A Policy Enforcement Framework for Android

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Master of Technology

by

Keskar Kaustubh Shriniwas
Roll No: 10305909

under the guidance of

Prof. D. B. Phatak

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai
Abstract

Android is an open-source Operating System and its architecture allows developers to design apps for Android without needing to know its internals. Wide variety of Android apps are available to the user for personalizing and customizing the phone. Android provides coarse-grained access control to the user when it comes to security. Researchers have proposed many policy enforcement (security) frameworks to enhance the Android’s default security mechanism. These frameworks allow the users to define security policies for malware mitigation, resource usage control or privilege escalation attack prevention. In our case, framework should monitor apps which are being launched. If the app does not adhere to the policies defined by user, it should not be allowed to launch or if it is already launched, it should be killed by the framework. At first we study how these existing frameworks are not suitable for the use-cases presented in this report, and then we propose a new policy enforcement framework to meet our requirements.
Contents

1 Introduction 1
  1.1 Problem Statement ........................................... 1
  1.2 Motivation .................................................. 2

2 Literature Survey 4
  2.1 Basics of Android System ..................................... 4
    2.1.1 Android Architecture ................................ 4
    2.1.2 Android Security .................................... 5
  2.2 Policy Enforcement Frameworks ............................... 6
    2.2.1 Kirin .................................................. 6
    2.2.2 Saint ................................................ 8
    2.2.3 Apex ............................................... 9
    2.2.4 CRePE .............................................. 11
    2.2.5 XManDroid ........................................... 12
    2.2.6 TrustDroid ......................................... 13
    2.2.7 ConUCON .......................................... 14
    2.2.8 YAASE ............................................. 16
    2.2.9 SEAF .............................................. 17
    2.2.10 Comparison .................................... 18

3 Polork - Implementation Details 22
  3.1 System Architecture ......................................... 22
  3.2 Policy Attributes .......................................... 23
  3.3 Commands .................................................. 24
  3.4 Policy Enforcement - Algorithm ........................... 26
  3.5 Comparison ................................................ 29
  3.6 Context Attributes ........................................ 29
    3.6.1 Time ............................................... 29
    3.6.2 Location .......................................... 30

4 Results 31
  4.1 Test Case 1 ............................................... 31
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Kirin [16]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Saint [18]</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>CRePE [13]</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Various Commands</td>
<td>25</td>
</tr>
</tbody>
</table>
List of Tables

2.1 ConUCON Policy Example ................................................. 15
2.2 Comparison of Varioud Policy Frameworks ............................... 19
2.3 Comparison of Varioud Policy Frameworks ............................... 20
2.4 Comparison of Varioud Policy Frameworks ............................... 21

3.1 Policy Tags ........................................................................ 24
3.2 Comparison of Possible Implementation Approaches .................. 29
3.3 Context Attribute (Time) ..................................................... 30
3.4 Evaluation of Location Attribute ......................................... 30

4.1 Result Set 1 .................................................................... 36
4.2 Result Set 2 .................................................................... 38
4.3 Result Set 3 .................................................................... 38
Chapter 1

Introduction

Android is a Linux-based open-source Operating System by Google, for mobile devices such as smartphones and tablets. Android owns 68% share of global smartphone market by Q2, 2012 after being launched in 2007 [19]. Developers can develop free or paid apps for Android using Android SDK. They can use public APIs, without having any knowledge of Android internals, to make use of device features and services provided by Android. Google has also established Google Play Store which allows developers to distribute their apps to Android users.

Android has employed various mechanisms like application sandboxing, application signing, permission model, etc., to enhance the security in Android. None the less, all these mechanisms have certain limitations. They provide coarse-grained access control to the user, as well as they cannot provide complete solutions for users’ security and privacy concerns. Android protects system features (accessed using APIs) by assigning them permissions. At the time of installation, users can see which permissions are requested by an app, but even even the decision taken by the user is all-or-nothing. This means that all the permissions should be granted to that app to continue the installation. If user decides not to grant these permissions, then the installation is aborted. Once the permission is granted, that app gets unlimited access to the resource. The only way to revoke the permissions is by uninstalling the app. For users, it is not possible to impose the constraints on the usage of the resources. Further, in case of school-students or employer-employees scenario, where Android devices are distributed by school or employer, it is not possible for them to have control over these devices. We focus on extending Android’s security mechanism to allow (third-party) users to have fine-grained access control on their device. Smartphones and tablets running on Android OS are referred to as ‘devices’ throughout this report. Users who own the device (have physical access to the device) are referred to as ‘end-users’.

1.1 Problem Statement

End-users of Android devices can install apps from various sources (Google Play Store, from SD-Card, etc.). Once these apps are installed, Android provides no mechanism to
control which apps can be launched. Also, it is not possible for third-party users to have control over these devices. Further, it is also not possible to have control over launching of apps, based on (system or external) environmental attributes like time, location, etc..

In this project, our aim is to design and implement a context-aware, multi-user policy enforcement framework for Android. The framework should allow or kill the app being launched depending on the policies defined by user. User should be allowed to define fine-grained context-aware policies using this framework. Context-aware policies depend upon various context attributes like time, location, battery usage, etc. Policy conflicts (occurring due to definition of more than one policy for the same package) should also be resolved by the framework. We have focused on implementing the framework in such a way, that trusted third-parties (schools, teachers, employers) can define the policies for the device. Once the policies are defined remotely by third-parties, the framework downloads and enforces those policies.

1.2 Motivation

Policy Enforcement Framework for Aakash

This section considers an idea of a policy enforcement framework for Aakash tablet. Aakash tablet would be getting distributed to thousands of school and college students. Some of the use-cases explaining why Aakash tablet would need a policy enforcement framework are as follows:

- If the tablet is being used for conducting quizzes or exams in schools, only the quiz or exam related apps should get open. Any request to start any of the remaining apps should be blocked.

- During school-time, students can open a limited set of apps. List of allowed apps would be defined by schools (teachers). For example, students should not be able to open social networking apps (like Facebook, Twitter) or gaming apps (like Angry Birds, Temple Run) during school-time (say, 09:00hrs to 15:30hrs).

- Every subject/course may require different set of apps. Teachers should be able to define apps that should be allowed during their lectures. For example, during History or Geography class, apps like YouTube, any video player app, Google Maps, Google Earth are useful, but these apps may not be appropriate for another subject (like Maths). Hence, during that class these apps should be blocked.

- Parents may want to have some control over how the tablet is getting used when the student is at home. Parents may wish to let their children use gaming apps for fixed amount of time during a day. For example, gaming apps like Angry Birds, Temple Run should automatically get closed if they have been used for more than two hours.
Context Attributes

The security frameworks Saint [18], CRePE [13], and ConUCON [9] allows users to define context-aware policies. All these frameworks have implemented policies based on time and location only. Battery is mentioned as one of the context attributes, but no framework allows users to define policies based on battery status or any other attributes.

Let’s consider an example where a user watches videos using MX Player app. He/She wishes to watch videos till MX Player app consumes 30% of the total battery. None of the existing policy enforcement frameworks allow user to define such kind of policies. Context-aware policies should be able to include rules based on relative battery consumption of applications. Some other context attributes are also explored.
Chapter 2

Literature Survey

This chapter talks about basic components in any Android Application and about Android security framework. It also discusses various existing policy enforcement frameworks.

2.1 Basics of Android System

2.1.1 Android Architecture

Any Android application has four components [14]:

- **Activity:** An Activity is a screen or window (UI) which is displayed to the user for interaction purpose. Only one activity has a focus at a time. An application can have many activities that can be used to implement different functionalities (UIs). One activity may start another activity and wait for that activity to finish to get back the result.

- **Service:** Service is equivalent to a background process which is invisible to the user. Other components can bind to the service and get the result back, using Remote Procedure Call (RPC) mechanism.

- **Content Provider:** Content Providers can perform SQL-like operations, e.g. SELECT, UPDATE, DELETE, INSERT on the underlying SQLite databases. Content Provider can define two permission labels to protect, read, and write access separately.

- **Broadcast Receiver:** A Broadcast Receiver component is an asynchronous event mailbox for Intent messages ‘broadcasted’ to an action string [16]. Application developers can add broadcast receiver in their applications, to receive events occurring at the system-level. Applications can also create broadcast receivers at runtime (that do not appear in manifest file), called as dynamic broadcast receiver.
Inter-Component Communication (ICC)

An intent is an abstract description of an operation to be performed. An intent provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities. It is basically a passive data structure holding an abstract description of an action to be performed [21].

Intents can be used by components to interact with other components (of same or different application). If the source components knows how to access target component (application name, component name), then used intent is called as explicit intent. If component broadcasts the intent (specifying action string) to the Android framework, then those intents are called as implicit intents. ICC functions identically regardless of whether the target is in the same or different application [14].

2.1.2 Android Security

Application Sandboxing

Every application gets a unique user identifier (UID) at the time of installation. Linux kernel is responsible for enforcing access control on the resources of the system. Hence, one application cannot access other application’s files unless they are marked as worldwide readable. Also, every application is run into separate VMs. It implies that the vulnerability found in one application will not affect remaining applications.

Permission Model

A reference monitor is present as a part of permission model in Android framework layer. Reference monitor enforces mandatory access control (MAC) on ICC calls. Android protects sensitive APIs by assigning them permissions. A permission label is simply a unique text string that can be defined by both the OS and third party developers [18]. Application developers can restrict access to application components by defining their own permission label. If any other application needs to access that protected component, then it must use (declare) same permission label to gain access. Permission labels are divided into four ‘protection levels’, description of which can be found in section 3 of Saint[18].

Application developer needs to define (list) all the permissions required to access protected APIs in application’s manifest file AndroidManifest.xml. Even though permissions are means of protecting components and sensitive APIs, once the permission is granted to the application, there is no way to revoke the permission other than uninstalling the application. Also, assignment of a permission to an application results in providing unrestricted access to the respective resources [17].

Application Signing

Android requires every application to be signed, not necessarily from third party certificate authority. Application developers can sign their own applications. This allows Android to build trust relationships among applications from the same developer. If two
applications are signed using same certificate and they ask for same shared UID, then either of the application can use permissions granted to other application.

2.2 Policy Enforcement Frameworks

Policy enforcement frameworks provide fine-grained access control to the devices. Security policies, when enforced, provide enhanced security to the user. This chapter summarizes various existing policy enforcement (or security) frameworks. Last part of the chapter compares all these frameworks.

