, removing a context, setting a rule for a certain policy, etc. One special case is that when a context whose Period attribute is short is deactivated, the context and its policy are removed from Policy Provider.

To support CRePE Permission Check, this component governs all active contexts and the set of active rules $\mathcal{R}$ established by active policies. Algorithm 1 is used to create $\mathcal{R}$. Policy Manager obtains $\mathcal{R}$ by collecting all rules from active policies and removing conflicts. Conflicts are resolved by high-priority-wins and more-restrictive-wins strategies presented in Section 4.3.2.

Likewise, Policy Manager needs to take care of optional rules. While updating the set of active rules $\mathcal{R}$, Policy Manager checks if there is any optional rule. If that is the case, Policy Manager sends a notification to the user and let the user choose to apply or not. Figure 5.2 depicts the notification and the form through which the user can choose to apply an optional rule. After a certain amount of time, if the user does not react, the rule will be applied.

5.3 CRePE Permission Check

As described in Section 4.3.3, the role of CRePE Permission Check is to enforce CRePE policies before the Android permission check is performed. There are two kinds of resources which CRePE needs to handle: phone resources and applications.

Android controls access to phone resources by using permission labels. When a process requests to access a resource, the required permission is checked. In Android, the check is performed by the method checkPermission in the class ActivityManager. To catch the request and
Algorithm 1 Algorithm for creating the set of active rules $\mathcal{R}$

function RuleSet create_set of active rules() {
    $\mathcal{R} \leftarrow \emptyset$
    for each active policy $P_i$ do
        for each rule $R_j$ in $P_i$ do
            add_and_resolve($\mathcal{R}, R_j$)
        end for
    end for
    return $\mathcal{R}$
}

procedure add_and_resolve(RuleSet $\mathcal{R}$, Rule $R$) {
{there is no pair of rules in $\mathcal{R}$ which controls over the same resource}
    for each rule $R_i \in \mathcal{R}$ do
        if $R_i$.ResourceName $\neq$ $R$.ResourceName then
            next {no conflict between $R_i$ and $R$}
        end if
        {the rest of code in for-loop handles the case that $R_i$ and $R$ have controls over the same resource}
        if $R_i$.Priority $> R$.Priority then
            return { $R_i$ wins and $\mathcal{R}$ is not updated}
        end if
        if $R_i$.Priority $< R$.Priority then
            remove($\mathcal{R}, R_i$)
            add($\mathcal{R}, R$)
            return { $R_i$ has more or equal restriction, $\mathcal{R}$ is not updated}
        end if
        {in case that $R_i$ has same priority with $R$, more-restrictive-wins is applied}
        if $R_i$.Permission == denied then
            return { $R_i$ has more or equal restriction, $\mathcal{R}$ is not updated}
        end if
        if $R$.Permission == denied then
            remove($\mathcal{R}, R_i$)
            add($\mathcal{R}, R$)
            return { $R$ and $R_i$ has the same control over the same resource, $\mathcal{R}$ is not updated}
        end if
    end for
}

Chapter 5. CRePE Implementation

enforce CRePE policy before Android policy, a hook is simply placed at the beginning of this method.

We note that Android has two background processes: the phone process under which the telephony code runs, and the system process under which the system code runs. These two processes need to use some phone resources (e.g. WAKE_LOCK, DEVICE_POWER, etc) very often. For some secure reasons, their access phone resources are checked by Android. CRePE does not enforce its own checks to Android background processes. In particular, interfering these processes may crash the phone. Also, if a trusted party or the user by accident defines a rule prohibiting access to a phone resource which is required by one of these processes, that will not affect the phone system.

Two Android’s background processes are assigned their own IDs and they are constant. Before the CRePE hook in checkPermission, the process which requests for accessing the resource is checked if it is one of Android’s background processes based on its own ID. If it is a background process, CRePE permission check is not performed and Android permission is executed as normally. This does not reduce Android security.

To handle applications, Section 4.3.3 shows that request to start Activity components should be governed. To support starting an Activity, the class ActivityManager provides the method startActivity. In this method, a check is performed to ensure that the caller has permission to start the activity. If the condition is not satisfied, the activity does not start and a denial
message is sent back. Similar to the case of phone resources, CRePE Permission Check places another hook before the Android code for permission check.

At each hook, CRePE Permission Check is executed based on the set of active rules $R$ and the requested resource. The check is performed by searching in $R$ to find if any rule defines control over the resource. If no rule is found or one found rule allows the access, `checkPermission` and `startActivity` will continue normally to perform Android permission check. In contrast, if one rule is found and it forbids the access, these methods will return with a negative result. In this case, a denial message is sent to the caller in the same way that Android refuses the access. Figure 5.3 demonstrates how Android denies starting an application and denies access to a phone resource. The messages shown in these figures do not reflect the reason for denying access. However, this is how Android refuses the access and CRePE currently uses this way.

![Figure 5.3: Deny Access.](image)

(a) A deny to use the phone resource READ_SMS  
(b) A deny to start the application Email

5.4 Action Performer

The purpose of Action Performer is to change the phone’s status according to rules in $R$. In detail, there are three cases Action Performer needs to handle. First of all, when a rule that denies access to an application becomes active, Action Performer is requested to stop the application if it is running. The class `ActivityManager` supports the method `restartPackage` which has the system perform a force stop of everything associated with the application package. In particular, all processes that share its ID are killed, all Services which it has running are stopped,
and all its Activities are removed. Figure 5.4 demonstrates how a running application is stopped because it is prohibited by the set of active rules $\mathcal{R}$.

