CRêPE: a System for Enforcing Fine-Grained Context-Related Policies on Android

Mauro Conti*, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich

Abstract—**Current smartphone systems allow the user to use only marginally contextual information to specify the behaviour of the applications: this hinders the wide adoption of this technology to its full potential. In this paper, we fill this gap by proposing CRêPE, a fine-grained Context-Related Policy Enforcement System for Android. While the concept of context-related access control is not new, this is the first work that brings this concept into the smartphone environment. In particular, in our work a context can be defined by: the status of variables sensed by physical (low level) sensors, like time and location; additional processing on these data via software (high level) sensors; or particular interactions with the users or third parties. CRêPE allows context-related policies to be set (even at runtime) by both the user and authorized third parties locally (via an application) or remotely (via SMS, MMS, Bluetooth, and QR-code). A thorough set of experiments shows that our full implementation of CRêPE has a negligible overhead in terms of energy consumption, time, and storage, making our system ready for a production environment.**

Index Terms—**Android Security, Smartphone Security, Context Policy.**

I. Introduction

In the world, there is an average of almost one mobile telephone per human being (with small differences between developed and developing countries). The computational capabilities of mobile phones have increased significantly in the last years, leading to so called smartphones. These devices (just "phones" in this paper) can actually run applications in such a way that is similar to desktop computers. However, because of the specific characteristics of smartphones (user mobility and communication features among others), the security and privacy of these devices is particularly exposed [1]. These challenges reduce the users' confidence and make it more difficult to adopt this technology to its full potential. To alleviate this problem, researchers have recently focused on enhancing phones' security models and their usability.

One significant challenge in the security of smartphones is to control the behaviour of applications and services (e.g. WiFi or Bluetooth). In several smartphone systems the behaviour of the applications is completely under the control of a centralized entity (e.g. once an application is installed, the user cannot control its behaviour). For example, Apple has complete control on the applications installed on iPhone devices. In fact, the only way to install applications onto a (non rooted) iPhone is by downloading them from the Apple App Store. And in turn, in order to appear in the App Store, an application has to pass an Apple vetting procedure.

However, even in systems where the user can control the behaviour of the applications, this is still mostly based on policies per application (non system-wide), and policies are set only at installation time. For instance, in the J2ME platform each MIDlet suite uses a JAD (Java Application Descriptor) file to provide the device at installation time with access control information. Similarly, in Android [2] an application developer declares in a manifest file all the permissions that the application must have, in order for it to access protected parts of the API and to interact with other applications. At installation time, these permissions are granted to the application based on its signature and interaction with the user [3]. While Android gives more flexibility than J2ME or other systems (the user is at least notified about the resources that the application uses), granting permissions all-at-once and only at installation time is still a coarse-grained control: the user has no ability to govern how the permissions are exercised after the installation. As an example, Android does not allow policies that grant access to a resource only for a fixed number of times, or only under some particular circumstances. Meanwhile, to protect users' privacy, the current security models restrict trusted third parties' control over mobile phones. Typically, only the device manufacturer and the network provider have control over the smartphone. There are no mechanisms to allow other authorized parties (e.g. a company that provides a smartphone to its employee or the private owner) to have full control over the behaviour of the phone.

Hence, there is a need for a system that will help the user to enforce the policies she defines, and help her to comply with the policies specified by authorized third parties. The following examples can be scenarios for which having a practical solution might extend the usability of the phone:

- A user might want her Bluetooth interface to be discovered when she is at home or in her office, not otherwise.
- A user might lend her phone to a friend, while the user does not want her friend to be able to use some applications or to have certain data available (e.g. SMSs).
- For privacy purposes a user might want to automatically restrict access to some data under certain conditions. For example, a weather forecast application might be prohibited to send out user's location when she is at home.
- User's smartphone might be a part of corporate infrastructure. However, the company wants to control its usage and the security, e.g. prohibit phone usage during meetings or forbid certain services when the employee is abroad.
- A smartphone can be used as a context detector compo-

Mauro Conti and Earlence Fernandes are with Vrije Universiteit Amsterdam, NL, {mconti,earlence}@cs.vu.nl; Bruno Crispo and Yury Zhauniarovich are with Università di Trento, IT, {crispo,zhauniarovich}@disi.unitn.it.
* Corresponding author.

nent, e.g. for fleet management, to associate drivers and vehicles and know vehicles' location and conditions.

We observe that currently there is no smartphone system that is able to handle this behaviour. In particular, there is no system that incorporates all features at once: definition and identification of fine-grained and dynamic contexts, policy enforcement (i.e. make sure that the system is compliant with the behaviour described by the policy) in a system-wide manner, acceptance of incoming settings [4], [5] at runtime—including the verification of the trusted third party sending management messages.

Contribution. In this work, we fill the exposed gap by presenting CRêPE, a fine-grained Context-Related Policies Enforcement for Android [6]. This work is developed based on our previous work [7], in which we presented the idea for the first time and described a possible architecture and a proof of concept implementation.

In the current work, we propose a new architectural design of CRêPE, separating our system into several loosely coupled modules. This separation gives CRêPE high flexibility of usage and openness (to other developers). For example, a context can be defined as a boolean expression that can take as input: the data reported by low level physical sensors (e.g. location, time, temperature, noise, light), processing on these data performed with high level software sensors (e.g. to determine whether the user is running, by using data from the accelerometer), or a particular interaction with authorized third parties. As for openness, other developers can design (and integrate in CRêPE) new high-level sensors—that might be required to fit new or very specific needs. For example, a new high level sensor could be designed to notify CRêPE when the phone has in its neighborhood a fixed number of other devices (e.g. as an indication that the user is in a crowded place). Also, parsers for different policy specification languages (in addition to Ponder [8], the one we already considered) can be easily integrated in CRêPE.

We clarify that in our work we assume the user is not malicious: she either wants to directly set the policy, or she wants to obey the policies set by authorized parties (e.g. her company). Hence, the security threats do not come from the phone users, but rather from malicious applications installed on the phone, or from unauthorized third parties that try to exploit the capability of CRêPE to process incoming messages. Furthermore, we observe that CRêPE is a modification of the Android operating system itself. Hence, removing CRêPE is not just as simple as removing an application, and its removal would result in a non working system.

We run a thorough set of experiments, whose results show a negligible overhead in terms of energy, time, and storage. This proves that our system is ready to be used in a production environment.

Roadmap. Section II gives an overview of the related work in the area, and describes the Android system. Section III presents the basic architecture of CRêPE, while Section V discusses the main peculiarities of our CRêPE implementation. Section IV illustrates the language used for the definition of contexts and policies. Section VI reports on a thorough set of experiments we run to assess the overheads caused by CRêPE. Finally, Section VII gives some concluding remarks.

II. Related Work

The increasing popularity of smartphones attracts OS designers to mobile platforms. The leaders in this market, such as Microsoft, Apple, Google, RIM, and Nokia proposed their own solutions. Moreover, different companies have joined together in alliances and projects (e.g. OpenMoko [9] and OMTP [10]), that aim to produce secure and usable mobile devices. The variety of the producers of mobile OS leads to a variety of architectural solutions. While one group supplies *closed systems* (e.g. Windows Mobile, Apple iOS), where Microsoft and Apple have complete control on the third-party application development and distribution, the second group (Google, RIM, Nokia) provides more *open systems*, where users have more control over the third-party applications.

When installing applications on the Apple iPhone, all the applications receive access to the same set of phone capabilities. At runtime, an application explicitly asks to the user permission for a particular functionality, e.g. GPS data [11]. A similar approach is adopted by Microsoft for Windows Phone 7 [12]. The developer has to define which capabilities her application has access to. During the installation, the system creates a sandbox which has access only to the specified capabilities. During the first usage of an application, the user is explicitly asked to grant access to restricted resources [13]. In general, these closed systems enforce policies, only, at installation time.