2.2.1 Kirin

Kirin [16] security framework performs lightweight certification of applications at install-time with the aim of mitigating malware. It compares the security configuration present in applications against predefined security rules to prevent malware. If one of the configuration matches, then that application is flagged as threat to the system. It is left up to the user whether or not to continue the installation.

Kirin framework defines all permission labels, broadcast Intent messages sent by the system and all components of applications as assets. Functional requirements consider the interaction between the assets and rest of the phone and third-party applications. Security rules describe threats to the system and are considered as abuse cases which violate the security goals. These rules state who can(not) perform certain functionality or conditions under which functionality may (not) occur. To define security rules, sets of functionalities that may compromise a threat are identified. These rules are matched against the security configuration present in application at the installation time. Every application has limited security information available in AndroidManifest.xml (Android package manifest file). Using permission labels and action strings declared in this manifest file, security configuration is constructed. As Kirin only considers permission labels and action strings, end-user can define security rules, which are various combinations of these two attributes only.

Kirin framework has modified Android Application Installer to place a hook inside it. As shown in figure, (1) Whenever user tries to install any application, (2) Application Installer will call Kirin Security Service. Kirin Security Service extracts package manifest file and reads security rules from the database. (3) Kirin Security Service checks for

![Figure 2.1: Kirin [16]](image-url)
malware threat and sends the reply back to Android Application Installer. (4) A Kirin based rating system would be helpful to the user to make installation decision.

To decide whether the configuration extracted from Android package file matches a given rule-set, the following logic is used: Let $P$ be the set of all permission labels and $A$ be the set of all possible action strings used by Activities, Services and Broadcast Receiver to receive Intents. Let $\mathcal{R}$ be the set of all rules that can be expressed using Kirin Security Language (KSL), i.e. $\mathcal{R}$ is $(2^P, 2^A)$, the power set of sets $P$ and $A$. $P_i$ is the union of sets of “permission” restrictions and $A_i$ is the union of sets of “receive” restrictions. The specific rule $r_i$ can be expressed as $(P_i, A_i)$, where $r_i \in \mathcal{R}$, $P_i \subseteq 2^P$ and $A_i \subseteq 2^A$.

Let $C$ be the set of all the possible configurations extracted from a package manifest file. $c_t \in C$ is used to denote a specific subset of permission labels and action strings used by an application $t$, where $c_t = (P_t, A_t)$, $P_t \in 2^P$ and $A_t \in 2^A$.

Function $\text{fail} : C \times R \rightarrow \{\text{true, false}\}$ is used to test if an application configuration fails a KSL rule. Let $c_t$ be the configuration for target application $t$ and $r_i$ be a rule. Then, $\text{fail}(c_t; r_i)$ is defined as:

$$(P_t, A_t) = c_t, \ (P_i, A_i) = r_i, \ P_i \subseteq P_t \land A_i \subseteq A_t$$

Let $F_R : C \rightarrow R$ be a function returning the set of all rules in $R \in 2^\mathcal{R}$ for which an application configuration fails:

$$F_R(c_t) = \{r_i | r_i \in R, \text{fail}(c_t, r_i)\}$$

Then, the configuration $c_t$ passes a given KSL rule-set $R$ if $F_R(c_t) = \phi$. The set $F_R(c_t)$ can be returned to the application installer to indicate which rules failed.

KSL syntax in BNF notation is given as:

\[
\langle \text{rule - set} \rangle ::= \langle \text{rule} \rangle | \langle \text{rule} \rangle \langle \text{rule - set} \rangle \\
\langle \text{rule} \rangle ::= \text{"restrict" } \langle \text{restrict - list} \rangle \\
\langle \text{restrict - list} \rangle ::= \langle \text{restrict} \rangle | \langle \text{restrict} \rangle \text{"and" } \langle \text{restrict - list} \rangle \\
\langle \text{restrict} \rangle ::= \text{"permission"} [\langle \text{const - list} \rangle ] | \text{"receive"} [\langle \text{const - list} \rangle ] \\
\langle \text{const - list} \rangle ::= \langle \text{const} \rangle | \langle \text{const} \rangle \text{",” } \langle \text{const - list} \rangle \\
\langle \text{const} \rangle ::= \text{""} [A - Z a - z 0 - 9.] + \text{“”}
\]

Every rule begins with keyword “restrict”, followed by conjunction of sets of “permission” and “receive”. Sample Kirin security rules can be defined as:

* restrict permission PHONE_STATE and permission RECORD_AUDIO and permission INTERNET (1)
* restrict permission ACCESS_FINE_LOCATION and permission INTERNET and permission RECEIVE_BOOT_COMPLETE (2)
* restrict permission SET_PREFERRED_APPLICATION_COMPLETE and receive CALL (3)

Rule (1) says that applications - having PHONE_STATE, RECORD_AUDIO and INTERNET permissions - may record audio when phone receives a call and it may send the recorded audio on the Internet. Rule (2) will detect the applications which are capable of starting on boot, getting GPS location and sending it on the Internet. Applications which can set itself as an alternative to default call application will be detected by rule (3).
2.2.2 Saint

Saint (Secure Application INTeraction) [18] allows application developers to protect their apps from other apps installed in the phone. Saint governs install-time permission assignment and their run-time use. Permission labels defined in Android restrict who can access the component protected by these permissions. But fine-grained access control, such as when, where, under what conditions the access is allowed, is not possible in Android. Saint allows app developer to define security policies to regulate the access control.

Install-time policies are used to protect the permissions defined by applications. Runtime policies restrict communication access between applications. Policies defined by both caller and callee are taken into consideration while enforcing runtime checks. Saint also recognizes context-aware policies, which restrict access based on runtime state (location, time, battery, etc.).

In the above figure, (1.1) and (2.1) are default mechanisms present in Android. Install-time policy enforcement mechanism is responsible for permission assignment and makes the decision whether or not to continue the installation. If app $A$ declares permission $P$, then Saint allows app $A$ to define the conditions under which permission $P$ can be granted to other apps. Signature-based policy (1.2) control how the permission defined by one app is granted to requesting app based on its signature. Configuration-based policies (1.3) consider various application-related parameters (set of requested permissions, version) to control permission assignment.

Runtime policy enforcement mechanism governs communication access between components of different applications. Signature of callee application is checked to control the access in case of signature-based policies (2.2). Desirable configuration for the opponent application can be defined using configuration-based policies (2.3). Context-based policies (2.4) allow applications to control runtime interaction based on contexts (location, time, battery, network state, etc.). Runtime policies have two types: access and expose. Caller’s security requirements are expressed using access policies and applications specifying access policy are considered as the target (callee) of the ICC. On the other hand,
expose policies specify callee’s security requirements and application specifying expose policies are considered as the source (caller) of the ICC. The ICC is allowed if all policies defined by both the caller and the callee are satisfied.

Syntax for declaring install-time and runtime policies is given as:

\[
\begin{align*}
\text{(permission — label) (owner) [!]cond_1 [!]cond_2] ... [!]cond_n} & \quad (1) \\
\text{(expose|access) (sourceapp, type, action) (destinationapp, component)} & \quad [!]cond_1 [!]cond_2] ... [!]cond_n & \quad (2)
\end{align*}
\]

Syntax (1) is used for specifying install-time policies while syntax (2) is used for runtime policies. For runtime policies source and destination can be an app, a component, an Intent, or an app/Intent combination. Examples of install-time and runtime policies:

\[(\text{com.xyz.abc}) (\text{com.xyz.loc}) \text{required-permission(ACCESS_FINE_LOCATION)} \]
\[(\text{access}) (\text{com.test.pqr, any, GET_LOC}) (\text{any, any}) \text{forbid-permissions(INTERNET)} \]

In first example, permission \text{com.xyz.abc} declared by app \text{com.xyz.loc} only be granted to applications already having \text{ACCESS_FINE_LOCATION} permission. In second example, \text{com.test.pqr} cannot start any interaction with action \text{GET_LOC} to any component in any application with permission \text{INTERNET}.

### 2.2.3 Apex

Apex (Android Permission EXtension) [17] is a policy enforcement framework which allows end-users to selectively grant/deny permissions to the applications and restrict the usage of the resources at runtime.

Android system allows user to grant the permissions to the application at the time of installation. Once these permissions is granted, the application gets unrestricted access to that resource. Android provides no means to monitor the usage of that resource. Apex framework allows user to define runtime constraints on the usage of resources.

To keep track of usage of resources, Apex maintains application state for every application. The state stores various attributes and their values. Whenever the values of these attributes are updated, we say that the application state is changed. These attributes are used as part of policies to impose runtime constraints on the resources.

Formal definitions of some of the terms (from section 3 of Apex [17]) are given here:

- **Application State.** An application state is a function \( \tau : \eta(A) \to \text{dom}(\eta(A)) \), where \( A \) is the set of applications, \( \eta \) is the function that maps an application to a set of attribute names, and \( \text{dom}(x) \) is the value domain of attribute set \( x \) of an application \( a \in A \).

- **Policy.** A policy defines the conditions under which an application is granted a permission. It consists of two input parameters - an application and a permission - on which it is applicable, an authorization rule composed of predicates that specify the conditions under which the permission is granted/denied and a set of attribute
update actions, which are to be performed if the conditions in the authorization rule are satisfied. Specifically:

\[ l(a, p) : q_1 \land q_2 \land q_3 \land ... \land q_n \rightarrow \{ \text{permit, deny} \} \]

\[ u_1; u_2; u_3; ...; u_n \]

where \( a \in A, p \in P, q_i \in Q, u_i \)'s are attribute update actions, \( l \in \Lambda \) and \( \Lambda \) is the set of policies in the system. The right-hand-side of the authorization rule defines the value returned by the policy. \( A, P \) and \( Q \) are set of applications, permissions and predicates respectively.

For an app \( a \) having permission \( p \), policy can be defined such that if all predicates \( q_1, q_2, ..., q_n \) are satisfied, then the request will either be permitted or denied. Irrespective of result of predicate, update actions are always performed.

**Dynamic Permission Function.** The dynamic permission function specifies the conditions under which a component \( c_1 \) is granted permission to call another component \( c_2 \) using intent \( i \). It incorporates the static checks as well as the dynamic runtime constraints in its evaluation. For a permission to be granted, Android’s permission checks must grant the permission and there must not be a policy that denies the permission. Formally:

\[ \rho(c_1, c_2, i) \iff A_p(c_2) = \text{null} \lor \exists p \in P, a \in A. a = \varsigma(c_1) \]

\[ \land p = A_p(c_2) \land p \in \mu(a) \]

\[ \land i \in A_f(c_2) \]

\[ \land \neg \exists l \in \Lambda. l(a, p) = \text{deny} \]

\( c_1 \) is a component of app \( a \) which is trying to call component \( c_2 \) of another app using intent \( i \). \( c_1 \) can access \( c_2 \), if \( c_2 \) does not define any permission to restrict any access to itself. If \( c_2 \) has defined a permission \( p \) to restrict the access, then access will be granted to app \( a \) if it possesses same permission \( p \), provided there is no policy \( l \) which denies the access to app \( a \).