![Figure 5.4: A running application is stopped because of breaking the active rules.](image)

Besides, Action Performer takes care of the situation that the special rule \texttt{AirplaneModeOn} becomes active. Action Performer then switches the phone into the flight mode, i.e. disables all wireless connections. Android provides the static method \texttt{Settings.System.putInt} which writes the phone's settings. To turn the phone to the flight mode, this method is invoked to write setting attribute the \texttt{Settings.System.AIRPLANE_MODE_ON} to 1. Then an intent message with the action \texttt{ACTION_AIRPLANE_MODE_CHANGED} is broadcasted to components that need to execute some appropriate tasks. To prevent the user from turning off the flight mode, we use a new setting attribute \texttt{Settings.System.CREPE_AIRPLANE_MODE_ON}. When this attribute's value is 1, the request to turn off the flight mode is denied.

One problem of the rule \texttt{AirplaneModeOn} is how to switch off the flight mode. Since no wireless connection is enable in the flight mode, no trusted party can reach the phone while the user cannot turn off the flight mode. To deal with this problem, we propose to use an activation code. When the user gets in the airplane, the rule \texttt{AirplaneModeOn} is sent together with an activation code. When the user chooses to turn off this mode, he is asked for the activation code. If the user enters in the correct code, the flight mode is turned off. The attribute \texttt{Settings.System.CREPE_AIRPLANE_MODE_CODE} is also reset so that switching on and off the flight mode can be performed normally after that. The activation code is shown to the user by the airplane company after landing. To implement this solution, Action Performer uses another setting attribute \texttt{Settings.System.CREPE_AIRPLANE_MODE_CODE} which stores the activation code. Figure 5.5 demonstrates how the user switches off the flight mode which was set by CRePE system before.

Finally, the rule \texttt{SoundOn} with denied permission requires turning off the sound. In this case, Action Performer uses the method \texttt{setRingerMode} from \texttt{AudioManager} to set the phone to vibrate mode. To prevent the user from turning on the sound, Action Performer
handles similarly to the case of the rule AirplaneModeOn, using a new setting attribute CREPE_SOUND_OFF. When this attribute is set to 1, only requests to switch to silent mode or vibrate mode are accepted. When the rule becomes inactive, Action Performer returns the phone to the old sound mode (before the rule is in force) and also writes CREPE_SOUND_OFF to 0 so that changing the sound mode can be done normally.

Creating new setting attributes, CRePE ensures that Action Performer is the only component that can write on them. Also, reading Settings.System.CREPE_AIRPLANE_MODE_CODE is always refused. These restrictions guarantee that CRePE airplane mode and CRePE sound mode can be switched on and off only by CRePE.

5.5 Context Detector

Context Detector is designed to observe the environment around the phone and decides when a context becomes active or inactive. When a context’s state changes, Context Detector notifies Policy Manager to update the set of active rules $\mathcal{R}$.

5.5.1 Entrance and Exit Events

As mentioned in Section 4.3.5, this component needs to determine when the phone gets in or out of a context’s physical domain. A context’s physical domain is defined by its Time and Location attributes (see Table 4.5). We define that the entrance event for a context happens when the
phone gets in its physical domain and the exit event happens when the phone gets out of the physical domain. These events are listened to when necessary, depending on the Activation and Deactivation attributes of the context. Table 5.1 explains when entrance and exit events of a context are listened to. Note that in the third case, Activation is notified and Deactivation is auto, the context’s state is set to active by explicit notifications and is set to inactive when the phone enters the context’s physical domain.

<table>
<thead>
<tr>
<th>Activation</th>
<th>Deactivation</th>
<th>Explanation for the Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>auto</td>
<td>When the context is inactive, the entrance event is listened to activate the context. When the context is active, the exit event is listened to deactivate the context.</td>
</tr>
<tr>
<td>auto</td>
<td>notified</td>
<td>When the context is inactive, the entrance event is observed to activate the context.</td>
</tr>
<tr>
<td>notified</td>
<td>auto</td>
<td>When the context is inactive, the entrance event is observed to deactivate the context.</td>
</tr>
<tr>
<td>notified</td>
<td>notified</td>
<td>No event is listened.</td>
</tr>
</tbody>
</table>

**TABLE 5.1: Entrance and exit events.**

When a context’s state is changed either automatically or by a notification, its entrance and exit events are listened to depending on the context’s Activation and Deactivation attributes. To keep track of all events to be listened to, Context Detector uses two lists mListTime and mListLocation. These two lists are updated when an event is starting to be listened to or is triggered.

As mentioned in Section 4.3.5, a time value represents a period of time defined by a begin time and an end time. Context Detector uses a list mListTime to keep all time events needed to listen to. Since the Time attribute of a context might be a repeated period, a time event can be the closest begin time or the closest end time of a context. When a context’s entrance event is examined and its Time attribute is defined, the closest begin time is added to mListTime. Similarly, when a context’s exit event is detected and the Time attribute is defined, its closest end time is added to mListTime. A time event is triggered when it matches with the phone’s time.