Open systems provide more fine-grained policy enforcement. As an example, the Java MIDP 2.0 security model restricts the use of a sensitive permission (e.g. network access) depending on the protection domain the application belongs to [14]. Similarly, the Symbian system gives different permissions to Symbian-signed programs [15]. There have been proposals to enforce more fine-grained policies at runtime. For instance, in [16] the authors present a system that allows users to define permissions for each application.

Among open systems, the most popular operating system is Android. In this system, during the installation of an application the user can agree or disagree with the required permissions. If the user does not want to grant these permissions, the application will not be installed. The developers of third-party applications are empowered to use the full capabilities provided by the system. At the same time, they should request only the necessary permissions. Otherwise, it is more likely that the user will not install the application. To help the users define which sets of permissions can be dangerous, a system called Kirin [1] was developed. Kirin can warn the user about an application that may implement dangerous functionality during the installation of this application.

Due to the lack of access control support at runtime in Android, several approaches on enforcing fine-grained policies at runtime have been also proposed [17], [18], [19]. In [17] the authors propose Saint, an installation- and run-time application management system for Android [2]. The authors start from this observation: Android protects the phone from applications that act maliciously, but provides severely limited infrastruc-

ture for installed applications to protect themselves. Leveraging on this observation, the authors built Saint in order to allow Android to be able to enforce application policies. Nauman et al. [19] propose a context-related policy enforcement system that can impose: resources-usage constraints that are determined by the context of a user/application, and resource-usage constraints that depend on the usage of this resource by the application. Bai et al. [18] have adopted UCON model to provide a continuous context-aware usage control framework for Android. The authors propose the ConUCON tool, which uses spatial and temporal context information to increase the user's privacy and the control of resource usage.

More recent papers [20], [21] concentrate on the protection of user's private data. In [20] the proposed system enables possibilities to limit the access of the installed applications to data (SMSs, contacts, calendar, location and device ID), and to the components of the Android OS (access to the Internet and broadcast intents). This approach is widened in [21]—the authors provide the user with the ability to define the accuracy level of the information revealed to the application.

To the best of our knowledge, the first solution to enforce context-related policies in Android has been proposed in our previous work [7]. In [7] we described a preliminary design and implementation of CRêPE, and its functional requirements. In particular, [7] is the first solution with the ability to enforce fine-grained Context-Related Policies on Android. Differently from Saint [17] (that focuses on application policies), CRêPE aims to enforce fine-grained context-related policies defined by the user (or other parties). Furthermore, the policies can be applied also in a system-wide manner, and can be set on the phones also at runtime from both users and authorized third parties. Differently from Apex [19] or other systems like the one in [18] (that focused on the providing user-centric policies) CRêPE provides the ability to enforce policies from trusted parties (which includes the user). This implies also an important functionality of CRêPE: the capability to resolve possible conflicts between policies coming from different (authorized) parties. Moreover, CRêPE can change the policies at runtime (not just at installation time, as it is for other systems—e.g. Apex).

A. Context-based Access Control Models

Researchers have already shown interest in context-based access control, even if the meaning of "context" can be very different (see [22], [23], [24], to cite a few). The concept of context that we consider in our work is similar to "environment roles" used in [25], which in turn has been specialized in [26]: accounting for the specificity of spacial information (e.g. multi-granularity of the position; spatial relationships that may exist between spatial elements in space). There are a lot of other works that uses context information [27], [28], [29], [30], [31]. The recent examples from the industry have shown that context aware security is a prominent area of research. For instance, VMWare [32] is developing a virtualization technology that separates personal and corporate parts of smartphone. Finally, we also note that CRêPE shares a common element with the access control model of web services [33]: where policies depending on a context might also be specified, when a proper access control model is provided [34].

In this paper, we have focused on a complete design and discussion of our approach, together with a thorough practical evaluation. Thus, we have completely redesigned the architecture of CRêPE, although many functionality remains the same as in [7].

B. Android Security Overview

We considered Android as reference platform because of its increasing adoption by manufacturers, developers, users, and researchers [1], [3], [17]. In this section, we give an overview of the Android security model [2], [3].

Android combines two levels of security [3], [35]: at Linux system level and at application framework level. At the former, each application is executed in a separate user process, within its own isolated address space (sandboxing). At the latter, as discussed in [3] Android provides Mandatory Access Control (MAC) [36] to application components, through the Inter-Component Communication (ICC) reference monitor. In fact, as opposed to discretionary access control, a component is not capable of passing its permission to other components. Protected features are assigned with unique security labels—*permissions*. To make use of protected features, the developer of an application must declare the required permissions in its package manifest file—`AndroidManifest.xml`. The protection level can be *normal* (these permissions are granted automatically), *dangerous* (the user has to explicitly grant these permissions), *signature* (calling and called applications must be signed with the same key) or *signature or system* (the applications should be signed with the system key). When the user has the chance to take a decision (protection case: *dangerous*), she has only two choices: either grant the requested permissions or refuse to do this. In the latter case, the application will not be installed. At application runtime, Android has no mechanism to modify permissions.

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

III. CRêPE

In this section, we describe the access control model, the architecture, the components, and the main algorithms of CRêPE. More precisely, in Section III-A we present the access control model of CRêPE, in Section III-B we give an overview of the CRêPE system. In Section III-C we consider its system architecture and the main building blocks. After that, we discuss Context Detection peculiarities in Section III-D. Section III-E explains our policy management and the fundamental algorithms of CRêPE.

A. CRêPE Access Control Model

Before discussing the implementation challenges of our proposal in the Android system, we provide here the Access Control Model of CRêPE. The model is illustrated in Figure

1, using standard concepts from XACML [37], [38]. Also, the model in Figure 1 can be considered as an instance of the UCON model [5], where obligations are enforced by the Action Executors, and the conditions are set by the Context Detector. We observe that while our model has similarities with several access control models involving contexts (e.g. [34]), in our model policies are associated to contexts, and the dynamic activation/deactivation of contexts (that determines which policies have to be enforced) is detected automatically by the sensors present in every modern smartphone. Overcoming existing limitations of Android, our model allows to change and/or adapt policies also at application run-time and not only at installation time. Furthermore, CRêPE supports dynamic policy management, thus at any time the administrator can set, delete and modify new contexts/policies at runtime. If such changes require the system to invoke/stop existing service/application (e.g., disable internet while entering in the meeting room not only require to prevent new connections but also require to shut down existing connections), this is supported as well by the model.

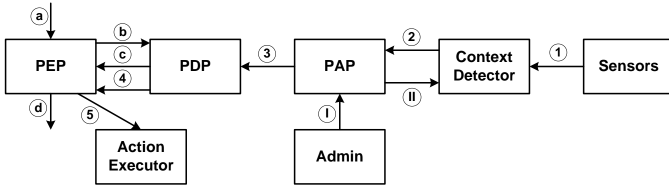


Fig. 1. CRêPE Access Control Model.

In particular, there could be three different flows at runtime, each identified by the different labeling of the arrows. The letters *a*, *b*, *c*, and *d* show the processing flow of a runtime access request: the request is intercepted (arrow *a*) by the Policy Enforcement Point (PEP); this, in turn asks (*b*) the Policy Decision Point (PDP) whether the request should be granted; based on the answer (*c*), the request will be granted (*d*) or not. The roman numbers *I*, and *II* show the flow of processing due to administrative commands. In fact, authorized parties can set contexts and associated policies in the system (arrow 1). As a consequence of this, the Policy Administration Point (PAP), notifies the Context Detector that the newly set contexts need to be monitored (*II*). The arabic numbers 1, 2, 3, 4, and 5 indicate the flow of processing initiated by context becoming active/non active. The Context Detector continuously monitors the environment via phone sensors (arrow 1). As soon as a registered context (set as described before) becomes active/non active, the Context Detector notifies the PAP (2) that has to activate/deactivate the new policy (composed by access control rules plus obligations). From all the contexts that are currently active, the PAP decides (e.g. resolving conflicts) the set of rules that need to be enforced. Hence, this information is passed on to the PDP (3). PDP stores the information related to access control, while forwards to the PEP the obligations (4). PEP, in turn, is in charge to take the actions specified by the obligation policies, this is done via a component that we call Action Executor (5).