In the example given below, first two policies are used to keep track of total number of SMS sent by the app “com.test.SmsApp”. If the number of SMS sent in a day exceeds 10, then permission to send SMS is denied. Third policy always denies INTERNET permission to the same app.

```
sms_count_allow ("com.test.SmsApp" as SMSApp,
    "android.permission.SEND_SMS" as SMS):
    SMSApp.sentSms \leq 10 \land SMSApp.lastUsedDay = System.CurrentDay
    -> permit(SMSApp, SMS);  
    SMSApp.sentSms = SMSApp.sentSms + 1;
```
sms_count_deny ("com.test.SmsApp" as SMSApp, "android.permission.SEND_SMS" as SMS):
SMSApp.sentSms > 10 ^ SMSApp.lastUsedDay = System.CurrentDay -> deny(SMSApp, SMS);

restrict_internet ("com.test.SmsApp" as SMSApp, "android.permission.INTERNET" as Net):
true -> deny(SMSApp, Net);

### 2.2.4 CRePE

CRePE [13] stands for “Context-Related Policy Enforcement”. The system allows end-user and trusted third parties to define fine-grained context-related security policies. The main aim of CRePE is to provide extended control on smartphones so that users and trusted third parties deal with their security and privacy concerns.

CRePE defines “security policy” as a statement that partitions the states of the system into a set of authorized (secure) states and a set of unauthorized (insecure) states. A “context-related policy” is a security policy whose enforcing requires the awareness of the context of the phone. CRePE considers different variables such as time, location, the presence of other devices to determine the context. These contexts can be automatically detected and activated by CRePE. The system also allows trusted third parties to define and activate security policies at run time.

![Diagram of CRePE](image)

**Figure 2.3: CRePE [13]**

As shown in the figure, any call made by an applicable (A.1) is first checked by CRePE and if granted, then passed on to Android Permission Check (A.3). CRePE store all the contexts and policies in PolicyProvider. CRePE PermissionChecker calls PolicyManager (A.2) to check if there is any rule present which denies the access. Users can manually activate the context by sending notification (C.1). ContextInteractor also maintains all
the contexts’ information to inform PolicyProvider when a context becomes (in)active (C.2). PolicyManager will evaluate policies and send the reply to ContextInteractor (C.3) and to ActionPerformer (M.2) so that it can perform actions related to activated policies.

CRePE allows users to define contexts based on various attributes like time and location. Every context has only one policy associated with it, i.e. context and policy have one-to-one relation. A policy can have one or more rules defined in it, i.e. policy and rules have one-to-many relation. A rule $R$ is defined as $R = (\langle r \rangle, \langle access \rangle)$ where $r$ is a resource (applications, system services) and $\langle access \rangle := allowed|denied|any$. For example, rule $R = (CALL, deny)$ denies the call to be made.

At a time, more than one context can be active, so can be the policy associated with that context. But if two active policies have conflicting rules related to the same resource, only one rule can be active based on conflict resolution. As mentioned above, CRePE can be used by more than one user (end-user as well as trusted third parties). CRePE assigns priority level to each of its user. If the rules are conflicting, then the rule defined by the user who has higher priority level will win. If the rules are defined by the user as same priority level, then more restrictive rule (deny over allow) will win.

It should be noted that context/rule defined by one user cannot be deleted or modified or deactivated by another user. For trusted third parties, CRePE provides feature that allows them to add, update, delete and activate the contexts remotely. User can send the notifications over the air using WiFi, Bluetooth or SMS.

2.2.5 XManDroid

The goal of “Extended Monitoring on Android” (XManDroid) [11] security framework is to detect and prevent application-level privilege escalation attacks at runtime. It monitors the ICC between applications and validates them against fine-grained security policies. It creates a system-level graph to dynamically analyze how the permission usage is transmitted between applications.

If the malicious application makes use of unprotected interface of benign privileged application, then the attack is called as confused deputy attack. If both applications are malicious and if they use each other’s permissions to perform unauthorized operation, then both of them are called as colluding applications. XManDroid is able to stop both of the attacks.

A system state of all the applications installed in the device and their established communication links is maintained by XManDroid. Any ICC request is first checked by Android reference monitor. Once the Android reference monitor allows the ICC, XManDroid is invoked. XManDroid checks whether the requested ICC call can constitute an attack. If any rule is matched, then ICC request is blocked. To improve the decision response time, XManDroid stores previously made decisions in a cache. Whenever a decision has to be made, it first checks the cache. If the decision is present in the cache, then same result is applied to requested ICC. If the decision is not present, then all corresponding rules are checked and result is stored in a cache.

XManDroid is able to block direct ICC, broadcast Intents and Pending Intents. Direct
ICC is blocked based on caller and callee applications. For broadcast Intents, if the receiver app is violating the policy, then that app is removed from list of apps to which the Intent is to be sent. Pending Intents are handled same as direct ICC.

Some of the examples of XManDroid to prevent privilege escalation attack:

An application that has ACCESS_FINE_LOCATION permission must not communicate to an application that has INTERNET permission.  
(1)

An application that has READ_SMS permission must not communicate to an application that has INTERNET permission.  
(2)

The first policy prevents leaking of location information through an application that can communicate over network. The second policy prevents leaking of SMS data through an applicable that communicate over network.

XManDroid also prevents data leakage through System Content Providers and System Services. Both components store extra field ‘WriterUID’ which is used to store who has modified the data. If there exists a policy which prevents data links between writer app and target app, then that information will be filtered out from the response.

2.2.6 TrustDroid

TrustDroid [12] security framework allows grouping of data and applications into isolated domains, where data and applications belonging to one trust level are considered in the same domain. Such isolation into domains prevents inter-domain application communication and data access. TrustDroid framework helps the enterprise to keep corporate applications isolated from private user applications to maintain the integrity and confidentiality of the corporate assets.

TrustDroid framework is designed for employer-device-employee scenario. When an employer issues a device to its employee, the enterprise has to make sure that any sensitive corporate assets are not leaked. To prevent the leakage, the applications are colored based on the certificates issued by the enterprise. Applications and data having same color are considered into same logical domain.

TrustDroid divides applications into three trust levels (domains): a) system apps that are pre-installed on device b) trusted third party apps published by enterprise c) untrusted third party apps downloaded by end-user from public sources. Any kind of access between apps from trusted and untrusted domain is never permitted and any kind of access to system apps is never denied.

TrustDroid uses certification based approach to determine the color of an application. The enterprise may optionally issue the certificate. This certificate is verified by a trusted service running on device, at the time of application installation. If the certificate is present and verified successfully, then the installation is allowed. If the certificate cannot be verified, then the installation is aborted. If the certificate is not present, then the installation is allowed and the application is listed as untrusted.
• **Direct ICC.** Communication between applications having different colors is always blocked. If the sender or receiver of ICC is system application, then the access is allowed. The certificate issuer should take care that certificates issued to different apps should have same color, if the communication between those two apps is to be allowed. TrustDroid uses XManDroid [11] framework for policy enforcement on ICC.

• **Broadcast Intents.** Intents broadcast by an app should only be received by set of apps belonging to same domain. System apps are not removed from the filtered list of receivers.

• **System Content Providers.** An extra field is added to all system content providers to keep track of which app has updated (insert, modify) the database. When the read request comes, only the data belonging to the same domain is sent as response.

• **System Service Providers.** Data values of the System Service Providers are marked with the color of the last app updating its value. Request arrived from another app is denied, if its color is different than color assigned to Service Provider.

TrustDroid can also prevent the privilege escalation attacks to some extent. If a vulnerable app (which can leak data because of unprotected component) is in one domain and another app trying to get the data is in different domain, then TrustDroid will block the communication request between these two apps. But if the vulnerability is found in a system app, it is not possible to prevent the attack, as TrustDroid allows all the requests if the sender or receiver of ICC is a system app.

### 2.2.7 ConUCON

“Context-Aware Usage Control” (ConUCON) [9] framework enables user to protect her data and impose constraints on resource usage. ConUCON allows users to define fine-grained context-related security policies. User can selectively grant or revoke the permissions assigned to applications using these fine-grained policies. To make the policies context-aware, the security framework keeps track of certain environmental and system attributes such as time, location, battery to activate different contexts as values of these attributes changes. One of the key features of ConUCON is that it performs continuous usage monitoring, as against other models which validate the constraints only when the request for accessing the resource is made. ConUCON keeps on monitoring the policy once it is activated until it becomes false.

ConUCON has following components: subjects, objects, states (which include subject attributes and object attributes), rights, permissions, obligations, and contexts.

A subject is an entity that can perform certain operations on objects. An object is an entity that is used by subject for performing certain actions. A state is collection of subjects, objects, their attributes and values. A right is an action or operation that can
be performed by a subject on an object. A permission is a credential that allows subject
to perform certain right on objects. Both subjects and objects can have permissions
assigned to them. An object defines permission so that access to it can be restricted. A
subject requires permission to access restricted object. Obligations are actions that must
be performed compulsorily before or during the access. Context is determined by various
environmental and system attributes. Location, time, battery, CPU rate are some of the
examples of context attributes. Formal definitions of these components can be found in
section 3 of ConUCON [9].

ConUCON considers different attributes like time and location to decide which context
to activate. Time is usually mentioned as duration during which the context should remain
activated. For the location, \(<\text{latitude, longitude}>\) and radius of the area is mentioned.
If the phone is within that area, related context will be activated.