Similarly, Context Detector keeps a list of location events mListLocation. Each element in this list is a location value including the longitude and latitude of the center point, the radius and the side value (see Section 4.3.5). The list mListLocation should keep the location information of a context if its entrance or exit event is listened to. In case the entrance event is examined, the context’s location is added to mListLocation. In case the exit event is listened, the context’s location with reversed value of side (in becomes out and out becomes in) is added. A location event is triggered when the phone gets in that location.

When list mListLocation is not empty, i.e. there is no location event being listened to, the phone’s position is not required to determined. Since getting the phone’s position is expensive, a location is inserted to mListLocation only when necessary. In detail, when a context’s entrance or exit event is examined, its location value is added to mListLocation if its Location attribute is defined while its Time attribute value is not. If both the Time and Location
attributes are defined, the location value appears in mListLocation only during the time period (defined by the time value).

A context’s entrance or exit event is triggered if the corresponding time and location events are triggered. In detail, if both Time and Location attributes are defined, the context’s entrance or exit event is triggered only when both of its time and location events are triggered. If only the Time attribute is defined, the context’s entrance or exit event is triggered when its time event is triggered. The similar procedure is applied for the case that only the Location attribute is defined. When an entrance or exit event is triggered, Context Detector informs Policy Manager to activate or deactivate the corresponding context, depending on the event specification (see Table 5.1).

5.5.2 Event Listener

To listen to time events, Context Detector uses Alarm Manager Service, a built-in service in Android. Context Detector registers a time event with this service and is notified when the system time matches with the time event. To avoid overwhelmed requests to Alarm Manager Service, Context Detector registers only the earliest time event in mListTime. Context Detector interacts with Alarm Manager Service through Broadcast Receiver mTimeReceiver which receives intents with the action CREPE_TIME_ALERT_ACTION. When the time event is reached, Alarm Manager Service broadcasts an intent with the action CREPE_TIME_ALERT_ACTION. mTimeReceiver catches the intent and triggers the corresponding time event. To prevent the user from changing the time, the phone is forced to update the time from the network operator by setting the system attribute Settings.System.AUTO_TIME to 1. Android middleware is modified so that only CRePE can write this attribute. The phone’s time is then updated automatically and the user and applications cannot interfere.

Similarly, to trigger location event, Context Detector registers for a location service with Location Manager Service. Location Manager Service sends notifications when possible. Registering for a location service requires two parameters: time $t$ and distance $d$. The first parameter indicates the minimum time interval for notifications while the second parameter is the minimum distance interval for notifications. That means the time period between two notifications must be greater than or equal to $t$; and the distance that the phone moves between two notifications must be greater than or equal to $d$. The time parameter is used as a hint to save the power. The actual interval for updating the phone’s location might be greater or lesser than this value [24].

To avoid overwhelmed requests to Location Manager Service, Context Detector registers to Location Manager Service at most once at a time. Since Context Detector needs only the first notification, it removes the service registration and replaces by a new one if necessary after receiving the first notification. Context Detector sets the time parameter as a constant
CREPE\_LOCATION\_TIME\_INTERVAL. This constant can be changed depending on the requirements. The distance parameter is the shortest distance from the phone’s position to all locations in list mListLocation, i.e. the minimum distance the phone has to move to reach one of locations in mListLocation. The equation for the distance parameter is shown in Equation 5.1.

\[
distance\_parameter = \text{Min}(x) \\
\forall L \in \text{mListLocation}, \\
x = |L.\text{radius} - \text{Distance}(L.\text{center}, \text{Phone.position})|
\]  

(5.1)

Context Detector receives notifications through a Broadcast Receiver mLocationReceiver which picks up intents with action CREPE\_LOCATION\_ALERT\_ACTION. When catching a notification, Context Detector checks if the phone’s position is in any location. In that case, the location event is triggered. Otherwise, it registers for a new location service based on the phone’s new location. By using Equation 5.1 to set the distance interval parameter, Context Detector guarantees that if the phone reaches a location in mListLocation, the corresponding location event will be triggered.

One possible problem is that the user might disable location providers and the phone’s position therefore cannot be retrieved. In that case, Context Detector forces to enable location providers when necessary (i.e. when mListLocation is not empty). This can be done by using the static method Settings.Secure.setLocationProviderEnabled. Since CRePE is implemented as a part of the system process in Android, it has the privilege to get the phone’s position even if there is an active rule prohibiting the Android permission ACCESS\_FINE\_LOCATION or ACCESS\_COARSE\_LOCATION.

### 5.5.3 Google’s Close Source Problem

To retrieve the phone’s location, Android Location Manager Service support A-GPS by using two kinds of providers: GPS provider and Network provider. The GPS provider determines the location using satellites. Depending on conditions, this provider may take a while to return a location fix. Network providers, on the other hand, determine the location based on availability of cell tower and WiFi access points. The GPS provider gives the most accurate values but it takes the most time and energy. In contrast, Network providers take less time and energy. In detail, the information from cell towers brings out the location fix fastest but least accurately. WiFi access points provide quite good accuracies at relevant speed. However, WiFi is not always
available. Consequently, a combination of these three location sources needs to be used to obtain a considerable accuracy at an appropriate cost.