B. Overview

CRêPE acts as a security mechanism in addition to the standard Android security mechanisms. It allows users and other predefined trusted parties to define context-related policies, which can be installed, updated and applied also system-wide at runtime. Alternatively, these policies can be applied in a fine-grained manner, e.g. for each application. A context-related policy is composed by two different type of policies:

- (i) an access control policy—composed of access *rules*;
- (ii) an obligation policy [5], [39]—that specifies *actions* (i.e. start or stop an application; activate or disable a system resource, like the camera).

Since there could be many policies and context providers, it is possible that several contexts fulfil current conditions (i.e. “being in Italy” and “being in Trento”). We call these contexts as *Active Contexts*, and the policies corresponding to these contexts as *Active Policies*. To resolve possible conflicts that may raise in the access control policies, we have introduced the *Union and Conflict Resolution* (*UCR*) function, which is discussed in Section III-E. The result of this function is the resolved union of active policies called the *Currently Enforced Policy* (*CEP*).

It is worth noting that *ActionExecutor* allows CRêPE to enforce ongoing Obligations [5], e.g. pause downloading when entering in a meeting room where connectivity is not allowed.

C. Architecture

CRêPE is implemented as a modification of Android. In fact, it consists of its own components integrated in the Android stack, as well as modified components of the Android Framework. The architecture of CRêPE is summarized in Figure 2, where dashed boxes and underlined names clarify the mapping with the model depicted in Figure 1.

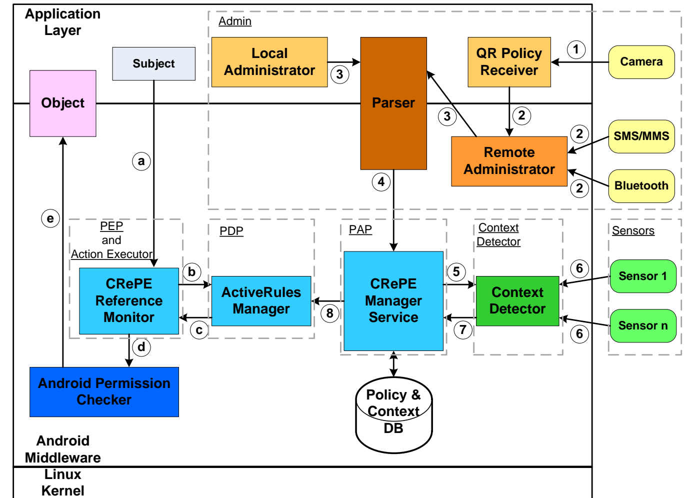


Fig. 2. CRêPE Architecture: steps labeled with numbers represent the system management procedure; steps labeled with letters represent enforcement procedure.

There are two main entry points for CRêPE policies: *LocalAdministrator* and *RemoteAdministrator*. *LocalAdministrator* is an application by means of which the owner of the device

can manage CRêPE via a GUI. *RemoteAdministrator* allows authorized third parties to manage CRêPE remotely, using SMS, MMS, Bluetooth, and QR-code (steps 1 and 2 in Figure 2). Once management messages are in the system, they are parsed by *Parser* (Step 3), which is partially implemented as a standalone application. In fact, the component of the *Parser* that actually parses policies can be easily replaced, to let our system understand policies defined in different languages. As shown in Figure 2, all these components that allow the administration of CRêPE correspond to the Admin component in the model described in Figure 1.

CRePEManagerService is a central component of our system (and corresponds to the PAP of the model in Figure 1). It is responsible for general system management. It receives a policy with an associated context (Step 4), stores them into the *PolicyAndContextDB* database, and asks *ContextDetector* to start monitoring (Step 5) the received context (*ContextDetector* is described in Section III-D). If *ContextDetector* detects (Step 6) that a context is activated or deactivated, it notifies *CRePEManagerService* about this (Step 7). *CRePEManagerService* calculates the *Currently Enforced Policy* (*CEP*) that regulates access control, and loads it into *ActiveRulesManager* (Step 8).

For the enforcement procedure, the first thing to be noticed is that when a policy becomes active, *ActiveRulesManager* (which correspond to the PDP of the model in Figure 1) notifies the *CRePEReferenceMonitor* (the PEP) about the actions specified by the activated obligation policy. A specific component within *CRePEReferenceMonitor*, *ActionExecutor*, is in charge to perform all the required actions. During system operation, when a subject tries to access an object, *CRePEReferenceMonitor* intercepts this call (Step a), and checks if the subject can access the object, according to *CEP* loaded into *ActiveRulesManager* (steps b and c). If the access is not allowed, *CRePEReferenceMonitor* prohibits further interaction. Otherwise, *CRePEReferenceMonitor* simply passes the call to the standard Android Permission Checking mechanism (Step d).

D. Context Detection

With CRêPE it is possible to specify the behaviour of a phone, depending on its current context. In particular, the behaviour is specified by couples $< C, P >$, where C is a context, and P is the policy (including access control rules and actions) associated with C. A context C can be active or inactive (depending on whether the conditions that define the context are satisfied). When the context C is active, the corresponding policy P is also active, i.e. the behaviour specified by P is enforced by CRêPE. At any time t, n couples of contexts and policies can be stored in CRêPE ($< C_1, P_1 >$, ..., $< C_n, P_n >$). At the same time, a subset of $n' \leq n$ of these contexts and policies can be active. Furthermore, we underline that CRêPE does not pose any restriction on the definition of two different contexts, e.g. a context can be also subsumed by another context.

CRêPE supports both physical contexts (i.e. location, time, online), which are associated to physical sensors (i.e. GPS, clock, Bluetooth, etc.), and logical contexts, which are defined by functions over physical sensors. Examples of logical sensors are those that tell whether "the user is running in an open space" (defined using the two physical sensors, location and accelerometer), or whether the user answering the phone call is authorized to do so [40].

Contexts are defined as boolean expressions over physical and logical sensors. *ContextDetector* contains those expressions, and checks when they are satisfied. When a boolean expression becomes true, *ContextDetector* notifies *CRePEManagerService* that the corresponding context is activated. CRêPE will hence enforce the corresponding policy (see Section III-E3). In our architecture, *ContextDetector* is decoupled from the core of CRêPE. That is, *ContextDetector* does not need to know anything about policies. Hence, it is possible to develop this component independently from the other CRêPE components, or to plug other context detector components into CRêPE (e.g. ContextDroid [41]).

E. Policy Management

In this section, we describe the main concepts and the behaviour of CRêPE with respect to policy management.

1) Policies, Rules, Actions: First, we introduce the concept of a *policy* and a *rule*. We can think of a policy P as a matrix (Figure 3), where the indexes of the rows are Subjects S (i.e. applications) and the indexes of the columns are Objects O (i.e. applications, and system resources like camera and Bluetooth interface). Within the matrix, the rule $R =< Access/Deny, Priority >$ corresponding to the subject S and the object O is identified as $P(S, O) = R$. A rule specifies: an *access* mode, which is whether the corresponding subject is allowed or denied to access to the corresponding object; and a *priority*, which is a number used to resolve conflicts when colliding rules apply to the same combination of a subject and an object. A context policy also includes an obligation policy to specify actions. The obligation policy can be seen as a simple vector of objects (not shown in the picture): for each object the action might specify to start or stop the object (i.e. the application or system resource corresponding to the object index).

Each context (and its corresponding policy, made of access control rules and actions) is defined by an authorized entity. We refer to this entity also as the *owner* of the context (policy). The owner can assign to each single rule or action a priority number. In particular, each owner has an associated maximum priority number: in her rules and actions, she can specify a priority that is at most equal to her assigned maximum priority. When a policy is installed on the phone, CRêPE checks that this constraint is not violated. CRêPE first verifies the validity of the certificate of the owner: the certificate includes the identity, the public key, and the maximum priority number, all these signed with the key of the certification authority. Then, CRêPE checks that all the priority numbers in the specified rules and actions are at most equal to the max priority number stated in the certificate.