<table>
<thead>
<tr>
<th>Components</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>All</td>
</tr>
<tr>
<td>Object</td>
<td>Camera, Record Audio</td>
</tr>
<tr>
<td>Right</td>
<td>Use</td>
</tr>
<tr>
<td>State</td>
<td>currentTime - lastForbiddenTime &gt; 1minute</td>
</tr>
<tr>
<td>Pre-context</td>
<td>((\neg([01/01/2012 00:00:00, 12/31/2012 23:59:59],[weeks+1, 4.day + 15hours \triangleright 2hours]) \lor \neg\text{meetingroom}))</td>
</tr>
<tr>
<td>Ongoing-context</td>
<td>((\neg([01/01/2012 00:00:00, 12/31/2012 23:59:59],[weeks+1, 4.day + 15hours \triangleright 2hours]) \lor \neg\text{meetingroom}))</td>
</tr>
<tr>
<td>Update</td>
<td>if (forbidden) lastForbiddenTime = currentTime</td>
</tr>
</tbody>
</table>

Table 2.1: ConUCON Policy Example

Say, user has a meeting on every Monday and Thursday from 3pm to 5pm (Table 2.1). She doesn’t want any camera or audio recording to happen during this period. Also whenever she is in a meeting room, recording is not allowed. The policy given in above example helps user to restrict all applications from doing so. \(\text{<Subject:All}>\) tells the security framework to enforce this policy for all applications. \(\text{<Object:Camera, Record Audio}>\) depicts that policy restrict applications from using these functionalities of the phone. If the request for accessing the objects comes within one minute of the last request, then current request will be denied according to the condition mentioned in ‘State’. ‘Pre-context’ and ‘Ongoing-context’ are used to specify time duration and location information. Once the policy is activated, ConUCON continuously checks the condition mentioned in ‘Ongoing-context’.
2.2.8 YAASE

“Yet Another Android Security Extension” (YAASE) model [20] is designed to mitigate malware to prevent the privilege escalation attacks. If one application makes use of unprivileged interfaces of another applications, both of them are called as colluding applications. Such types of attacks can be prevented by YAASE. The main focus of the model is on tracking the data flow through the applications as opposed to other models which mainly handle the inter-component communication requests between applications.

YAASE makes use of TaintDroid framework [15] for data tagging and tracking. Fine-grained policies can be defined by the user to filter out the data. These policies define the data labels that an application is authorised to handle. The data tracking is transparent to the applications and to the user.

Dynamic taint analysis is performed by TaintDroid model to prevent data leakage and privilege escalation attacks. TaintDroid considers four levels of taint propagation: variable-level, method-level, message-level, and file-level. The variable which stores sensitive data is tainted by assigning it a data label. Whenever another variable is evaluated which depends on first variable, we say that the taint is propagated, i.e. same data label is assigned to the second variable. In message-level taint propagation, data label of variables contained in the message is assigned to the message. In file-level tainting, the data label of application performing write operations on the file, is assigned to the file and it is stored along side with the file.

YAASE uses the labeling framework of TaintDroid for the attack prevention. The taint values used to tag sensitive information are stored in labeling store. User can perform certain actions on the tainted data which is specified as part of the policy. The framework employs fine-grained filters to suppress the data that is not relevant to the application making request. This way YAASE controls the data flow through applications.

The syntax for YAASE policy language is given as:

\[
\text{PolicyName: Requester can do operation on Resource} \\
\hspace{1cm} \text{[have to perform action]} \\
\hspace{1cm} \text{handle dataLabelExpression}
\]

Here, PolicyName is the name of policy. Requester is the name of an application that can perform operation on certain Resource. have to perform is an optional clause. If it is defined, an action has to be performed when the Resource is being accessed. As a response to an operation, only the data tagged with dataLabelExpression can be sent to the Requester.

PolicyP1: XYZ can do getSMS on SMS
\hspace{1cm} have to perform filterOut(‘‘Pr’’, returnData);
\hspace{1cm} handle ‘‘Pub’’

User marks some of the SMS as public (‘‘Pub’’ label) and rest as private (‘‘Pr’’ label). App XYZ has permissions to read the SMS contents. But according to above example,
only the SMS marked as public will be sent to the app. It cannot read the SMS marked
with private label.

YAASE can prevent privilege escalation attack. Say app A is a third party app which
has permission to read contacts. App A has unprotected service (no permissions assigned
for accessing the service) which sends all the contact details. There is another third
party app, App B, which can access Internet. In normal scenario, app B can bind to the
unprotected service of app A to retrieve the contact list. Since app B has INTERNET
permission, it can upload the contact list anywhere on the Internet. This way privilege
escalation attack will succeed. But in case of YAASE, the data sent from service of app
A will be tagged with its data label. When app B will try to send the data, data label
of fetched contact list would not match with that of app B. This will filter out all the
contacts and no data will be uploaded on the server.

2.2.9 SEAF

"Security Enhanced Android Framework" (SEAF) [10] helps the user to evaluate the
behavior of an application. It monitors the permission exercising patterns of the applic-
cations to inform the user about any potentially harmful activity. Dangerous behavior
of applications are captured and blocked to mitigate the malware threats to the smart-
phone. The security framework allows user to selectively grant/deny the permissions to
the applications at runtime.

A security policy specify sequential pattern of permissions which can be considered as
harmful. Whenever the application makes use of granted permissions in certain sequential
pattern and if the pattern matches to any of the policy declared in the repository, then
user is notified about the dangerous activity of the application.

The following definitions are taken from section 3 of SEAF [10]:

• Application access resources through a set of permissions. We define function that
  maps an application A to a set of those requested permissions P.

  \[ \beta(A) \to (P) \]

  \[ \text{where } P = \{P_1, P_2, P_3, \ldots, P_n\} \]

• An application runtime behavior graph is a directed acyclic graph (S:P) where S is a
  set of nodes representing states of an application, and P is set of edges representing
different permissions exercised. For example, we write \( P_1 \to P_2 \to P_3 \) to describe
an application that has exercised permission p1, followed by p2 and p3.

• We define \( \delta \) as a function that represents specific sequence of permissions that could
  be either H (Harmful) or B (Benign). Generally \( \delta(P_1 \to P_n) = H \lor B \)

As mentioned in the last point, SEAF stores specific sequence of permissions as harmful
or benign in its repository. Whenever an application is approaching sequential pattern
defined as a policy, user can decide whether or not to allow the execution. She can then
grant or revoke the permissions assigned to that application.
SEAF classifies applications into two modes: *restricted* and *unrestricted*. If the application is trusted by the user, then it will be listed into *restricted* mode and its behavior will not be monitored by SEAF. If the user is unsure about application’s trustworthiness, then that application will be considered as *unrestricted*. SEAF will govern permission exercising pattern for that application and result will be notified to the user.

Consider an application which has *READ_CONTACTS* ($P_1$) and *INTERNET* ($P_2$) permissions. The repository has following two rules defines:

$$\delta(P_1 \rightarrow P_2) = H \quad (1)$$
$$\delta(P_2 \rightarrow P_1) = B \quad (2)$$

The application may read the contacts and upload it on the server using *INTERNET* permission, as mentioned in rule (1). This dangerous behavior will be blocked by SEAF if the application performs similar activity. On the other hand, $P_2 \rightarrow P_1$ is treated as benign.

Drawback of this framework is that it cannot detect suspicious behavior of an application spanned over more than one session. Let’s assume that an application has permissions $P_1$, $P_2$, and $P_3$ and repository has a rule $\delta(P_1 \rightarrow P_2) = H$. If application executes $P_1$ in one session and $P_2$ in next session, then SEAF will not be able to track any potential activity. Another drawback is SEAF cannot detect privilege escalation attacks, because SEAF does not monitor permission execution sequence across applications.

### 2.2.10 Comparison

- In the comparison, ‘I-T’ stands for *install – time* and ‘R-T’ stands for *run – time*.
- The last comparison point ‘Target’ has two values: *General* or *Specific*. *General* means that a single policy can be enforced on more than one application. *Specific* means name of target application is specified as part of a policy, so it can be enforced on that application only.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Aim</td>
<td>To mitigate malware (Protect phone from apps)</td>
<td>To protect apps from other apps</td>
<td>To Extended security mechanism (Security and privacy concerns)</td>
</tr>
<tr>
<td>Framework design summary</td>
<td>Lightweight certification of apps at install-time</td>
<td>Allows developer to define (context-aware) fine-grained policies</td>
<td>- Selectively grant permissions to apps</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Impose runtime constraints on usage of resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- I-T policies decide assignment of permission to requesting app – R-T policies decide access control between caller and callee</td>
<td>By maintaining the application state for every app and comparing policies against attributes in application state</td>
</tr>
<tr>
<td>Framework design details</td>
<td>By comparing security policies against permissions and action strings defined in app’s manifest file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Policy definitions</td>
<td>-</td>
<td>I-T policy definitions</td>
<td>I-T and R-T permission selection, constraints definitions</td>
</tr>
<tr>
<td>Policy enforcement</td>
<td>I-T enforcement</td>
<td>I-T and R-T enforcement</td>
<td>R-T enforcement</td>
</tr>
<tr>
<td>Policies defined by</td>
<td>End-user</td>
<td>App developer</td>
<td>End-user</td>
</tr>
<tr>
<td>(Additional) file reqd.</td>
<td>AndroidManifest.xml</td>
<td>Policy configuration file in apk</td>
<td>-</td>
</tr>
<tr>
<td>Decisions</td>
<td>Whether to allow app installation</td>
<td>Whether to continue app installation; Whether to allow ICC</td>
<td>Whether to allow current call (protected by permission)</td>
</tr>
<tr>
<td>What can be blocked?</td>
<td>Applications from getting installed</td>
<td>Applications from getting installed; All components (call to start component); Broadcast Intents</td>
<td>All components (call to start component)</td>
</tr>
<tr>
<td>Parameters considered for policy</td>
<td>Permission labels; Intents (Action strings)</td>
<td>Permission labels, action strings, app, components, signature, version, etc.</td>
<td>Permission labels; Application/System attributes</td>
</tr>
<tr>
<td>Modifications</td>
<td>Package Installer</td>
<td>Framework + Package Installer</td>
<td>Framework + Package Installer</td>
</tr>
<tr>
<td>Policies defined for</td>
<td>Caller</td>
<td>Caller, Callee</td>
<td>Caller</td>
</tr>
<tr>
<td>Target</td>
<td>General</td>
<td>General, Specific</td>
<td>Specific</td>
</tr>
</tbody>
</table>