However, using Network providers necessitate some APIs from Google specific packages (Section 3.1.2) which are not open at this moment. However, Google has committed to make it open soon. In the current implementation, CRePE uses only GPS provider to locate the phone.

5.6 User Interactor

User Interactor is an Android application allowing the user to manipulate his own contexts and policies. The first form of the application is CRePE Setting Form which lists all the user’s contexts. From this form (Figure 5.6(a)), the user can enable, disable, add and remove contexts. To edit context information, the user employs the form like Figure 5.6(b). This form allows setting attributes for a context. To facilitate time and location settings, the application supplies two other forms (Figure 5.6(c) and Figure 5.6(d)). The Location Setting Form especially supports getting current position of the phone. This functionality helps to use the current position as a part of the context’s Location attribute.

To manipulate with policies, the user can define rules for resources. One policy is represented by a set of rules. The form in Figure 5.6(e) displays a policy’s rules. From this form, the user can edit these rules’ attributes. Also, to add rules, the user can access the form as in Figure 5.6(f). Thanks to the radio group in this form, the user can choose which kinds of resources he wants to handle. For application resources, the form displays all current applications on the phone. The phone resource form shows some certain phone resources. In general, Android provides more than one hundred permission labels to protect phone resources. However, most of them are at very low level and the user should not touch these phone resources. To avoid misuses from the user, we expose only some necessary phone resources to the user:

- **BLUETOOTH.** The user can control if the Bluetooth service is accessible or not.
- **CHANGE_CONFIGURATION.** The user can choose to allow his phone configuration to be changed or not.
- **INTERNET.** The user can decide to allow or disallow using the Internet.
- **INSTALL_PACKAGES.** This permission decides if applications can be installed to the phone.
- **DELETE_PACKAGES.** The user can decide to allow or disallow deleting application packages.
Chapter 5. CRePE Implementation

(a) CRePE Setting Form  
(b) Context Setting Form  
(c) Time Setting Form  
(d) Location Setting Form  
(e) Rule Setting Form  
(f) Rule Adding Form  
(g) Log In Form  
(h) Password Change Form

**Figure 5.6:** User Interactor Forms.
• **READ_SMS.** Using this permission, the user controls if SMS messages can be read or not.

• **RECEIVE_SMS.** This permission controls if the phone can receive SMS messages.

• **SEND_SMS.** This permission controls if the phone can send SMS messages.

• **READCALENDAR.** This permission controls if reading the user’s calendar data is allowed.

• **WRITECALENDAR.** This permission controls if writing the user’s calendar data is allowed.

• **READOWNER_DATA.** Using this permission, the user decides if his own data can be read or not.

• **WRITEOWNER_DATA.** Using this permission, the user decides if his own data can be written or not.

• **READCONTACT.** This permission is used to allow or disallow reading contacts.

• **WRITECONTACT.** This permission is used to allow or disallow writing contacts.

Moreover, to ensure only the real user can access the application, the user must enter a password to use this application (Figure 5.6(g)). After installation, the application’s password is a default value 0000. The user later can change the password through Password Change Form (Figure 5.6(h)). The CRePE system guarantees that only this CRePE application can operate on CRePE contexts and policies. Also, no rule is able to control access to this application. That means this application can always be started but to actually use it, the user must be authenticated by entering the password.

### 5.7 Trusted Party Interactor

As designed in Section 4.3.7, this CRePE component provides mechanisms to interact with trusted parties. Currently, only the SMS mechanism is implemented. Also, authentication and authorization are not yet implemented. We leave these functionalities for the future works.

To distinguish a normal SMS message from a CRePE SMS message, Trusted Party Interactor uses a certain prefix which is a special string and does not normally appear in a SMS message. When receiving an SMS message, Trusted Party Interactor scans the message. If it does not start with the correct prefix, i.e. it is not a CRePE message, the message is shown to the user. Otherwise, the message is not displayed.
CRePE messages must be in the correct format. The syntax for CRePE messages is shown in Table 5.2. Basically, a correct CRePE message must start with a prefix. This prefix is reflected in the field <Crepe_Header>. The next field, <Sender>, provides a trusted party’s identifier. This information is used later for authentication and authorization. The field <Reply> provides the reply number which Trusted Party Interactor uses to send responses if necessary. Each <Statement> is a request which includes an operation represented by <Operation> and its parameters provided by <Parameters>.

```
<CrepeMessage> ::= <Header> <Statements>

<Header> ::= <Crepe_Header> <Sender> <Reply>
<Crepe_Header> ::= !#%%!Crepe Command By SMS
<Sender> ::= Sender = <Sender_Name> ;
<Reply> ::= Reply = <Reply_Number> ;
(Sender_Name) ::= ['A'..'Z' 'a'..'z' '0'..'9']*
<Reply_Number> ::= ['0'..'9']*

<Statements> ::= <Statement> <Statements> | <Statement>
<Statement> ::= <Operation> <Parameters>

<Operation> ::= Operation = <Operation_Name> ;
<Operation_Name> ::= ModifyContext | AddContext
keiten RemoveContext | ActivateContext | DeactivateContext
 ::= EnableContext | DisableContext
 ::= SetRule
 ::= SetAirplaneMode

<Parameters> ::= <Parameter> <Parameters> | <Parameter>
<Parameter> ::= <Parameter_Name> = <Parameter_Value> ;

<Parameter_Value> ::= ['A'..'Z' 'a'..'z' '0'..'9' '']*
<Parameter_Name> ::= ContextName | ContextOwner | Activation | Deactivation
 ::= BeginHour | BeginMinute
 ::= EndHour | EndMinute | Repeat
 ::= Latitude | Longitude | Radius | InOut
 ::= Period | Enabled | ContextInfo
 ::= ResourceName | Permission | Level | Optional
 ::= AirplaneModeCode
```