2) Policies Activation and Deactivation: When a context (and its corresponding policy P) becomes active, CRêPE has to perform some operations. Protocol 1 describes the procedure that runs when a new policy P becomes active, assuming

CEP is the currently enforced policy. Basically, CRêPE has to integrate *P* in *CEP*. This is done by building a list of policies (Protocol 1, Line 2)—where *CEP* is the first element of the list—, and invoking the *UCR* function (Line 3) on that list. How the *UCR* function works will be explained later.

Protocol 1 $Activate(P)$

1: Add P in $ActiveP_{list}$ \\ $ActiveP_{list}$ is the list of all active policies
2: $TempP_{list} = <CEP,P>$ \\ CEP is the first element of $TempP_{list}$
3: $CEP \leftarrow UCR(TempP_{list})$

When a context (and its corresponding policy) becomes inactive, CRêPE basically has to recompute *CEP* based on the policies that are still active. This is done as described in Protocol 2, again making use of the *UCR* function. We note that, differently from the activation scenario, in this protocol the *UCR* function is called (Protocol 2, Line 2) on the list of all the policies that remain active after removing the one just deactivated (Line 1).

Protocol 2 $Deactivate(P)$

1: Remove P from $ActiveP_{list}$
2: $CEP \leftarrow UCR(ActiveP_{list})$

3) Defining the Currently Enforced Policy (CEP): Each context has an associated policy. However: 1) several contexts might be active at the same time; 2) policies might specify conflicting rules (e.g. a policy allows a subject to access a given object, while another policy does not allow this). Hence, CRêPE has to enforce a policy that is the result of the union of all the policies ($P_1,\ldots,P_N$) corresponding to all active contexts. Meanwhile, all conflicts among these policies have to be solved. We refer to the resulting policy as the *Currently Enforced Policy* (*CEP*). The computation of *CEP* is done by the *Union and Conflict Resolution* (*UCR*) function, as illustrated in Figure 4.

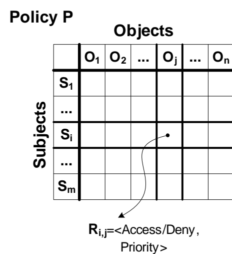


Fig. 3. Policy and Rules.

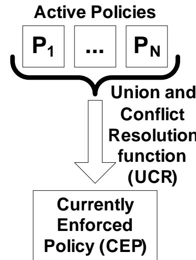


Fig. 4. Currently Enforced Policy (CEP).

In CRêPE, we implemented the *UCR* function as described in Protocol 3. First of all, *UCR* initializes *CEP* as the first policy element in the list (Protocol 3, Line 2). Then, it adds to *CEP* subjects and objects that are present in other policies but not already in *CEP* (Protocol 3, Lines 4-15). After this, *CEP* is filled. This is done using the $ConflictResolution$ function (Protocol 3, Line 19), which takes as an input a set of rules (referring to the same combination of subject and object; see Line 18) and computes the rule that is the outcome of the resolution of all the rules in the set.

Protocol 3 UCR(P_{list})

```
1: P_1,...,P_N ← P_list
2: CEP ← P_1
3: \\ Add indexes for Subjects and Objects in CEP
4: for all P_i ∈ P_2,...,P_N do
5:     for all S ∈ Subjects(P_i) do
6:         if S ∉ Subjects(CEP) then
7:             Add S in CEP
8:         end if
9:     end forall
10:    for all O ∈ Objects(P_i) do
11:        if O ∉ Objects(CEP) then
12:            Add O in CEP
13:        end if
14:    end forall
15: end forall
16: \\ Set the rules for CEP
17: for all (S,O) ∈ CEP do
18:     R ← ∪P_i(S,O)
19:     CEP(S,O) ← ConflictResolution(R)
20: end forall
```

Protocol 4 describes how $ConflictResolution$ works with respect to a set $\mathcal{R}$ of (potentially) conflicting rules. We assume that each policy *P* that becomes active is processed independently from the others, in order to insert its rules in the Currently Enforced Policy (*CEP*). This protocol first selects the rules with the highest priority (Protocol 4, Lines 2-3)—hence, following the principle of least privilege. Thus, if there are several rules with the same highest priority number, it checks if there is among them a deny rule (Protocol 4, Line 4). In this case, it returns a deny rule (Line 5). Otherwise, it returns an allow rule (Line 7).

Protocol 4 $ConflictResolution(\mathcal{R})$

```
1: R_1,...,R_N ← R
2: max_priority = Max(priority_i) \\ R_i =< access_i, priority_i >
3: R′=Select R_i ∈ R with priority_i=max_priority
4: if ∃ R_i ∈ R′ such that access_i='deny' then
5:     return R =< max_priority, deny >
6: else
7:     return R =< max_priority, allow >
8: end if
```

Concluding, both $Activate$ and $Deactivate$ (Protocol 1 and 2, respectively) use *UCR* (Protocol 3), which in turn uses $ConflictResolution$ (Protocol 4). As a result, the computation and storage complexity of $Activate(P)$ (as well as $Deactivate(P)$) is $O(|ActiveP_{list}| \cdot |S| \cdot |O|)$, i.e. it depends on three key factors: the number of active policies, the number of subjects, and the number of objects in the system—when the procedure is called. Since all these three variables are bounded by small constants in practical systems, the computation time and storage is negligible, as confirmed by our experiments (Section VI-B).

IV. CRêPE Language

In this section, we describe the incoming messages that CRêPE can handle. No matter the way a message comes into the phone (e.g. via SMS or MMS), after the system opens the outermost packet (e.g. the one for handling the

SMS), it obtains what we call the *CRêPE packet*. This packet is specified using XML. The format of a CRêPE packet is shown in Figure 5, where `type` specifies whether the packet is referred to:

(i) the specification of a context together with its associated policy (`type=policy+context`). Both are stored in the CRêPE database. The policy is activated/deactivated depending on the status of the corresponding context.
(ii) a policy specification (`policy`). The reason for this type of message is to get a policy into the phone. The policy is not associated to a context: the only way to activate (or deactivate) this policy is via commands.
(iii) a CRêPE command (`command`): i.e., an instruction referred to policies or CRêPE database (commands are described in Section IV-C).
(iv) a command with a policy (`command+policy`). In this case, the policy is sent together with the command of activating it.

In Figure 5, `CONTEXT_DEFINITION` and `POLICY_DEFINITION` are placeholders for the definition of a context and a policy, respectively. Finally, `SIGNATURE_STRING` represents the signature of the sender of the message. `CERTIFICATE_STRING` represents the certificate of the sender (it can be presented as a whole, or just as the ID of a certificate that is in the proper cache).

A. Context Specification

A context is defined via a simple boolean expression (AND, OR, NOT), with mathematical comparisons ($<, >, =, !=$) involving objects that refer to sensors (which can output boolean value or real numbers). For example, the context defined as `(Time>8)AND(Time<16)AND(isRunning=True)` becomes active if the user is running between 8am and 4pm.

B. Policy Specification

In order to be compatible with the current standards of policy specification, we implemented a policy parser for (a subset of) the Ponder language [4], [5]. The parser is designed as an independent component: CRêPE can be easily extended to understand other policy specifications, just by installing a proper parser. Ponder is a declarative, object oriented language for specifying security and management policies. In particular, it allows general security policies to be specified as a set of rules. Hence, we found Ponder appropriate to CRêPE for a simple subject/object specification. Furthermore, Ponder also support usage control policies [5] and obligations that we need to specify actions (e.g. close an application, or start another one). In Figure 6, we report an example of the specification of a policy (i.e. a possible substitution of `POLICY_DEFINITION` placeholder of Figure 5). In this example, the policy consists of only two rules: the first one allows all the subjects in *CEP* to access the Internet (this rule has priority 11); the second rule forbids the music application to use the Bluetooth interface (this rule is with priority 10).

```
<?xml version="1.0"
    encoding='UTF-8'?>
<crepepkt type=POLICY_TYPE>
<context name=CTXNAME>
CONTEXT_DEFINITION
</context>
<policy>
POLICY_DEFINITION
</policy>
<signature>
SIGNATURE_STRING
</signature>
<certificate>
CERTIFICATE_STRING
</certificate>
<crepepacket>
```

Fig. 5. XML message example.

```
<policy>
auth my_policy_id_string[+]{
subject *;
  object android.permission
                .INTERNET;
  action allow;
  priority 11;
subject com.android.music;
  object android.permission
                .BLUETOOTH;
  action deny;
  priority 10;
}
</policy>
```

Fig. 6. Policy example.