Table 2.2: Comparison of Various Policy Frameworks
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Aim</td>
<td>To extend control of user and trusted third parties on smartphone to deal with security and privacy concerns</td>
<td>To detect and prevent application-level privilege escalation attacks at runtime</td>
<td>To keep corporate apps isolated from private user apps to maintain the integrity and confidentiality of the corporate assets</td>
</tr>
<tr>
<td>Framework design summary</td>
<td>- Can define fine-grained context-related security policy - Contexts are activated based on context attributes</td>
<td>Monitors communication links between apps and verifies them against security rules</td>
<td>Domain isolation by preventing inter-domain application communication and data access</td>
</tr>
<tr>
<td>Framework design details</td>
<td>Contexts can be activated manually by user or automatically by monitoring context attributes; Every context has set of rules associated with it</td>
<td>By creating a system level graph of all apps to find path between two communicating apps</td>
<td>By coloring communication and data access between apps; Using XManDroid model to allow/deny access based on colors</td>
</tr>
<tr>
<td>Policy definitions</td>
<td>R-T context/policy/rule definition</td>
<td>I-T policy definition</td>
<td>I-T coloring of apps (no policies)</td>
</tr>
<tr>
<td>Policy enforcement</td>
<td>R-T enforcement</td>
<td>R-T enforcement</td>
<td>R-T enforcement</td>
</tr>
<tr>
<td>Policies defined by</td>
<td>End-user and trusted third parties</td>
<td>End-user</td>
<td>Trusted third parties</td>
</tr>
<tr>
<td>(Additional) file reqd.</td>
<td>–</td>
<td>–</td>
<td>Optional certificate file (issued by the enterprise to decide the color of app)</td>
</tr>
<tr>
<td>Decisions</td>
<td>Whether to allow access to resource (other apps or system service)</td>
<td>Whether to allow communication or data access</td>
<td>Whether to allow communication or data access</td>
</tr>
<tr>
<td>What can be blocked?</td>
<td>Apps and system services</td>
<td>All components, Broadcast Intents</td>
<td>All components, Broadcast Intents</td>
</tr>
<tr>
<td>Parameters considered for policy</td>
<td>Apps and system services, Context Attributes</td>
<td>Permission labels</td>
<td>Colors assigned to apps</td>
</tr>
<tr>
<td>Modifications</td>
<td>Framework</td>
<td>Framework + Package Installer</td>
<td>Framework + Package Installer</td>
</tr>
<tr>
<td>Policies defined for</td>
<td>Callee</td>
<td>Caller</td>
<td>Caller</td>
</tr>
<tr>
<td>Target</td>
<td>Specific</td>
<td>General</td>
<td>General</td>
</tr>
</tbody>
</table>

Table 2.3: Comparison of Various Policy Frameworks
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Aim</td>
<td>Privacy and security concerns (data protection) + resource usage control</td>
<td>To mitigate malware + privilege escalation attack - Privacy and security</td>
<td>To mitigate malware</td>
</tr>
<tr>
<td>Framework design summary</td>
<td>- Can define fine-grained context-aware security policy - Context is activated based on context attributes</td>
<td>Performs data tagging and tracking to control the data flow through applications</td>
<td>Helps to identify potentially dangerous behavior of applications</td>
</tr>
<tr>
<td>Framework design details</td>
<td>Contexts are activated using environmental/system attributes</td>
<td>Policies can be defined to send tainted data with predefined data labels as response; rest of the data is filtered out from the response</td>
<td>Governs dynamic behavior of applications to capture sequential pattern of permissions which could be treated as malware threat</td>
</tr>
<tr>
<td>Policy definitions</td>
<td>R-T context/policy definition</td>
<td>I-T, R-T policy definitions</td>
<td>R-T definitions</td>
</tr>
<tr>
<td>Policy enforcement</td>
<td>R-T enforcement; Continuous monitoring after policy is enforced</td>
<td>R-T enforcement</td>
<td>R-T enforcement</td>
</tr>
<tr>
<td>Policies defined by</td>
<td>End-user</td>
<td>End-user</td>
<td>End-user</td>
</tr>
<tr>
<td>(Additional) file reqd.</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Decisions</td>
<td>Whether to allow access to resources / apps / system services</td>
<td>To filter out data</td>
<td>Whether to allow current call (protected by permission)</td>
</tr>
<tr>
<td>What can be blocked?</td>
<td>All components (call to start component)</td>
<td>Data passed through Intents, Services and CP</td>
<td>Resources (components) protected by permissions</td>
</tr>
<tr>
<td>Parameters considered for policy</td>
<td>Subject-Object-Right, Context Attributes</td>
<td>Tagged data</td>
<td>Permission labels</td>
</tr>
<tr>
<td>Modifications</td>
<td>Framework</td>
<td>Framework + core libraries + Package Installer</td>
<td>Framework</td>
</tr>
<tr>
<td>Policies defined for</td>
<td>Caller</td>
<td>Caller</td>
<td>Caller</td>
</tr>
<tr>
<td>Target</td>
<td>General, Specific</td>
<td>Specific</td>
<td>General</td>
</tr>
</tbody>
</table>

Table 2.4: Comparison of Various Policy Frameworks
Chapter 3
Polork - Implementation Details

This chapter describes architecture of our framework, Polork (name derived from Policy Enforcement Framework). It explains functioning various components of Polork, along with algorithm that we are using while evaluating policies. Note that, user (administrator) who creates policies is referred to as ‘admin’ in this chapter.

3.1 System Architecture

- **XmlParser:** Every policy created by an admin must have specific format, so that it is understood by our framework. Xml schema (xsd) for creating policies is given in appendix I. All the policies created by an admin are stored in files (having extension ‘pf’). This class reads and parses such policy files. For every policy, the parser creates an instance of RawPolicy class.

- **RawPolicy:** This is an intermediate class. The policies stored in files are not directly converted to Policy instance. This class stores the policies and context attributes associated with them. Context attributes are stored as strings.

- **Policy:** This class is used to store policy information. Context attributes are not stored in Policy class. Rather, they are stored separately and Policy class stores a key to retrieve its context attributes. This is because, one than one policy can have same context attributes defined for them. Also, every app can have more than one policy associated with it. Hence, all the policies defined for same app are stored in a single array list (List<Policy> in Java).

- **PolicyContainer:** All the policies (instances of Policy) are stored in PolicyContainer. The class has 2 ‘Map’ (Map<String, List<Policy>> in Java) instances to store the policies. The policies having ‘Command’ as ALLOW_EXCEPT, BLOCK_EXCEPT, ALLOW_ALL, BLOCK_ALL are stored in one of ‘Map’ instances, because such policies can be enforced on any of the applications installed in a device. Policies having ‘Command’ ALLOW_EXCEPT and BLOCK_EXCEPT are stored in one list, whereas policies having ‘Command’ as ALLOW_ALL and
All policies having ‘Command’ as ALLOW or BLOCK are stored in another instance of ‘Map’. The package name, for which the policy is defined, is used as a key for ‘Map’ instance. All the policies defined for a single package are stored in array list (List<Policy>) sorted by <Priority, Sequence Number>. Since, the key is package name, policies for a particular app can be retrieved quickly.

- **ContextAttribs**: As mentioned previously, ContextAttribs can be accessed using a key stored in every Policy instance. One instance of ContextAttribs can have any number of context attributes (time, location, etc.) stored in it. Every attribute is given a name. It also stores ‘context’ (a logical expression represented using names of the attributes, e.g.: (tm01 && loc01) || !tm2). While evaluating the policy, this ‘context’ is evaluated.

- **ConAttrContainer**: This class stores all the context attributes (instances of ContextAttribs) for policies. More than one policy can have same context attribute defined for them. Hence, only one instance of ContextAttribs will be created and a key to retrieve it will be stored in all Policy instances. Similar to PolicyContainer, it uses ‘Map’ (Map<Long, List<ContextAttribs>> in Java) to store context attributes.

- **Polork**: It stores the instances of PolicyContainer and ConAttrContainer. Whenever user tries to launch an app, a function (boolean isAccessAllowed(String pkgName)) defined in this class whether or not to allow the request. This class also has 2 instances of ‘Thread’. Both the ‘Thread’ run after predefined interval. One ‘Thread’ gets all the running processes and then checks if these running apps should be allowed to continue. If not, they are killed. Our framework also allows to disable the apps, as explained in appendix II. Other ‘Thread’ checks whether disabled apps should be enabled, if the policy that was enforced has become invalid.

- **PolorkService**: This class is the entry point in our framework. It holds the instance of Polork. When our framework is started, this class has a method which opens all policies files and pass it to the instance of XmlParser to load all the policies. Android allows apps to register their components with ‘AlarmManager’ using ‘PendingIntent’, so that Android periodically starts registered component. Our framework makes use of these functionality to start itself even after the framework crashes for some reason.

### 3.2 Policy Attributes

We have created a syntax which should be followed while creating policies. Various attributes used in policies are defined in table 3.1. Also, created policies can be validated against XML schema given in appendix I. Some of the examples of policies are given in chapter 4.
<table>
<thead>
<tr>
<th>Tag</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name of the policy</td>
</tr>
<tr>
<td>ValidFrom</td>
<td>Indicates a DateTime from which a policy should be considered as valid. If the current (system) DateTime is before ‘ValidFrom’ DateTime, then that policy is not applicable.</td>
</tr>
<tr>
<td>ValidTill</td>
<td>Indicates a DateTime till which a policy should be considered as valid. If the current (system) DateTime is after ‘ValidTill’ DateTime, then that policy is not applicable.</td>
</tr>
<tr>
<td>Priority</td>
<td>A positive integer assigned to the policy. It is used to sort the policies. The lower the assigned value, higher is the priority.</td>
</tr>
<tr>
<td>SeqNumber</td>
<td>A positive integer assigned to the policy. If ‘Priority’ of two policies is same, then ‘Number’ will be used to sort them (to choose highest priority). A smaller ‘Number’ means higher priority.</td>
</tr>
<tr>
<td>Cmd</td>
<td>Possible values are: ALLOW, ALLOW_EXCEPT, ALLOW_ALL, DENY, DENY_EXCEPT, DENY_ALL. Whether to allow launching of an application depends upon value assigned to ‘Cmd’.</td>
</tr>
<tr>
<td>Context</td>
<td>Contains logical expression (using &amp;&amp;,</td>
</tr>
<tr>
<td>CA-Time</td>
<td>Defined as “&lt;From-time&gt;, &lt;To-time&gt;”. Its value in logical expression is considered as true, if the current (system) time is within this range. A ‘name’ is also required, so that it can be added in ‘Context’ for evaluation.</td>
</tr>
<tr>
<td>CA-Location</td>
<td>Defined as “&lt;Latitude&gt;, &lt;Longitude&gt;, &lt;Range&gt;”. If the current location (received from LocationManager) is within the range of mentioned location, then its value is considered as true. Similar to CA-Time, ‘name’ is required for evaluation. Admin can also specify additional attribute ‘strictlyRequired’. Refer to section 3.6.2 for more details.</td>
</tr>
<tr>
<td>Pkg</td>
<td>Specifies a package name.</td>
</tr>
</tbody>
</table>

Table 3.1: Policy Tags

3.3 Commands

The decision of whether or not to allow starting (launching) of an app depends upon command. The fig. 3.1 lists all the possible commands. In all the cases, outer rectangular
box represents a superset, i.e. a set of all installed apps in the device. The inner circle, if present, represents that the apps P1, P2, P3 and P4 are also installed in the device.