Table 5.2: CRePE Message Syntax.

Each operation has different parameters. Operations on contexts includes ModifyContext, AddContext, RemoveContext, ActivateContext, DeactivateContext, EnableContext and DisableContext. All of them require ContextId parameter to specify which context to be operated on. Only ModifyContext and AddContext operations need a subset of the following parameters: ContextName, ContextOwner, Activation, Deactivation, BeginHour, BeginMinute, EndMinute, Repeat, Latitude, Longitude, Radius, InOut, Period, Enabled, ContextInfo. These parameters represent the corresponding attributes of the context, e.g. ContextName indicates the...
name of the context. If some of these parameters are not specified, corresponding attributes are unchanged in case of \textit{ModifyContext} or are set by default values in case of \textit{AddContext}.

There is only one operation on rules, \textit{SetRule}, which is used to define a control over a resource. This operation requires the \textit{ContextId} parameter indicating the context’s policy which the rule belongs to. Also, this operation necessitates \textit{ResourceName}, \textit{Permission}, \textit{Level}, \textit{Compromise} corresponding to a rule’s attributes. Parameter \textit{Level} and \textit{Compromise} are optional. Similarly, the corresponding attribute is set with a default value if a parameter is not defined. To unset a rule, this operation is used and specifies the \textit{Permission} parameter as \textit{not defined}.

Additionally, the special operation \textit{SetAirplaneMode} is used to turn the phone to the flight mode. Parameter \textit{AirplaneModeCode} provides the activation code is required for this operation. The activation code is used to turn off the airplane mode as presented in Section 4.3.7.

All extracted operations are performed in the order that they appear in the message. Two examples of CRePE messages are shown in Figure 5.7.

(a) The CRePE message sent when the user gets in a museum

(b) The CRePE message sent when the user gets out a museum

These messages are sent by a museum which is assumed to be a trusted party. There are three requested operations in the first message (Figure 5.7(a)). The first operation is to add a new
context specifying that the phone is inside the museum. The Activation attribute is 1, i.e. the context is activated by a notification. The Period attribute is 1 which represents short, i.e. CRePE system does not keep the context and its policy after it becomes inactive. Next is the operation to set a rule for the context’s policy. This rule forbids Camera usage in the context. To activate the context is the last operation. This message is sent when the user gets in the museum. While he still stays inside the museum, the Camera cannot be used. When he gets out of the museum, the second message (Figure 5.7(b)) is sent to deactivate the context.
Chapter 6

System Evaluation

The CRePE system is an extension of Android, supporting context-related policies. By placing hooks before some Android permission checks, CRePE does not reduce Android security. Moreover, the current delegation mechanism of Android has a flaw from which the phone security can be compromised and CRePE lessens it.

On the other hand, CRePE extends Android by adding some components which cause overhead. There are three points in CRePE which cause additional tasks for Android without the user’s interception: (i) CRePE permission check, (ii) Context Detector, and (iii) Trusted Party Interactor. Considering overhead from the perspective of energy consumption and execution time, we do some experiments to evaluate the performance of CRePE.

Besides, CRePE includes one application which allows the user to operate on his contexts and policies. An experiment is therefore designed to obtain response time to the user’s interactions.

6.1 Security Discussion

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

Furthermore, we observe that the current delegation mechanism of Android has a weakness that CRePE fixes to some extent. In particular we consider a weakness as following. An application App is allowed to access a resource Res (e.g. to use the Bluetooth service). App has component Comp that actually uses Res. This component is public and therefore can be used by other
applications. To control access to $Comp$, $App$ defines permission $Perm$ at normal level which is automatically granted without asking for an explicit approval from the user. This would imply that any other application can use $Res$, while the user is not actually aware of this. To some extend, CRePE helps to prevent this kind of compromise. A CRePE rule could be defined by the user to limit the access to resources in some situations. For instance, the user can define a policy allowing the usage of Bluetooth only at home or office which are trusted environments. In this case, this kind of policy compromise to access Bluetooth can still happen but only in trusted environments (home and office).

Finally, we underline that an adversary (either the user or an application) cannot skip the CRePE enforcement. We remind that CRePE is designed as an extension of Android, and it will run with the privileges of the Android Middleware. The only part of CRePE outside the middleware is User Interactor. User Interactor is the only application CRePE trusts and the user must be authenticated to use it (by entering password to log in). The adversary cannot influence context activation or deactivation since the values considered for these operations (e.g. the 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) [25] can be used. However, the discussion of these mechanisms is outside the scope of this thesis.