C. CRêPE commands

A command is a self contained instruction that is sent to the phone. CRêPE supports the following commands to handle policies: `ACTIVATE <POLICY_ID>`, `DEACTIVATE <POLICY_ID>`, and `DEL <POLICY_ID>`. They can be used to activate, deactivate, and delete the policy specified by `POLICY_ID`, respectively. The policy itself can be either: (i) already in the system, or (ii) come together with the message that contains the command itself. In the latter case, `POLICY_DEFINITION` component on the received message is also filled. Finally, `DEL *` can be used to reset CRêPE, i.e. reset *CEP*, and the Policy & Context database.

V. Implementation

In this section, we describe the most important implementation details of CRêPE, which is a modification of the Android OS. The entire system is contained in 4,830 lines of code, added to the base Android system (also referred to as stock Android), in addition to changes to existing Android system components. In particular, our code is based on the standard AOSP (Android Open Source Project) [42] which we forked off in December 2010. CRêPE implementation impacts system services, framework data structures, and system applications. For the policy-parsing functionality we have ported the ANTLR Java Runtime [43] to Android. The system is available for download at [6].

A. CRêPE components

CRêPE operates on each of the three levels of abstraction of the Android software stack (see Figure 2): the User Level, the Framework Level, and the Kernel Level (for Internet access regulation, not shown in the figure), with the bulk of logic in the framework. The system depends on a few base data structures that must be protected from all processes except the system process. One of these data structures is the CRêPE central database, which is managed (as explained later) by the *CRePEDatabaseManager* component. We make use of standard Unix-like access permissions to protect this SQLite database file. Another relevant data structure is the cache of certificates. During the boot of the Android Runtime, we create a `/data/crepe` directory to hold our data structures and set appropriate permissions, so that only the `system_server` process can read/write this directory.

CRePEManagerService: (also named *CMS*; it works at Framework Level). *CMS* is the core of the system. It is responsible for the orchestration of all tasks of CRêPE. *CMS* encapsulates the database manager (which is the only component that talks to the CRêPE database described earlier) and the access rules matrix. It serves as a callback point for context detection. Finally, it also contains the code for the CRêPE permission check which is hooked from the Activity Manager Service. All policy resolution algorithms to calculate *CEP* are contained here.

ContextDetector: (Framework Level). This component is responsible for context detection. *CMS* registers for callbacks from *ContextDetector*. The responsibility of this component is to notify *CMS* when a particular context is activated or deactivated. *ContextDetector* combines inputs from several sensors, which can be physical sensors (time, GPS, accelerometer, orientation) as well as logical sensors (e.g. the one that detects whether a user is running).

Authenticator: (Framework Level). This component performs all cryptographic operations required by CRêPE, e.g. the certificate and signature verification for commands coming from third parties. We support X.509 format certificates. The *Authenticator* component works in co-ordination with *CertificateCache*, which caches the certificates. This behaviour results in a smaller size of incoming commands: if a corresponding certificate is already in the cache it does not need to be sent together with the command.

PonderPolicyParser: (User Level). We have implemented a parser for a small variation of the Ponder Policy Specification Language [8]. It resides in a service exposed by an APK (android package). The parser must register itself with *CMS*. With this mechanism, we have a flexible solution: it is possible to make the system understand a totally different policy language just by installing a proper APK.

CrepeReaper: (Framework Level). *CrepeReaper* is responsible for shutting down the processes in accordance with currently active policies. In particular, we first check if the process is in background (i.e. it is not at the top of the Activity Stack). If it is so, the process will be just killed. Otherwise (if the process is the one in foreground), we launch a "decoy" activity which forces the previous activity to be pushed to run in background. This, in turn, forces the execution of `onPause()` in the Activity lifecycle, which gives developers a chance to gracefully save the process state. We then terminate the process and the "decoy" as well.

CRePEIPTables: (Kernel Level). It communicates with `iptables` (userspace Linux application program), which in turn manages the `netfilter` modules. *CRePEIPTables* is hence used to setup firewall rules for network access.

B. Access Regulation

We place the hook for CRêPE checks in the regular Android check. The public method `checkPermission(Permission, ProcessID, UserID)` inside `ActivityManagerService` is the only public entry point for all permissions checking. Inside `checkPermission` method, and before the logic of the Android permission check, we invoke our own `checkCrepePermission`. If the particular operation is allowed by *CEP*, the system performs a normal Android permission check. We implemented the access matrix as a Hash Map, which gives us an efficient access time (just constant in most cases) to a particular (subject, object) combination. In the following, we give some details how the access is regulated for specific objects.

Applications can act as subjects and objects in our model. When an application acts as a Subject, we make use of Binder API to discover its UID (i.e. the UID of the caller process). An application acts as an Object when it is to be started. During the installation and reinstallation of an application it can change the UID. For this reason, in our policies we use package names, which are unique across the system. During the uninstallation of an applications, we remove the row and the column of the UID corresponding to our application from *CEP* (we can do this using the function that transforms a package name into the UID). During the installation of the application we reset *CEP* and run the *UCR* function for all currently active policies.

Restricting access to the Internet in the Android framework is done through permissions which control whether a process is part of the `inet` group. In fact, only applications that are included into `inet` Linux group (granted with permission `android.permission.Internet`) have access to the Internet. This cannot be changed at runtime. To create/remove rules at runtime, based on UIDs to drop or to forward packets, we use *IPTables* (based on `netfilter`). For Bluetooth, Permission check hooks are placed in `BluetoothSocket` and `BluetoothServerSocket` methods, which at first consult with *CEP* before establishing any connections. Resources protected by Android permissions (like the camera, the microphone, or application components) are also protected by Android permissions—these permissions strings acts as Objects in our system.

C. System Management

In this section we describe how to manage CRêPE. In particular, this can be done both locally and remotely by all the authorized parties, including the user.

1) Local management: Local management is done via the *LocalAdministrator* component. It includes a GUI to create new policies and activate/deactivate them. With this GUI (whose screenshots are not reported for space limitation) the user can define a context and its associated policy, i.e. a set of rules. For each rule, the user can specify: subject (including "*", i.e. any) and object involved, rule type (Allow/Deny), and Priority number. The user can also manually activate or deactivate policies already defined on the phone. Deletion and modification is also supported. While in the current implementation the context (e.g. a location) must be defined via a textual interface specifying a boolean expression (e.g. with *Latitude*, and *Longitude* variables), we are working to make the specification of the area more user friendly, like drawing an area on a map. Access to *LocalAdministrator* is protected via a password—which is stored in the CRêPE central database.

2) Remote management: The trust architecture for remote management (via messages sent to the device) is done via a Public Key Infrastructure (PKI). An incoming message for CRêPE has to come with the certificate of the sender. A certificate can be transmitted in-band or just as an ID (corresponding to a cached certificate). All certificates should be in the X.509 format. We use standard Java APIs to manipulate and verify certificates. The CA certificate is embedded in the system image at build time. All other certificates are cached in the `/data/crepe/certificates` directory. The algorithm used for signature is SHA1 with RSA and a 2048-bit RSA public key. For all the algorithms, we use the BouncyCastle APIs—as also done by Android itself.