• **ALLOW:** The policy “ALLOW P1, P2, P3, P4” means if user tries to launch any
of these apps, the request to do so is allowed. Policies having ‘ALLOW’ command are applicable for only the apps specified in those policies. They do not have system-wide impact, i.e. if user tries to launch app P5, then policy “ALLOW P1, P2, P3, P4” is not considered. This is the reason why area outside the circle in fig. 3.1 is shown in gray color.

- **DENY**: The policy “DENY P1, P2, P3, P4” means if user tries to launch any of these apps, the request to do so is denied (the app process is killed). Similar to ALLOW, this command is applicable for only the apps specified in the policy. Policy “DENY P1, P2, P3, P4” will not be considered for evaluation if user tries to launch app P5.

- **ALLOW_EXCEPT**: “ALLOW_EXCEPT P1, P2, P3, P4” means if user tries to launch any of these apps, the request to do so is denied, and if she tries to launch apps other than P1, P2, P3 and P4, then those apps will be started normally. They do have system-wide impact, i.e. if user tries to launch app P5, then the request is allowed.

- **DENY_EXCEPT**: “DENY_EXCEPT P1, P2, P3, P4” means if user tries to launch any of these apps, the request to do so is allowed, and if she tries to launch apps other than P1, P2, P3 and P4, then those apps will not be started. They do have system-wide impact, i.e. if user tries to launch app P5, then the request is denied.

- **ALLOW_ALL**: “ALLOW_ALL” means user can launch any app, its system process will not be killed. The command is applicable for all the installed apps, hence, it is not required to specify any apps while using this command.

- **DENY_ALL**: “DENY_ALL” means user cannot launch any app, its system process(es) will be killed by our framework. Similar to “ALLOW_ALL”, it is not required to specify any apps while using this command.

### 3.4 Policy Enforcement - Algorithm

Now that we have described system architecture, various policy attributes and commands, we can explain how our framework decides which policy to enforce when any app is launched by the user.

As mentioned in previous sections, we use ‘Map’ data structure to store the policies. If the policy has command ‘ALLOW’ or ‘DENY’, then they are stored separately in a list (every app has its own list) and package name is used as a key for storing the list in ‘Map’. Policies having remaining commands are stored in a separate ‘Map’ (one list for ALLOW_EXCEPT/DENY_EXCEPT policies and another list for ALLOW_ALL/DENY_ALL policies). All the policies are sorted based on <Priority, SeqNumber> after every insertion.
Along with the policies to be enforced, our framework also maintains a white-list of apps/processes. There are some processes (or apps) which are critical for the well functioning of the system. If these processes are killed, then system may crash (and reboot). Hence, this white-list will have entries for all such apps/processes which should not be killed (no policies will be enforced).

The following pseudocode explains how Polork decides whether to allow launching of an application. When user launches an app having package name ‘pkgName’, isAccessAllowed (pkgName) gets called by the framework.

1: function isAccessAllowed(pkgName)
2:    L ← []
3:    Call getValidPolicyForPkg(pkgName) and insert return value in L
4:    Call getValidSpecialPolicy('EXCEPT') and insert return value in L
5:    Call getValidSpecialPolicy('ALL') and insert value in L
6:    Sort L based on <Priority, SeqNumber>
7:    p ← L[0]
8:    res ← p.checkAccess(pkgName)
9:    return res
10: end function

11: function getValidPolicyForPkg(pkgName)
12:    list := list of all policies for pkgName
13:    for every Policy p in list do
14:       if p.isPolicyValid() = false then
15:          continue
16:       end if
17:       if p has no context attributes then
18:          return p
19:       else if isContextValid() for p = true then
20:          return p
21:       end if
22:    end for
23:    return null
24: end function

25: function getValidSpecialPolicy(str)
26:    if str = ‘ALLOW_EXCEPT’ || str = ‘DENY_EXCEPT’ then
27:       list := list of ALLOW_EXCEPT/DENY_EXCEPT policies
28:    else
29:       list := list of ALLOW_ALL/DENY_ALL policies
30:    end if
31:    for every Policy p in list do
if $p$.isPolicyValid() = false then
    continue
end if
if $p$ has no context attributes then
    return $p$
else if isContextValid() for $p = true$ then
    return $p$
end if
end for
return null
end function

function isPolicyValid
    if currentTime $\geq$ validFrom $\&\&$ currentTIme $\leq$ validTo then
        return true
    else
        return false
    end if
end function

function isContextValid
    for every context attribute $ca$ in context do
        Evaluate and substitute $ca$ in context
    end for
    ret ← evaluate context
end function

function checkAccess($pkgName$)
    if cmd = ‘ALLOW’ $\|$ cmd = ‘ALLOW\_ALL’ then
        return true
    else if cmd = ‘ALLOW\_EXCEPT’ $\&\&$ policy $p$ is not defined for $pkgName$ then
        return true
    else if cmd = ‘DENY\_EXCEPT’ $\&\&$ policy $p$ is defined for $pkgName$ then
        return true
    else
        return false
    end if
end function
3.5 Comparison

There are two ways to implement Polork. We can modify the existing Android source code to incorporate Polork or we can develop it as an Android app. The table 3.2 compares these two approaches. We have chosen to implement it as an Android app.

<table>
<thead>
<tr>
<th>Android Source code</th>
<th>Android App</th>
</tr>
</thead>
<tbody>
<tr>
<td>A request to launch an app can be denied (filtered) by enforcing a policy.</td>
<td>Android SDK contains no APIs which will notify that particular app has been launched. So, we will have to run a Android service at specific to get all running apps and then enforce the policies against them.</td>
</tr>
<tr>
<td>A system process for launched app will not be created, so app cannot be launched at all.</td>
<td>App may run for some interval before being killed.</td>
</tr>
<tr>
<td>If user has root access, she can replace the framework. But it is more difficult to replace the framework as compared to removing app, as it involves setting up of Android source code for the device and compiling it.</td>
<td>If user has root access, she can remove the app.</td>
</tr>
<tr>
<td>As it involves modifications in Android source code, it is easy for device manufacturer to make the changes before flashing the ROM on the device. Also, different versions of Android OS may require different modifications.</td>
<td>App can be installed on any device, provided it has been compiled for that API version of Android. End-user can do it using Android SDK.</td>
</tr>
</tbody>
</table>

Table 3.2: Comparison of Possible Implementation Approaches

3.6 Context Attributes

3.6.1 Time

Our framework allows admin to include time as one of the context attributes while defining the policy. Admin is supposed to assign a name to that attribute and specify a time range (to and from) separated by a comma. If the current time (system clock) of the device is within the specified range, then that attribute will be evaluated to true. If current time does not fall within the range, then it will be evaluated to false.
In current Android-based devices, it is possible for end-user to set the system time. Time can be set using ‘Settings’ app under ‘Date & time’ tab. As user can change the time, policies would be evaluated incorrectly based on set system time.

### 3.6.2 Location

Admin can also enforce the policies based on user’s (device’s) current location. Admin is required to specify latitude, longitude and range (in meters). If the user’s current location is within the range of specified location (Calculate distance between two locations [5]), then the attribute will be evaluated to true. If not, then it will be evaluated to false.

User has to enable use of GPS satellites to pinpoint their location. It can be enabled from ‘Settings’ app under ‘Location services’ tab. Also, continuous use of these services drain the device battery very quickly. So, we have added an extra variable ‘strictlyRequired’ in location attribute, which can be set to either true or false. If the GPS coordinates are not available, then the context attribute will be evaluated based on value specified for ‘strictlyRequired’, as explained in the following table:

<table>
<thead>
<tr>
<th>GPS Coordinated available</th>
<th>strictlyRequired</th>
<th>evaluate</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>evaluate</td>
<td>false</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>evaluate</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 3.4: Evaluation of Location Attribute

In the above table, ‘evaluate’ means specified attribute will be evaluated, as the GPS coordinates are available. Whereas, true and false are the values that will be substituted in place of specified attribute, since the GPS coordinates are not available.
Chapter 4

Results

This chapter describes various scenarios that were created for testing purpose. Here, we have considered 3 test cases, each of them having different policies, to test and validate the working of Polork.

4.1 Test Case 1

This test case is used to show the working of ‘ALLOW’ and ‘DENY’ commands. It is assumed that the apps (Android packages) mentioned in the policies are already installed in the device. Additionally, we have also created a policy ‘polA1’ which blocks all the apps. But, if we set the date-time of system which is not in range of ‘valid-from’ and ‘valid-till’, then this policy should not get enforced. Also, if no policy is found for any application, then the default action (request is allowed) should take place.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<policies>
  <policy>
    <name> polA1 </name>
    <valid-from> 2013-06-25 09:30:00 </valid-from>
    <valid-till> 2013-07-10 18:00:00 </valid-till>
    <priority> 1 </priority>
    <seq-num> 1 </seq-num>
    <cmd> Deny_All </cmd>
  </policy>
  <policy>
    <name> polC3 </name>
    <valid-from> 2013-05-15 14:00:00 </valid-from>
    <valid-till> 2013-06-15 15:30:00 </valid-till>
    <priority> 1 </priority>
    <seq-num> 2 </seq-num>
    <cmd> Allow </cmd>
  </list>
  <pkg> com.bartat.android.elixir </pkg>
```
The results of this case are shown in table 4.1.

The first column App has a name of an app which was being launched. The next two columns Date and Time display the date-time at which the app was launched. Policy columns list all the policies that are considered for enforcement. The column Context shows the context evaluation result for the policy at specified date-time and location. The last column Access is the result of enforcing the policies. The name of policy which is getting enforced is shown in bracket. For some policies, Policy column says ‘NOT CONSIDERED’. It means, that policy is not considered, because it has less priority or sequence number. For some results in Access column, the policy name is mentioned as ‘DEFAULT’, which means that no valid policy is found, and hence our framework allows such requests by default.

As mentioned previously, our framework allows the request, if no valid policy is found. This can be verified from second row (request at 2013-06-20 13:30:00, for app com.bartat.android.elixir). Also, policy ‘polA1’ is not considered for enforcement till 2013-06-25, 09:30:00.
4.2 Test Case 2

In this test case, we will check the functionality of context. If context is not specified, then it is always considered as true. But, if it is specified, then it will be evaluated and its result decide whether the policy should be considered as valid. It also shows use of ‘ALLOW_EXCEPT’ and ‘DENY_EXCEPT’.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<policies>
  <policy>
    <name> polB2 </name>
    <valid-from> 2013-05-15 14:00:00 </valid-from>
    <valid-till> 2013-06-20 15:30:00 </valid-till>
    <priority> 1 </priority>
    <seq-num> 1 </seq-num>
    <cmd> Allow_Except </cmd>
    <context> tm02 </context>
    <ca-time name="tm02"> 15:00:00, 17:45:00 </ca-time>
    <list>
      <pkg> com.bartat.android.elixir </pkg>
      <pkg> com.keramidas.TitaniumBackup </pkg>
      <pkg> com.imangi.templerun </pkg>
    </list>
  </policy>
  <policy>
    <name> polC3 </name>
    <valid-from> 2013-05-15 14:00:00 </valid-from>
    <valid-till> 2013-06-20 15:30:00 </valid-till>
    <priority> 1 </priority>
    <seq-num> 2 </seq-num>
    <cmd> Deny </cmd>
    <list>
      <pkg> org.proxydroid </pkg>
      <pkg> com.mxtech.videoplayer.ad </pkg>
    </list>
  </policy>
  <policy>
    <name> polA1 </name>
    <valid-from> 2013-05-15 09:30:00 </valid-from>
    <valid-till> 2013-07-10 18:00:00 </valid-till>
    <priority> 2 </priority>
    <seq-num> 1 </seq-num>
    <cmd> Deny_Except </cmd>
    <context> tm01 </context>
    <ca-time name="tm01"> 12:00:00, 15:30:00 </ca-time>
  </policy>
</policies>
```
The results of this case are shown in table 4.2.