### 6.2 Overhead Discussion

In the section, we report on experimental results we conduct in order to evaluate the performance of CRePE. Since execution time and energy consumption are two main issues of smartphones, we evaluate time and energy overhead for the different features of CRePE. The experiments were conducted using a HTC Magic smartphone [26]. In particular, we use the developer version (Android Dev Phone 2 [27]) of the HTC Magic phone, featuring it an unlocked bootloader that we need to install our custom system images of Android, which includes CRePE. We identify three main characteristics of CRePE that carry out additional tasks for Android without the user’s interception:

- CRePE Permission Check is called before the Android Permission Check.
- Context Detector may continuously listen to contexts’ entrance and exit events.
- Trusted Party Interactor performs some tasks when receiving requests from trusted parties.
6.2.1 CRePE Permission Check Overhead

As mentioned in Section 4.3.3, CRePE Permission Check is called when access to resources are requested. CRePE aims to protect two kinds of resources: phone resources and applications. CRePE Permission Check is executed only if there is a request to start an application or if an application requests to access a phone resource. These checks are not performed frequently but on demand. Energy consumption is therefore insignificant and only execution time is considered.

Since CRePE Permission Check is to look up if any rule in the set of active rules $\mathcal{R}$ has control over the resource, the overhead of CRePE Permission Check depends on the size of $\mathcal{R}$. We do some experiments to measure execution time of CRePE Permission Check. Each rule represents one resource and we argue that in most cases the maximum size of $\mathcal{R}$ can be 50 rules. We therefore propose experiments for different sizes of $\mathcal{R}$: 0 rule, 1 rule, 5 rules, 10 rules, 25 rules and 50 rules.

Figure 6.1 shows the time overhead for CRePE Permission Check according to the size of the set of active rules $\mathcal{R}$. For each value of the size of $\mathcal{R}$, the time to execute CRePE Permission Check is measured 100 times. As shown in Figure 6.1, the time overhead (measured in nanoseconds) of CRePE Permission Check is very small.

![Figure 6.1: Time overhead for CRePE Permission Check.](image)
The red point is the average value and the green vertical bar shows the standard deviation.
6.2.2 Context Detector Overhead

From the prospective of Context Detector, 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 checking frequency is—hence, the higher overhead. This is true for contexts which are required to be activated or deactivated automatically. As discussed in Chapter 4, there are two context attributes that CRePE needs to detect: *Time* and *Location*. For detecting time period, we simply use the system timer which informs CRePE when the period is reached. It is more complicated for detecting location. The simplest approach is to consider a continuous polling to get the phone’s position. This approach is improved by combining with the time period when possible (discussed in Section 5.5). Because of the close source problem, the current implementation of CRePE is not able to utilize Network providers to determine the phone’s position. CRePE therefore can only use the GPS provider which consumes a considerable amount of energy.

Since the overhead for time detection is small compared to locating the phone, we do experiments for location detection only. We run experiments to evaluate our current implementation of CRePE—that only leverages the GPS provider and the polling mechanism. We consider 3 contexts which define the *Location* attributes but not the *Time* attributes. The experiment is done with two different values of polling interval (CRePE\_LOCATION\_TIME\_INTERVAL): 5 minutes and 15 minutes. These three contexts are enabled and they require obtaining the phone’s position after a period of CRePE\_LOCATION\_TIME\_INTERVAL. We run the experiment for 5 consecutive hours, starting with a fully charged battery. The result is shown in Figure 6.2.

![Figure 6.2: Energy overhead for GPS locating.](image)

While these results are not trascurable, the energy consumption for checking every 15 minutes is quite promising. In fact, in the problem addressed by CRePE, some optimizations are possible.
from this point of view. As we mention above, the optimization that combines location with the time period has been done by CRePE. Another promising optimization is to utilize Network providers. In addition, we also propose one optimization as the future work, that is to vary the polling interval according to the phone’s position and accelerometer (see Chapter 7).

Also, Context Detector may cause some time overhead without the user’s interception. That is the case when a context is detected to be active or inactive. Policy Manager then has to activate or deactivate the context and update the set of active rules $\mathcal{R}$. When a context becomes active or inactive, the workload depends on the number of rules in the associated policy, if there is any conflict between the new active policy and current active policies, and how many contexts are currently active. We do the experiment with a quite complex situation as follows. The current system has two active contexts, each of which has 10 rules. There is no conflict among these rules and that means the size of $\mathcal{R}$ is 20 rules. A new context, which includes 10 rules, becomes active. Among these new rules there are 5 rules in conflict with 5 rules in the current $\mathcal{R}$. We measure the time to activate this context. The new active context is then deactivated. Time for deactivating this context is also measured. Each value is measured for 20 times to obtain the average. The results including average and standard deviation values are shown in Table 6.1.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Execution Time (in nanosecond)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context activation</td>
<td>$72.4868 \times 10^6 \pm 19.7426 \times 10^6$</td>
</tr>
<tr>
<td>Context Deactivation</td>
<td>$42.9420 \times 10^6 \pm 4.5362 \times 10^6$</td>
</tr>
</tbody>
</table>

TABLE 6.1: Average time for context activation and deactivation.