Messages can be sent as SMS, over Bluetooth and as QR-codes.

VI. System Evaluation

This section is devoted to the evaluation of CRêPE. We first discuss the effectiveness of its security (Section VI-A). Then, we report and discuss the experimental evaluation of its efficiency[1] (Section VI-B).

A. Effectiveness: Security

We recall that we assume the user is non malicious, while the security threats for our system come from: (i) malicious applications; (ii) or even from unauthorized third parties that try to exploit the communication system of CRêPE. Although CRêPE could be extended to enforce security against a malicious phone user (e.g. based on solutions like ARM TrustZone [44]), this is out of the scope of this work.

Considering malicious applications, we observe that CRêPE does not reduce the security of Android, though it can improve its security significantly in several cases. We first discuss why CRêPE does not reduce the security of Android. In fact, for each requested access to an application or system service, CRêPE only adds further checks, i.e. its own checks that depend on the active CRêPE policies. Each access that is not denied by CRêPE is passed on to the Android Permission Check and not influenced by CRêPE anymore. As a result, CRêPE can only reduce the number of accesses allowed, but not reduce the security of stock Android, because its checks on actually executed actions occurs in any case.

We observe that CRêPE can also improve the security of stock Android in several cases. For instance, let us consider a recent Android vulnerability [45]: an intruder in public WiFi networks can eavesdrop and then use for two weeks an authorisation token used by a number of applications. With CRêPE this problem could have been solved, by sending to the phone a policy restricting the use of these applications on their network. As another example of security improvement, we observe that the current delegation mechanism of Android has a weakness that CRêPE fixes to some extent. In particular, we consider the following to be a weakness. An application App is allowed to access a resource (e.g. to use the Bluetooth service). App defines a permission *Perm* for its component App_1 (that actually uses the resource). App defines *Perm* with *normal* level which is automatically granted without asking for explicit approval from the user. This would imply that any other application can use the same referred resource, while the user is not actually aware of this. To some extent, CRêPE helps to prevent this kind of compromise. A CRêPE policy could be defined by the user to limit the access to resources in some necessary situations. For instance, the user can define a policy allowing to use Bluetooth only at home or the office, which are trusted environments (in this case, the context will be detected by the CRêPE *ContextDetector* component via the GPS sensor). The policy will be applied system-wide, irrespective of which application requests access.

Another important observation is that since CRêPE has a modular architecture, the interactions between the components must be also protected. For this purpose we use standard Android mechanisms. All exported CRêPE components are protected with permissions with the protection level: *signature*. This means that only components and applications signed with the same key can access the protected components. Thus, basically only other CRêPE components can interact with these exported components. Therefore, the parts of CRêPE that work at the application level (i.e. *LocalAdministator* and *Parser*) must be also signed with CRêPE signature.

The remaining security issue to be discussed is the threat coming from unauthorized third parties that try to exploit the communication system of CRêPE. We protect the system from such type of attacks by a PKI system with X.509 certificates. The CRêPE system is installed with the certificate of a root certification authority (CA), e.g. a company will put its certificate as root authority in the phones given to its employees. For each incoming message, CRêPE checks that the message is coming either from the CA or from another entity with a certificate issued by the certification authority (e.g. a specific department of the company). New authorized certificates will be stored in the CRêPE cache. Certificates can have an expiration date or can be revoked via a specific command. Finally, to avoid replay attacks, each sent message has to carry a time stamp.

B. Efficiency: Overhead

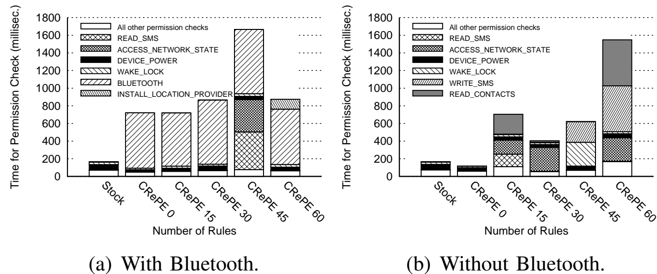
In this section, we report the results of a set of thorough experiments we run in order to evaluate the design and implementation of CRêPE. In particular, we investigated the following issues that we believe are fundamental in smartphone usage: time overhead (Section VI-B1), energy overhead (Section VI-B2), and storage overhead (Section VI-B2). For all the experiments reported in this sections we used the Google Dev 3 phone (HTC Nexus One).

1) Time Overhead: For all the experiments related to time overhead, we used a call to `System.nanoTime()` before and after the operations to be measured. In particular, we considered the points where CRêPE can add delay.

a) CRêPE permission check: Each time a subject accesses an object within the set of the "controlled" objects (i.e. the objects in the *CEP* matrix, see Section III-E3), CRêPE has to run its check. To understand what is the time overhead introduced by this CRêPE check, we ran the following

[1]Results data can be found at http://www.crepedroid.org/eval_raw.zip.

experiments. First, we considered a smartphone running stock Android. We then simulated regular phone usage by having the phone behave as follows for 120 minutes: at minute 0, and every ten minutes, the phone started a call (lasting 110 seconds), then started and closed a set of applications (in order: MMS, Contacts, Gallery, Email, Music, and Calendar). From this, we obtained a large body of check permission timing information. We measured the time spent within the Android check. In particular, we measured the time spent in checking each specific permission type. Then, we ran a similar experiment on a Dev 3 phone with CRêPE: the time spent for permission check now including also the CRêPE checks. In this scenario, we ran the experiment for different number of rules in the system: 0, 15, 30, 45, and 60. In the case of CRêPE installed, we ran all the experiments twice. First, assuming that CRêPE was able to get commands via Bluetooth (hence turning on the Bluetooth interface every 5 minutes); then without this functionality. The results of this experiment are shown in Figures 7(a) and 7(b), for Bluetooth active, and not active, respectively. To help the comparison, we report the value for stock Android in both figures.



(a) With Bluetooth. (b) Without Bluetooth.

Fig. 7. Time overhead for permission check

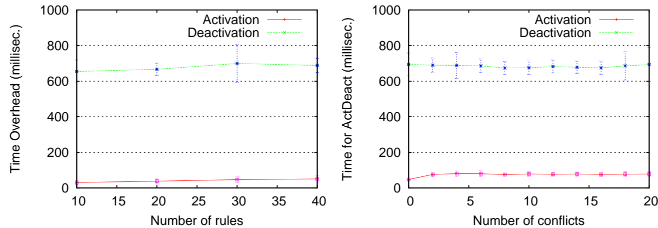
From these figures, we observe that the time overhead for both stock Android and CRêPE permission check is negligible and not noticeable by the user, no matter the specific setting. In the worst case (that is observed for CRêPE with 45 rules, and Bluetooth active), the overall time overhead during 120 minutes is less than two seconds. Another important observation is the following: the time overhead for CRêPE permission check is almost independent from the number of subjects and objects. We note that this was not the case in the previous proof-of-concept implementation of CRêPE [7]. In fact, in [7] the active rules were organized in a list, hence requiring a check time overhead linear with the number of active rules. Current CRêPE enforcement is implemented via a table of active rules (*CEP*), where lookup is constant.

From both Figure 7(a) and Figure 7(b), we note that the time overhead is not uniformly increasing with the increase of the number of rules. In fact, while adding a rule adds some overhead due to the check of this new rule, this might also decrease the total overhead. This is due to the following fact: even a single rule might change the execution path of the application. For example, let us consider the case of an application that requests access to a resource (which implies a permission check), and then it does a lot of operations on that resource (each operation implying a permission check). If it is added a new rule that avoid the application to get the resource in the first place, then all the operations on that resources will be just skipped (hence, saving all the time overhead for all the associated permission checks). This motivation is also supported by the fact that increasing the considered set of rules, the time spent in a specific permission check decreases in some cases. For example, let us consider the scenario without Bluetooth active (Figure 7(b)). If we move from the setting with 15 rules (x-axis), to the one with 30 rules (which include the previous 15 rules), the time spent in checking the `READ_CONTACTS` permission goes from more than 225ms (in over 120 minutes of experiment), to some 16ms.

b) Policy activation and deactivation: Each time a policy is activated (deactivated) as a result of the corresponding context becoming active (inactive), CRêPE needs to update the *CEP* matrix. First, we investigated the cost of activation and deactivation of a policy, assuming no conflicting rules. We started the experiment with policies P_1 and P_2 already active (P_1 having 10 rules; P_2 having 10 rules; P_1 and P_2 not having conflicting rules). Then, we considered a policy P_3 with a number of rules varying from 1 to 40. In none of these cases there were conflicting rules. For each considered number of rules for P_3, we activated and deactivated policy P_3 100 times. We measured the time overhead for each activation and deactivation. Results (average and standard deviation, s.d.) are shown in Figure 8(a).