### 4.3 Test Case 3

This test case shows how different context attributes can be used to create a logical expression for context. We use an extra attribute ‘strictly-reqd’ for location. It specifies how any location attribute should be evaluated if current location (GPS coordinates) is not available. In this case, we have defined one location attribute with ‘strictly-reqd’ as no and the other one set as yes.
<ca-location name="loc01" strictly-reqd="yes"> 19.1305,72.9159, 200
</ca-location>

<list>
    <pkg> com.bartat.android.elixir </pkg>
    <pkg> com.buak.Link2SD </pkg>
    <pkg> com.iitb.proxymity </pkg>
</list>

</policy>
</policies>

The results of this case are shown in table 4.3.
<table>
<thead>
<tr>
<th>App</th>
<th>Request</th>
<th>Policy</th>
<th>Context</th>
<th>Access (Enforced Policy)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Date</td>
<td>Time</td>
<td>Name</td>
<td>Is policy valid?</td>
</tr>
<tr>
<td>com.bartat.android.elixir</td>
<td>2013-06-10</td>
<td>13:30:00</td>
<td>polC3</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>polA1</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>2013-06-20</td>
<td>13:30:00</td>
<td>polC3</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>polA1</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>2013-06-25</td>
<td>13:30:00</td>
<td>polA1</td>
<td>Yes</td>
</tr>
<tr>
<td>org.proxydroid</td>
<td>2013-06-10</td>
<td>13:30:00</td>
<td>polD4</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>polA1</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>polB2</td>
<td>NOT CONSIDERED</td>
</tr>
<tr>
<td></td>
<td>2013-06-20</td>
<td>13:30:00</td>
<td>polD4</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>polA1</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>polB2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>2013-06-25</td>
<td>13:30:00</td>
<td>polD4</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>polB2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>polA1</td>
<td>Yes</td>
</tr>
<tr>
<td>com.kaustubh.testapp</td>
<td>2013-06-10</td>
<td>13:30:00</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>2013-06-20</td>
<td>13:30:00</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>2013-06-25</td>
<td>13:30:00</td>
<td>polA1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4.1: Result Set 1
<table>
<thead>
<tr>
<th>App</th>
<th>Request</th>
<th>Policy</th>
<th>Context</th>
<th>Access (Enforced Policy)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Date</td>
<td>Time</td>
<td>Name</td>
<td>Is policy valid?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>False</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>polB2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>2013-06-10</td>
<td>13:30:00</td>
<td>polA1</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>2013-06-10</td>
<td>13:30:00</td>
<td>polB2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>2013-06-10</td>
<td>15:30:00</td>
<td>polA1</td>
<td>NOT CONSIDERED</td>
</tr>
<tr>
<td>com.bartat.android.elixir</td>
<td>2013-06-10</td>
<td>13:30:00</td>
<td>polB2</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>2013-06-10</td>
<td>13:30:00</td>
<td>polA1</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>2013-06-10</td>
<td>13:30:00</td>
<td>polA1</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>2013-06-25</td>
<td>13:30:00</td>
<td>polB2</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>2013-06-25</td>
<td>16:45:00</td>
<td>polA1</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>2013-07-12</td>
<td>13:30:00</td>
<td>polB2</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>2013-07-12</td>
<td>13:30:00</td>
<td>polA1</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>False</td>
</tr>
<tr>
<td>org.proxydroid</td>
<td>2013-06-10</td>
<td>13:30:00</td>
<td>polB2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>2013-06-10</td>
<td>15:30:00</td>
<td>polB2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>2013-06-25</td>
<td>13:30:00</td>
<td>polC3</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>2013-06-25</td>
<td>16:45:00</td>
<td>polC3</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>2013-06-25</td>
<td>16:45:00</td>
<td>polB2</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>polC3</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>False</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>False</td>
</tr>
<tr>
<td>App</td>
<td>Request</td>
<td>Policy</td>
<td>Context</td>
<td>Access (Enforced Policy)</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------</td>
<td>--------</td>
<td>---------</td>
<td>--------------------------</td>
</tr>
<tr>
<td></td>
<td>Date</td>
<td>Time</td>
<td>Name</td>
<td>Is policy valid?</td>
</tr>
<tr>
<td>org.proxydroid</td>
<td>2013-07-12</td>
<td>13:30:00</td>
<td>polB2</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>polC3</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>polA1</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.2: Result Set 2

<table>
<thead>
<tr>
<th>App</th>
<th>Request</th>
<th>Location</th>
<th>Policy</th>
<th>Context</th>
<th>Access (Enforced Policy)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Date</td>
<td>Time</td>
<td>Available?</td>
<td>Coordinates</td>
<td>Name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td>19.1304, 72.9157</td>
<td>polAA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>polBB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td>19.1304, 72.9157</td>
<td>polAA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>polBB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td>19.1304, 72.9157</td>
<td>polAA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>polBB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No</td>
<td>–</td>
<td>polAA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>polBB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td>19.1351,72.9047</td>
<td>polAA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>polBB</td>
</tr>
</tbody>
</table>

Table 4.3: Result Set 3
Chapter 5

Conclusion and Future Work

5.1 Conclusion

Android Operating System provides coarse-grained access control to its users, which is not enough in many cases - as described in this report. In this project, we studied various security frameworks that extend the default Android security mechanisms. These frameworks allow the users to define (context-aware) fine-grained security rules to deal with their security and privacy concerns. We also compared all the frameworks depending on common attributes of those frameworks.

Since these existing frameworks cannot provide complete solution for the goals described in this report, we proposed and implemented a new policy framework Polork. It continuously monitors the system for apps being launched. The policies defined by the user are enforced on these apps. The app is allowed to execute only if the policy permits, otherwise, it is killed by Polork. Polork is also able to evaluate policies which have context attributes. Current implementation can evaluate context (logical expression) having time and location attributes.

Polork can be used by schools (or employers) to have control over students’ (employees’) devices. They can define policies to allow or restrict apps that can be launched by students (employees) at any particular time and/or location.

5.2 Future Work

- **Integration of Context Attributes in Polork:** Currently, time and location are the only context attributes which can be used while defining policies. The framework can be enhanced by adding more context attributes. Some of the possible attributes are discussed in appendix III. Users will get to define policies based on more attributes, if these are integrated into the framework.

- **Adding Support for Resource/Component Level Policies:** Policies defined for Polork allows or kills launching of an application. In other words, policies are only targeted at application level. It does not allow users define policies for
components of an application or for the various permissions held by that application. For example, if the application PQR has SEND_SMS permission, then policy like ‘Allow PQR, if total SMS sent is less than 10’. Users can have more fine-grained control over the devices if this support is added to Polork.

• **Creating an User Interface to Define Policies:** A web interface can be created, which will be accessed by users (administrators) to create/modify/update policies. Policy files will be automatically generated from newly created/modified policies and downloaded to the devices.

• **Implementing Polork as a System Service:** It is also possible to implement Polork as a system service. The advantages and disadvantages of doing so are already discussed in section 3.5. Implementation related details on how system service can be created are given in appendix III.
Appendix I

Policies - XML Schema

Policies created by users can be validated against following XML schema:

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:simpleType name="PolicyNameType">
        <xs:restriction base="xs:string">
            <xs:maxLength value="20"/>
            <xs:pattern value="[a-zA-Z][a-zA-Z0-9-\_]+"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="PriorityType">
        <xs:restriction base="xs:integer">
            <xs:minInclusive value="1"/>
            <xs:maxInclusive value="5"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="CmdType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="ALLOW"/>
            <xs:enumeration value="ALLOW_EXCEPT"/>
            <xs:enumeration value="ALLOW_ALL"/>
            <xs:enumeration value="DENY"/>
            <xs:enumeration value="DENY_EXCEPT"/>
            <xs:enumeration value="DENY_ALL"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="CATimeSimpleType">
        <xs:restriction base="xs:string">
            <xs:pattern value="[00-23]:[00-59]:[00-59], [00-23]:[00-59]:[00-59]"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```
<xs:restriction>
<xs:simpleType>
<xs:complexType name="CATimeType">
<xs:simpleContent>
<xs:extension base="CATimeSimpleType">
<xs:attribute name="name" use="required" /></xs:extension>
</xs:simpleContent></xs:complexType></xs:simpleType>

<xs:complexType name="CALocSimpleType">
<xs:restriction base="xs:string">
<xs:pattern value="" /></xs:restriction></xs:simpleType>

<xs:complexType name="LocReqdType">
<xs:restriction base="xs:string">
<xs:enumeration value="yes" /></xs:restriction></xs:simpleType>

<xs:complexType name="CALocType">
<xs:simpleContent>
<xs:extension base="CALocSimpleType">
<xs:attribute name="name" use="required" />
<xs:attribute name="strictly-reqd" type="LocReqdType" default="no" /></xs:extension>
</xs:simpleContent></xs:complexType>

<xs:complexType name="ConAttr">
<xs:sequence>
<xs:element name="context" type="xs:string" minOccurs="0" />
<xs:element name="ca-time" type="CATimeType" minOccurs="0" maxOccurs="unbounded" />
<xs:element name="ca-location" type="CALocType" minOccurs="0" maxOccurs="unbounded" /></xs:sequence></xs:sequence>

<xs:complexType name="PkgType">
<xs:restriction base="xs:string" />
</xs:complexType>
    <xs:pattern value="[a-z][a-z.]+" />
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="ListType">
  <xs:sequence>
    <xs:element name="pkg" type="PkgType" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="PolicyType">
  <xs:sequence>
    <xs:element name="name" type="PolicyNameType" />
    <xs:element name="valid-from" type="xs:dateTime" />
    <xs:element name="valid-till" type="xs:dateTime" />
    <xs:element name="priority" type="PriorityType" />
    <xs:element name="seq-num" type="xs:positiveInteger" />
    <xs:element name="cmd" type="CmdType" />
    <xs:group ref="ConAttr" />
    <xs:element name="list" type="ListType"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="policies">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="policy" type="PolicyType" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
Appendix II

Android Programming

I Killing a Process

This section lists all the possible methods that can be used to kill a process. Some of them are required to be executed as system user, while some of them can be executed after gaining root access.

