A Policy Enforcement Framework for Android

Stage I Report

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
## Contents

1 Introduction ................................. 1
   1.1 Motivation .............................. 1

2 Basics of Android System .................. 3
   2.1 Android Architecture .................. 3
   2.2 Android Security ...................... 4

3 Policy Enforcement Frameworks .......... 5
   3.1 Kirin .................................. 5
   3.2 Saint ................................ 7
   3.3 Apex .................................. 8
   3.4 CRePE ................................ 9
   3.5 XManDroid .............................. 11
   3.6 TrustDroid ............................. 12
   3.7 ConUCON ............................... 13
   3.8 YAASE ................................. 14
   3.9 SEAF ................................ 15
   3.10 Comparison ............................ 16

4 Problem Formulation ....................... 20
   4.1 Proposed Solution ..................... 20

5 Conclusion and Future Work .............. 21
   5.1 Conclusion ............................. 21
   5.2 Looking Ahead .......................... 21
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. But, 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 users to define security policies for malware mitigation, resource usage control or privilege escalation attack prevention. We first 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 solve those issues.
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 [12]. Developers can develop free or paid apps for Android using Android SDK. Developers 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. But 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. Users can see which permissions are requested by an app at the time of installation. But even user’s decision has to be all-or-nothing, meaning 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 its not possible to impose the constraints on the usage of the resources. We focus on extending Android’s security mechanism to allow users to have fine-grained access control on their device.

1.1 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, 9:00am to 3:30pm).
• Every subject/course may require different set of apps. Teachers should be able to define which apps are 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 [11], CRePE [5], and ConUCON [1] 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.

Let’s consider an example where a user watches videos using MX Player app. 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 security rules based on relative battery consumption of applications.

Remote Access Mechanism
CRePE [5] and TrustDroid [4] allow third parties to define the policies remotely. Users can send the policies to the device using mediums like SMS, WiFi, Bluetooth. SMS is a paid service and WiFi would require the device to query the server repeatedly to get policy updates.

No framework has considered the option of “Google Cloud Messaging for Android” (GCM) [14]. GCM is a free service from Google that helps developers send data from servers to their Android applications on Android devices. GCM is a push mechanism that app developers can use to notify their apps about availability of new data. Once the device gets the notification, it contacts app developer’s server to download the data. If the device is offline, the notification message is stored on Google GCM servers till the device becomes online.

GCM can be used by schools, colleges to send policy updates to students’ tablets. Whenever these tablets connect to WiFi, new policy updates will be downloaded and enforced by policy enforcement framework present in the device.
Chapter 2

Basics of Android System

This chapter talks about basic components in any Android Application and about Android security framework.

2.1 Android Architecture

Any Android application has four components [6]:

- **Activity.** 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 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 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 permissions 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 [8]. Application developers can add broadcast receiver in their applications to receive events occurring at 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 [15].

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 [6].

2.2 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 world-wide 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 [11]. 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[11].

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 to protect 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 [10].

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 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.
Chapter 3

Policy Enforcement Frameworks

Policy enforcement frameworks provide fine-grained access control to the device. 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.

3.1 Kirin

Kirin [8] 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 Diagram](image)

Figure 3.1: Kirin [8]

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 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 \( r_i \in \mathcal{R}, P_i \in 2^P \) and \( A_i \in 2^A \).

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 \( \text{fail} : C \times \mathcal{R} \rightarrow \{\text{true}, \text{false}\} \) 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:

\[
\text{(P_t, A_t) = c_t, (P_i, A_i) = r_i, P_i \subseteq P_t \land A_i \subseteq A_t}
\]

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:

\[
F_R(c_t) = \{r_i \mid r_i \in R, \text{fail}(c_t, r_i)\}
\]

Then, the configuration \( c_t \) passes a given KSL rule-set \( R \) if \( F_R(c_t) = \emptyset \). 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:

\[
\langle \text{rule - set} \rangle ::= \langle \text{rule} \rangle | \langle \text{rule} \rangle \langle \text{rule - set} \rangle \quad (1)
\]

\[
\langle \text{rule} \rangle ::= \text{“restrict”} \ \langle \text{restrict - list} \rangle \quad (2)
\]

\[
\langle \text{restrict - list} \rangle ::= \langle \text{restrict} \rangle | \langle \text{restrict} \rangle \text{ “and” } \langle \text{restrict - list} \rangle \quad (3)
\]

\[
\langle \text{restrict} \rangle ::= \text{“permission”} [^\text{\textquoteleft	extquoteleft} \langle \text{const - list} \rangle \text{\textquoteright	extquoteright} \text{\textquoteleft	extquoteleft}] | \text{“receive”}[^\text{\textquoteleft	extquoteleft} \langle \text{const - list} \rangle \text{\textquoteright	extquoteright}] \quad (4)
\]

\[
\langle \text{const - list} \rangle ::= \langle \text{const} \rangle | \langle \text{const} \rangle \text{ “} \langle \text{const - list} \rangle \text{ “} \quad (5)
\]

\[
\langle \text{const} \rangle ::= ^\text{\textquoteleft	extquoteright}[^A - Z a - z 0 - 9.] + ^\text{\textquoteleft	extquoteright} \quad (6)
\]