From this figure, we first observe that the time cost of policy deactivation is higher than the one of policy activation. As expected, this is due to the mechanism described in Section III-E3. In fact, deactivating a policy implies: 1) re-initialization of the data structure for *CEP*; 2) re-activation of all the policies except the deactivated one. From other experiments (not shown in Figure 8(a)), we noticed that even when a single policy is active on the system, the deactivation of this single policy costs more than its activation (for a policy with 10 rules activation cost some 90ms, while deactivation costs some 140ms). For the activation, the cost of adding a rule is the one of a lookup in the table. In the cases when a rule is already specified for the given combination of a subject and an object, the cost of conflict resolution should be added. By implementation, having n active rules the cost for the deactivation of a rule is equal to the activation of $n-1$ rules. Finally, we also note that the number of rules in the policy that is deactivated (P_3) does not play any role in the cost for deactivation. Again, this is due to the specific implementation of the deactivation (i.e. initialization, and re-activation of P_1 and P_2). Finally, we observe that the number of rules does not have any significant impact on the time overhead both for activation and deactivation of a policy.

After the experiment without considering conflicting rules we investigated the time overhead also in this latter case. In particular, we considered the starting settings as in the previous experiment. Given this setting, policy P_3 (40 rules) was considered with a varying number of conflicts (from 1 up to 20) with policies P_1 and P_2. For each number of conflicting rules activation and deactivation of P_3 was done 100 times. Results (average and standard deviation) are shown in Figure 8(b). Observe that the first point of Figures 8(b) (number of conflicts = 0) corresponds to the last point of Figure 8(a).



(a) Influence of number of rules.

(b) Influence of number of conflicts.

Fig. 8. Time overhead: CRêPE Policy Activation and Deactivation

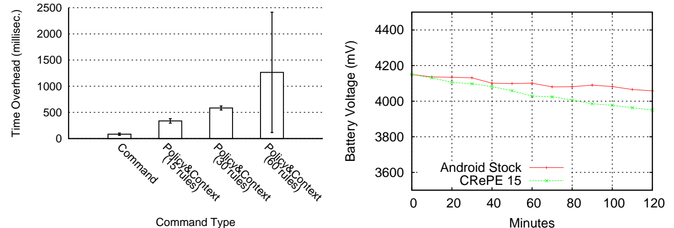


Fig. 9. Message Processing Time. Average (and standard deviation).

Fig. 10. CRêPE Energy Consumption.

From Figure 8(b), we observe that again the overhead is negligible. Also, as observed from the previous experiment (with no conflicting rules in P_3), the overhead for deactivation is constant (since the deactivated rules do not influence the deactivation process). Finally, we observe that the activation time overhead is influenced by the number of conflicting rules—while remaining negligible.

c) Incoming commands: Here we report on the investigation of the time overhead to handle incoming messages intended for CRêPE. In particular, we recall that we have four different types of incoming messages (see Section IV): (i) Command; (ii) Policy; (iii) Context and Policy; (iv) Command and Policy. For each message the time overhead is composed of two elements. The first one is the time to receive the message—hence it is dependent on the specific technology used (e.g. Bluetooth or SMS). The second one is the time for processing the received message. The processing time includes also: parsing, signature verification, and certificate verification.

In our experiment, we focused on a simple message (Command type) and a more common and a longer message (Context and Policy type). For the second type, we also considered different possible sizes: 15, 30, and 60 rules. For each of this messages, we sent it 100 times to CRêPE, and measured the time overhead. The results for processing time are summarized in Figure 9. From the figure, we can observe that the time overhead for processing is at most 1,264.46ms (s.d. 1,149.59)— that is, for processing a Context and Policy message with 60 rules.

2) Energy Overhead: To assess energy overhead we used the tool described in [46]. We ran several experiments considering the phone behaviour as described in Section VI-B1 (automatically placing a call and starting few applications every 10 minutes, over a period of 120 minutes). In particular, we repeated the experiment 10 times for each of the following: stock Android, and CRêPE for 15 active rules. Figure 10 shows the resulting average for the battery voltage. The battery starts at 4,150mV in both systems. However, after two hours of usage, the battery has a voltage of some 4,058mV and 3,950mV, for stock Android and CRêPE, respectively. Note that, as reported by the power tutor tool, these values correspond to a residual battery level of 91.2% and 78.4% after 120 minutes of usage, for stock Android and CRêPE, respectively. Observe that this percentage does not directly correspond to the residual mV, since, for example, the minimum amount of mV required for the phone to work is $>>0$.

With regard to the energy consumption for the context detection, we observe that the main impact comes from the technology (e.g. GPS) used for detecting the context. In particular, checking the time variable is not noticeable in terms of decreasing battery level. Also the energy required by the Bluetooth interface is quite small: we observed a contribution to the battery consumption of less than 1%. However, when the context depends on GPS, energy consumption becomes more significant. For example, having the GPS interface always on for one hour, brings the battery level on an average (10 experiments) of 93.4% (s.d. 1.95%), which corresponds to a voltage of 4,053mV (s.d. 22.9mV). We observe that optimization on this point (i.e. mainly for GPS consumption) can be done if a less responsive system is acceptable—e.g. turning on the GPS interface only at fixed time intervals, or considering only other less grained localization technologies in A-GPS [47].

3) Storage Overhead: The most critical components of CRêPE from the storage point of view are the CRêPE policy and context database, *CEP*, and the cache of certificates for authorized third parties. The sizes of the policies considered in the previous experiments (Section VI-B1a) are 4,643, 5,993, 7,335, and 8,773 bytes, for 15, 30, 45, and 60 rules, respectively—i.e. considering the first (oldest) Google Dev Phone (HTC Dream, with 192 MB of RAM and 256 MB of internal flash memory), this represents just 0.0023%, 0.0030%, 0.0036%, and 0.0044% of the RAM, respectively.

CEP, when the policy with 60 rules is loaded, takes 2,146 bytes (0.0011% of RAM), while the corresponding CRêPE database takes 9,216 bytes (0.0034% of flash). We remind that when the policy is in the database it has a different representation than *CEP*. Furthermore, the database contains also the information about the context associated to the policy. The message of the command with the policy considered in the experiment in Section VI-B1c is 3,248 bytes. Finally, our certificates have a size of some 2,700 bytes (0.0010% of flash). We argue that all the storage requirements are very feasible for Android smartphones currently on the market.

VII. Concluding remarks

The lack of the possibility—for users and authorized third parties—to regulate the behaviour of smartphones, based on the context in which they are, makes it difficult to adopt this technology to its full potential. As an example, a user might avoid to install an application if she cannot control its behaviour at any time. Furthermore, the user might want the

phone to change (even automatically) its behaviour accordingly to some contextual situations. In this paper, we propose a solution for these problems: CRêPE (Context-Related Policy Enforcing for Android). This is the first system that enforces fine-grained context-related policies that can be set by both the phone users and the authorized third parties. Also, policies can be set at runtime and remotely, via SMS, MMS, Bluetooth, or QR-code. We have designed and implemented CRêPE. Experimental results support not only the feasibility of the proposal, but also its efficiency against the main issues of mobile devices (energy consumption, responsiveness, and storage). While in some scenarios CRêPE could be managed remotely by experts (e.g. the IT Department of a company that gives phones to its employees), in other cases regular users have to do it. As future work we plan to: (i) study to which extent users would be willing and able to manage CRêPE; (ii) pre-define general purpose policies that the user might just activate/deactivate based on her needs.