I.1 Using ActivityManager

ActivityManager service can be used to kill background processes. The app should have ‘android.permission.KILLBACKGROUND_PROCESSES’ permission, otherwise the system will throw an exception. Note that only background processes can be killed, so if the app is in foreground, ‘killBackgroundProcesses’ function cannot be used to kill its process.

Code snippet:

```java
import android.app.ActivityManager;

private void someFunction() {
    ActivityManager am = (ActivityManager)
            context.getSystemService(context.ACTIVITY_SERVICE);
    // 'pkgName' is the package name of an app to be killed...
    am.killBackgroundProcesses(pkgName);
}
```

I.2 Using IActivityManager

IActivityManager can also be used to kill the process. Any app, even one running in foreground, can be killed using ‘killApplicationWithUid’ function. But, the app or service calling this function should have system uid. So, to make use of this function call, our framework should be implemented as a system service.

Code snippet:

```java
import android.app.IActivityManager;
```
private void someFunction() {
    IActivityManager iActMngr = ActivityManagerNative.getDefault();
    // 'pkgName' and 'uid' are package name and user id of an app to be killed...
    iActMngr.killApplicationWithUid(pkgName, uid);
}

I.3 Using Process

Many online tutorial suggest that ‘killProcess’ function can be used to kill a process.
But, it is not possible for an app to kill the processes other than its own, as mentioned
in Android documentation [23]. Hence, ‘killProcess’ is not used in our framework.

    Code snippet:

private void someFunction() {
    // 'pid' is process id of an app to be killed...
    android.os.Process.killProcess(pid);
    // Instead of killProcess, sendSignal can also be used...
    android.os.Process.sendSignal(pid, android.os.Process.SIGNAL_QUIT);
    android.os.Process.sendSignal(pid, android.os.Process.SIGNAL_KILL);
}

I.4 Using Root Permission

If the device is rooted, any app can ask for root permissions. A linux command ‘kill’
can be used to kill any process under sudo permission. We need to pass pid (process id) of
an app to be killed. We are using this method to kill the processes.

    Code snippet:

private void someFunction() {
    // Get root user access and use 'kill -9' to kill a process
    Process p = Runtime.getRuntime().exec("su");
    DataOutputStream os = new DataOutputStream(p.getOutputStream());
    os.writeBytes("kill -9 " + pid + "\n");
    os.flush();
    os.close();
}
II Disabling the Applications

Whenever an app is launched, and one of the policies denies the request, we kill the process of that app. We can go on step further and disable that app. Disabling of an app is called as ‘Freezing’ in Android terminology. The explanation given below is taken from [6], [3]:

- Freezing an app makes it disabled in the system. The app is still installed, but it cannot run (cannot be opened, turned on, accessed, etc).
- The benefit of doing this can be preventing it from running in the background, downloading data and killing your battery when you don’t even use it.
- Freezing has the same benefits as uninstalling while allowing you to get updates.
- It doesn’t clear up space on your phone, the application files are still where they are.

There are few apps available in Google’s Play Store which are capable of disabling the app:

- AntTek App Manager allows user to freeze the apps. It does so by renaming apk file of an app to be disabled, which is present in /data/app directory.
- Apps like Link2SD, Titanium Backup, Elixier2 are also capable of disabling the app, but the method followed to do so is unknown.
- Android SDK provides an API setApplicationEnabledSetting(...) to disable the application.

Android API

As mentioned above, an API setApplicationEnabledSetting(...), available in Android SDK, can be used to disable the application. But, the app must have “android.permission.CHANGE_COMPONENT_ENABLED_STATE” permission and it should be a system app.

Code snippet:

```java
import android.content.pm.PackageManager;

private void someFunction() {
    PackageManager pm = context.getPackageManager();
    // To disable an app having package name 'pkgName'
    pm.setApplicationEnabledSetting(pkgName,
        pm.COMPONENT_ENABLED_STATE_DISABLED, 0);
    // To restore the state of an app...
    pm.setApplicationEnabledSetting(pkgName, pm.COMPONENT_ENABLED_STATE_DEFAULT, 0);
}
```
It is necessary to enable disabled apps after the corresponding policy becomes invalid. If not enabled, user has no option to launch that app. A database can be maintained to keep track of all disabled apps, so that those apps can be enabled again later.

III Implementing Polork as a System Service

We have implemented Polork as an Android application. We have already compared different possible implementations in section 3.5.

Whenever an app is launched, Android’s ‘ActivityManagerService’ (AMS) [1] is called, which creates a new process for that app. startProcessLocked(...) function is called by AMS, which calls Process.start(...) [4]. We can modify this function to call Polork’s service, passing package name and uid of an app being launched. Polork will then evaluate the policies and return the result. If the result is false, it means that app is not allowed to be launched. The control will be returned from that point skipping the call to Process.start(...).

Tutorials on how to create system service by modifying Android source code are available on [7], [2], [8].
Appendix III

Additional Context Attributes

This chapter explains some of the (system) properties which can be used as context attributes. We have not included these attributes in our framework, but the supplementary code is provided, wherever possible, so that these attributes can be incorporated very easily.

I Battery Level

The current battery level (0% to 100%) can be used as a context attribute. Admin can create policies such as:

- Allow app x.y.z, if battery level is more than 50%
- Block app p.q.r, if battery level is less than 30%

The current battery level can be obtained as mentioned below:

```
// http://stackoverflow.com/a/5097828
BroadcastReceiver batteryReceiver = new BroadcastReceiver() {
  @Override
  public void onReceive(Context context, Intent intent) {
    int level = intent.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
    int maxLevel = intent.getIntExtra(BatteryManager.EXTRA_SCALE, -1);
    float batteryPercentage = ((float)level / (float)maxLevel) * 100;
    // Do something with batteryPercentage (lies between 0-100).
  }
};
// Register the Receiver in some function...
IntentFilter filter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
registerReceiver(batteryReceiver, filter);
```
II Battery Consumption Per App

The battery consumed by particular app can be a deciding factor while enforcing a policy. Policies such as

- Allow app x.y.z, if battery consumed by that app is less than 25%
- Block app p.q.r, if battery consumed by that app is more than 10%

can be created by admin. In Android, top 10 battery consuming components (apps/services) are displayed under ‘Battery’ tab of ‘Settings’ app. We need to make same information accessible to our framework.

The tests performed on tablets like Motorola Xoom, Aakash, Micromax Funbook suggest that this attribute should not be used while creating a policy. The battery consumed by any app hardly reaches 10% mark, which is not useful in many cases. Also, what happens to the battery consumption level when the device is connected for charging, still needs to be investigated.

III Application Foreground Time

Application foreground time means the time for which that app is displayed to the end-user so that she can interact with it. Admin can specify for how much time a particular app can be used daily. Some examples of how policies can be created:

- Allow app x.y.z, if foreground time is less than 100min%
- Block app p.q.r, if foreground time is more than 75min%

No system service in Android keeps track of application’s foreground time. Instead, we can build a service to do so, using the following code:

```java
import android.app.IProcessObserver;

private IProcessObserver.Stub mProcessObserver = new IProcessObserver.Stub() {

@Override
public void onForegroundActivitiesChanged(int pid, int uid, boolean foregroundActivities) throws RemoteException {
    // Logic here...
}

@Override
public void onProcessDied(int pid, int uid) throws RemoteException {
    // ...
}
};
```
// Register the ProcessObserver instance in some function...
ActivityManagerNative.getDefault().registerProcessObserver(mProcessObserver);

Note that, it is not possible to compile this code simply using Android SDK, since it does not have a reference for ‘IProcessObserver’. Application’s source code should be added to Android source code for successful compilation. The tutorial on how to setup and build Android can be found at [22].

In the given code,

- **pid**: Process id of an app
- **uid**: User id of an app (Every app has unique uid)
- **foregroundActivities**: Its value (true) means the app having user id (uid) and process id (pid) has been brought to the foreground. Value (false) indicate that the app is no longer in the foreground.

So, whenever a new app is launched, Android system calls `onForegroundActivitiesChanged` two times. First call has record for an app which is sent to background (`foregroundActivities` is set to false). Second call is for an app which is presently in the foreground (`foregroundActivities` is set to true). We can save this information in a database, along with the time at which calls are made, to get application’s foreground time.

### IV Data Usage

The total data transferred by the device (using WiFi, GPRS, 3G, etc.) is counted under data usage. Some examples are:

- Allow app x.y.z, if data usage is less than 20MB%
- Block app p.q.r, if data usage is more than 35MB%

```java
import android.net.TrafficStats;
// ...
private void someFunction() {
    // User id (uid) of an app is required to get data usage...
    long tx = TrafficStats.getUidTxBytes(uid);
    long rx = TrafficStats.getUidRxBytes(uid);
}
```

The given code can be used to get total data sent (variable ‘tx’) and received (variable ‘rx’) by an app having user id ‘uid’. The disadvantage of using this as a context attribute is that the app may get killed while the data transfer is in progress. So, when the app starts next time, the whole file has to be transferred once again.
References


Acknowledgements

I would like to express my deepest gratitude to my guide, Prof. D. B. Phatak for his patience and guidance throughout the project. I would like to thank Mr. Nagesh Karmali for his continuous inputs in my work. I would also like to thank Ms. Firuza Aibara for reading my report and giving valuable feedback. I would also like to thank every one else who supported me in this work.