The execution time is measured in nanosecond and is very small. We therefore conclude that time overhead for context activation and deactivation is trascurable.

6.2.3 Trusted Party Interactor Overhead

Trusted Party Interactor performs tasks only when receiving a message from a trusted party. These tasks include parsing the message to extract operations and performing these operations through Policy Manager. Possible operations are to add/delete/modify a context, to set/unset a rule, or to activate/deactivate a context. These operations are not done frequently but on demand. We therefore examine only the time overhead for Trusted Party Interactor.

There are several possible operations and each operation has different workloads depending on the current status of the CRePE system. For example, enabling a context may require setting up event listeners depending on its $Activation$ and $Deactivation$ attributes. We assume that there is not much difference among operations and configurations and we do experiments in some situations in order to provide an overview of interaction overhead from trusted parties.

We consider five following situations:
Situation 1. This situation is similar to the one used in Section 6.2.2. The current system has two active contexts, each of which has 10 rules. There is no conflict among these rules, making 20 rules in $\mathcal{R}$. A new context, which includes 10 rules, is activated by an SMS message from the context’s owner, also a trusted party. Among these new rules, there are 5 rules conflicted with 5 rules in the current $\mathcal{R}$. The time for activating this new context is measured.

Situation 2. The current configuration is the system after the new context is activated in Situation 1. Then, this context is deactivated by an SMS message. The time to deactivate the context is measured in this situation.

Situation 3. A new context is added by an SMS message. This context is not enabled, i.e. there is no need to set up event listeners for the context. The time to add a new context is measured.

Situation 4. A rule is added to a policy by an SMS message. This policy is not active, i.e. the set of active rules $\mathcal{R}$ is not updated. The situation aims to obtain the time for adding a new rule.

Situation 5. A context is enabled by an SMS message. The Activation and Deactivation attributes of the context are auto. Both of the Time and Location attributes of this context are defined. That means event listeners must be set up to detect the context. The time to enable the context is measured in this situation.

Each measurement is done 10 times. The results of all measurements are shown in Table 6.2 including average and standard deviation values. Each measurement is separated into two parts: message processing including the time to read, parse the message and extract the operations; and action performing, which is the execution time for Policy Manager to operate on contexts and policies. The results are measured in nanoseconds and are very small. Until now, the time overhead for trusted party interaction seems small. We note that the current implementation of CRePE does not include authentication and authorization. After these features are added, the part for message processing will be higher. In addition, multiple operations can be combined in one message to save time for message processing.

<table>
<thead>
<tr>
<th>Situation</th>
<th>SMS Processing (in nanosecond)</th>
<th>Action Performing (in nanosecond)</th>
<th>Total Time(in nanosecond)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$20.6104 \times 10^6 \pm 4.8953 \times 10^5$</td>
<td>$55.2795 \times 10^6 \pm 4.8750 \times 10^5$</td>
<td>$75.8900 \times 10^6 \pm 5.2946 \times 10^5$</td>
</tr>
<tr>
<td>2</td>
<td>$21.3287 \times 10^6 \pm 4.6299 \times 10^5$</td>
<td>$42.9420 \times 10^6 \pm 4.5962 \times 10^5$</td>
<td>$64.2708 \times 10^6 \pm 4.9714 \times 10^5$</td>
</tr>
<tr>
<td>3</td>
<td>$23.4882 \times 10^6 \pm 4.0373 \times 10^5$</td>
<td>$44.8353 \times 10^6 \pm 10.4409 \times 10^5$</td>
<td>$68.3235 \times 10^6 \pm 10.8266 \times 10^5$</td>
</tr>
<tr>
<td>4</td>
<td>$26.6480 \times 10^6 \pm 6.7770 \times 10^5$</td>
<td>$49.5331 \times 10^6 \pm 8.6273 \times 10^5$</td>
<td>$76.1810 \times 10^6 \pm 12.1072 \times 10^5$</td>
</tr>
<tr>
<td>5</td>
<td>$28.9000 \times 10^6 \pm 9.1350 \times 10^5$</td>
<td>$37.2731 \times 10^6 \pm 6.6094 \times 10^5$</td>
<td>$66.1731 \times 10^6 \pm 10.7812 \times 10^5$</td>
</tr>
</tbody>
</table>

Table 6.2: Time overhead for trusted party interactions.
6.3 Response Time

User Interactor is the only CRePE component at the application level. In fact, it is an application that allows the user to operate on his contexts and policies. Basically, the user interacts with this application through the touch screen and the keyboard. The user may interact to request some operations on the CRePE system. For example, the user can add a new context, remove a context or activate a context. As discussed in Section 6.2.3, there are many possible operations and workloads for each operation changes according to the system’s configuration. We therefore choose some situations and do the experiment on them. We use the same situations as in Section 6.2.3 but all requests do not come from SMS messages but from the user’s interactions. Each value is measured 10 times to obtain the average. The results are shown in Table 6.3 including average and standard deviation values.