Acknowledgments

This work is partly supported by the project S-MOBILE, funded by STW-Sentinels, NL. We would like to thank Vu Thien Nga Nguyen for her contribution in implementing the first prototype of CRêPE.

References

[1] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. CCS '09*, 2009, pp. 235–245.

[2] Android Project. Internet Webpage. [Online]. Available: http://www.android.com

[3] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Security and Privacy*, vol. 7, no. 1, pp. 50–57, 2009.

[4] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *Proc. POLICY '01*, 2001, pp. 18–38.

[5] J. Park and R. Sandhu, "The UCONABC usage control model," *ACM Trans. Inf. Syst. Secur.*, vol. 7, pp. 128–174, February 2004.

[6] CRePEdroid Project. Internet Webpage. [Online]. Available: http://www.crepedroid.org/crepedroid.html

[7] M. Conti, V. T. N. Nguyen, and B. Crispo, "CRePE: context-related policy enforcement for Android," in *Proc. ISC'10*, 2010, pp. 331–345.

[8] K. Twidle, E. Lupu, N. Dulay, and M. Sloman, "Ponder2 - a policy environment for autonomous pervasive systems," in *Proc. POLICY '08*, 2008, pp. 245–246.

[9] Openmoko Project. Internet Webpage. [Online]. Available: http://www.openmoko.com

[10] OMTP: Mobile terminal platform. Internet Webpage. [Online]. Available: http://www.omtp.org

[11] H. Dwivedi, *Mobile Application Security*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2010.

[12] Application Certification Requirements for Windows Phone. Manual. [Online]. Available: http://msdn.microsoft.com/en-us/library/hh184843(v=VS.92).aspx

[13] Windows Phone 7 Security Model. Windows Phone 7 Security Model_FINAL_122010.pdf. [Online]. Available: http://www.microsoft.com/

[14] Nokia Forum. Signed MIDlet Developer's Guide. MIDP_2_0_Signed_MIDlet_Developers_Guide_v2_0_en.pdf. [Online]. Available: http://www.forum.nokia.com/

[15] Symbian Ltd. Simbian Signed. Internet Webpage. [Online]. Available: https://www.symbiansigned.com

[16] I. Ion, B. Dragovic, and B. Crispo, "Extending the java virtual machine to enforce fine-grained security policies in mobile devices," in *Proc. ACSAC '07*, 2007, pp. 233–242.

[17] M. Ongtang, S. McLaughlin, W. Enck, , and P. McDaniel, "Semantically rich application-centric security in android," in *Proc. ACSAC '09*, 2009, pp. 73–82.

[18] G. Bai, L. Gu, T. Feng, Y. Guo, and X. Chen, "Context-aware usage control for android," in *Proc. SecureComm 2010*, 2010, pp. 326–343.

[19] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proc. ASIACCS '10*, 2010, pp. 328–332.

[20] A. R. Beresford, A. Rice, and N. Skehin, "MockDroid: trading privacy for application functionality on smartphones," in *Proc. HotMobile '11*, 2011, to be published.

[21] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh, "Taming Information-Stealing Smartphone Applications (on Android)," in *Proc. TRUST 2011*, 2011, to be published.

[22] G. Sampemane, P. Naldurg, and R. H. Campbell, "Access control for active spaces," in *Proc. ACSAC '02*, 2002, pp. 343–352.

[23] Andromaly—anomaly detaction in android platform. Internet Webpage. [Online]. Available: http://andromaly.wordpress.com

[24] W. Han, J. Zhang, and X. Yao, "Context-sensitive access control model and implementation," in *Proc. CIT '05*, 2005, pp. 757–763.

[25] M. C. Matthew, . Matthew, J. Moyer, and M. Ahamad, "Generalized role-based access control for securing future applications," 2000.

[26] M. L. Damiani, E. Bertino, B. Catania, and P. Perlasca, "GEO-RBAC: A spatially aware RBAC," *ACM Trans. Inf. Syst. Secur.*, vol. 10, no. 1, pp. 1–42, 2007.

[27] F. Hansen and V. Oleshchuk, "Srbac: A spatial role-based access control model for mobile systems," in *Proc. NORDSEC'03*, 2003, pp. 129–141.

[28] J. Chin, N. Zhang, A. Nenadic, and O. Bamasak, "A context-constrained authorisation (cocoa) framework for pervasive grid computing," *Wirel. Netw.*, vol. 16, pp. 1541–1556, August 2010.

[29] N. N. Diep, L. X. Hung, Y. Zhung, S. Lee, Y.-K. Lee, and H. Lee, "Enforcing access control using risk assessment," in *Proc. ECUMN '07*, 2007, pp. 419–424.

[30] A. Ahmed and N. Zhang, "A context-risk-aware access control model for ubiquitous environments," in *Proc. IMCSIT '08*, 2008, pp. 775–782.

[31] A. Corradi, R. Montanari, and D. Tibaldi, "Context-based access control management in ubiquitous environments," in *Proc. NCA '04*, 2004, pp. 253–260.

[32] N. Gohring. (2011, February) VMWare Shows off Mobile Virtualization on Android. Internet Article. [Online]. Available: http://www.pcworld.com/article/219671

[33] E. Yuan and J. Tong, "Attributed based access control (abac) for web services," in *Proceedings of the IEEE International Conference on Web Services*, ser. ICWS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 561–569. [Online]. Available: http://dx.doi.org/10.1109/ICWS.2005.25

[34] R. Bhatti, E. Bertino, and A. Ghafoor, "A trust-based context-aware access control model for web-services," *Distributed and Parallel Databases*, 2005.

[35] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google android: A comprehensive security assessment," *IEEE Security and Privacy*, vol. 8, pp. 35–44, 2010.

[36] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Commun. ACM*, vol. 19, pp. 461–471, August 1976.

[37] T. Moses, *eXtensible Access Control Markup Language (XACML) Version 2.0*. OASIS Standard, 2005.

[38] P. Mazzoleni, B. Crispo, S. Sivasubramanian, and E. Bertino, "XACML policy integration algorithms," *ACM Trans. Inf. Syst. Secur.*, vol. 11, no. 1, pp. 1–29, Feb. 2008.

[39] Z. Liu, A. Ranganathan, and A. Riabov, "Specifying and enforcing high-level semantic obligation policies," *Web Semant.*, vol. 7, pp. 28–39, 2009.

[40] M. Conti, I. Zachia-Zlatea, and B. Crispo, "Mind how you answer me! (transparently authenticating the user of a smartphone when answering or placing a call)." in *Proc. ASIACCS '11*, 2011, pp. 249–259.

[41] B. van Wissen, N. Palmer, R. Kemp, T. Kielmann, and H. Bal, "Contextdroid: an expression-based context framework for android," in *Proc. PhoneSense '10*, 2010, pp. 1–5.

[42] Android Open Source Project (AOSP). Internet Webpage. [Online]. Available: http://source.android.com/

[43] ANother Tool for Language Recognition (ANTLR). Internet Webpage. [Online]. Available: http://www.antlr.org/

[44] ARM Trustzone Technology. Internet Webpage. [Online]. Available: http://www.arm.com/products/processors/technologies/trustzone.php

[45] D. Wallach. (2011, February) Things overheard on the Wi-Fi from my Android smartphone. Internet Article. [Online]. Available: http://www.freedom-to-tinker.com/blog/dwallach/things-overheard-wifi-my-android-smartphone

[46] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. CODES/ISSS '10*, 2010, pp. 105–114.

[47] G. M. Djuknic and R. E. Richton, "Geolocation and assisted gps," *Computer*, vol. 34, no. 2, pp. 123–125, 2001.