<table>
<thead>
<tr>
<th>Situation</th>
<th>Response Time (in nanosecond)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$116.1620 \times 10^6 \pm 13.222 \times 10^6$</td>
</tr>
<tr>
<td>2</td>
<td>$92.2037 \times 10^6 \pm 13.4115 \times 10^6$</td>
</tr>
<tr>
<td>3</td>
<td>$67.8803 \times 10^6 \pm 8.7889 \times 10^6$</td>
</tr>
<tr>
<td>4</td>
<td>$45.1511 \times 10^6 \pm 4.2759 \times 10^6$</td>
</tr>
<tr>
<td>5</td>
<td>$38.6688 \times 10^6 \pm 2.0960 \times 10^6$</td>
</tr>
</tbody>
</table>

**Table 6.3:** Response time for user interactions.

The results are in nanosecond and small. We assume that the workloads for other operations are not very different. We then conclude that the response time for user interactions are very small and acceptable.
Chapter 7

Conclusion and Future Work

This thesis proposed a solution to enforce context-related policies on smartphones. The solution is reflected by the CRePE system, an extension of Android. In this thesis, we designed and implemented CRePE. Furthermore, we performed a set of experiments to evaluate the performance of CRePE. From the result of these experiments, we concluded that the solution is feasible. Although CRePE’s design is implemented, there are still some points that can be improved. Friendly announcements can be used to inform the user when access is denied or an application is stopped by CRePE. The energy overhead of context detection can be improved by utilizing the phone’s accelerometer. The authentication and authorization for interactions with trusted parties must be addressed. Finally, some additional mechanisms for trusted party interaction as well as physical attributes of contexts should be supported.

7.1 Conclusion

Smartphones have become widely used. However, the lack of the user’s possibility to specify fine-grained security policies makes it more difficult to adopt this technology in its full potential. As an example, the user might avoid using an application if he does not really trust the application for not sending out his private information. In this thesis, 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.

This thesis provided a complete design for CRePE as an extension of the Android platform. CRePE can manage contexts and policies defined by the user and trusted parties. CRePE is able to detect when a context becomes active or inactive. Also, CRePE allows contexts to be activated or deactivated by explicit notifications from the user or trusted parties. CRePE governs
active contexts and policies and changes the phone’s security configuration depending on active contexts. CRePE enforces its own security policies before Android’s permission enforcement and that does not reduce Android’s security. CRePE also supplies a friendly interface for the user to interact with CRePE. Finally, CRePE introduces a mechanism supporting trusted parties to have some control over the phone.

CRePE is implemented on Android 1.6 and is tested to ensure that the main requirements in the design are satisfied. We also performed some experiments to examine the overhead of CRePE compared to the original Android. The execution time is very small, not even half of a second for one of the most complicated operations. The energy overhead is quite high for the context detection which is phone location in this case. However, this problem may be due to inability to use Network providers discussed in Section 5.5.3.

In conclusion, using context-related policies is a good approach for security enforcement at run time and CRePE is a feasible solution for this approach in smartphones. From the user’s point of view, CRePE is a promising system supporting fine-grained and flexible security policies. For trusted parties, this is a relevant solution to have some control over the smartphones.

7.2 Future Work

In this section, we identify some points in the current implementation which can be improved. As seen in Figure 5.3 and Figure 5.4, CRePE currently uses the way in which Android denies access to resources and stops applications. These ways do not supply any reason to the user and are indeed unfriendly. One future work is therefore proposed to provide friendly announcements to the user. The form shown to the user should provide the reason to deny access or to stop the application. For example, the context information can be included into the form.

We discuss in Section 6.2.2 that power consumption for locating the phone is quite high. This may be due to the problem of Google close source for Network providers. However, Google has committed to make it open. It is very likely that the energy overhead will be decreased when network providers can be used for locating the phone. In addition, we offer an optimization that is to use the phone’s accelerometer. As mentioned in Section 5.5.2, a location event is triggered by a locating service from Location Manager Service. The current implementation of CRePE uses the constant CREPE\_LOCATION\_TIME\_INTERVAL as the time interval parameter for the locating service. The locating service therefore retrieves the phone’s location after each time interval. If the time interval is too small, the phone’s position is retrieved very often and that takes much energy. If the time interval is too large, the phone’s position is retrieved not often and that causes delays for location events. One approach is to vary the time interval parameter depending on the distance from the phone’s position to the location being listened to and the
phone’s accelerometer. For example, when the phone’s position is quite far from the location being listened to and it is moved slowly, the time interval can be high.

Besides, there is another problem that CRePE does not address yet. That is the problem of authentication and authorization for interactions with trusted parties. One possible approach is to use PKI. The authentication mechanism is used to confirm the identity of the trusted party. The authorization mechanism ensures that a trusted party can operate only on its own contexts and policy. Also, authorization can be used to limit some resources a trusted party can operate on. For example, the rule \texttt{AirplaneModeOn} can only be used by airplane companies.

Some extra mechanisms should be supported for the interactions from trusted parties. The current implementation supplies SMS as the only means for trusted parties to send/receive messages to/from CRePE. Some other interaction mechanisms can be considered such as Bluetooth and NFC.

Finally, CRePE can be enhanced with additional physical attributes of contexts. One example of these attributes is the user’s presence. This attribute can be used to protect the user’s privacy. In this case, the context \texttt{friend-using} mentioned in Chapter 1 can be activated or deactivated automatically by detecting the user’s presence. To detect the user’s presence, we suggest a face-recognition mechanism.
Bibliography