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

\[
\text{restrict permission PHONE\_STATE and permission RECORD\_AUDIO and permission INTERNET(1)}
\]

\[
\text{restrict permission ACCESS\_FINE\_LOCATION and permission INTERNET and permission RECEIVE\_BOOT\_COMPLETE(2)}
\]

\[
\text{restrict permission SET\_PREFERRED\_APPLICATION and receive CALL(3)}
\]

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).
3.2 Saint

Saint (Secure Application INTeraction) [11] 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.).

![Figure 3.2: Saint](image)

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:

(\text{permission} - \text{label}) (\text{owner}) [!]cond_1 [[!]cond_2] ... [[!]cond_n] \quad (1)
(expose\text{\(\mid\)access}) (\text{sourceapp}, \text{type}, \text{action}) (\text{destinationapp}, \text{component}) [!]cond_1 [[!]cond_2] ... [[!]cond_n] \quad (2)

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:

(com.xyz.abc) (com.xyz.loc) \text{required-permission(ACCESS\_FINE\_LOCATION)}
(access) (com.test.pqr, any, GET\_LOC) (any, any) \text{forbid-permissions(INTERNET)}

In first example, permission \text{com.xyz.abc} declared by \text{app 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}.

3.3 Apex

Apex (Android Permission EXtension) [10] 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 [10]) are given here:

- **Application State.** An application state is a function \( \tau : \eta(A) \rightarrow \text{dom}(\eta(A)) \), 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 predicates, 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:

\[
\rho(c_1, c_2, i) \iff A_p(c_2) = null \lor
\exists p \in P, a \in A.a = s(c_1) \land
p = A_p(c_2) \land p \in \mu(a) \land
i \in A_f(c_2) \land
\neg \exists l \in \Lambda.l(a, p) = \text{deny}
\]

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

```plaintext
sms_count_allow ("com.test.SmsApp" as SMSApp,
    "android.permission.SEND_SMS" as SMS):
    SMSApp.sentSms <= 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 \land SMSApp.lastUsedDay = System.CurrentDay
    -> deny(SMSApp, SMS);

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

### 3.4 CRePE

CRePE [5] 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.

![CRePE Diagram](image)

Figure 3.3: CRePE

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 stores 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 $R = (r, \langle access \rangle)$ 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.

### 3.5 XManDroid

The goal of “Extended Monitoring on Android” (XManDroid) [3] 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.
3.6 TrustDroid

TrustDroid [4] 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 [3] 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.
3.7 ConUCON

“Context-Aware Usage Control” (ConUCON) [1] 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 [1].

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, <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>(¬[(01/01/2012 00:00:00, 12/31/2012 23:59:59], weeks+ 1, 4.day + 15hours &gt; 2hours)) ∨ (~meetingroom))</td>
</tr>
<tr>
<td>Ongoing-context</td>
<td>(¬[(01/01/2012 00:00:00, 12/31/2012 23:59:59], weeks+ 1, 4.day + 15hours &gt; 2hours)) ∨ (~meetingroom))</td>
</tr>
<tr>
<td>Update</td>
<td>if (forbidden) lastForbiddenTime = currentTime</td>
</tr>
</tbody>
</table>

Table 3.1: ConUCON Policy Example

Say, user has a meeting on every Monday and Thursday from 3pm to 5pm (Table 3.1). 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. (Subject:All) tells the security framework to enforce this policy for all applications. (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’.

3.8 YAASE

“Yet Another Android Security Extension” (YAASE) model [13] 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 [7] 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:

```
PolicyName: Requester can do operation on Resource
            [have to perform action]
            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
  have to perform filterOut(‘’Pr’’, returnData);
  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.

3.9 SEAF

“Security Enhanced Android Framework” (SEAF) [2] helps the user to evaluate the behavior of an application. It monitors the permission exercising patterns of the applications to inform the user about any potentially harmful activity. Dangerous behavior of applications are captured and blocked to mitigate the malware threats to the smartphone. 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 [2]:

- Application access resources through a set of permissions. We define function that maps an application A to a set of those requested permissions P.
  \[ \beta(A) \to (P) \]
  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:

\[
\delta(P_1 \rightarrow P_2) = H \\
\delta(P_2 \rightarrow P_1) = B 
\]

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.

### 3.10 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><strong>Points</strong></th>
<th><strong>Kirin [8]</strong></th>
<th><strong>Saint [11]</strong></th>
<th><strong>Apex [10]</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Aim</strong></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><strong>Framework design summary</strong></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 - Impose runtime constraints on usage of resources</td>
</tr>
<tr>
<td><strong>Framework design details</strong></td>
<td>By comparing security policies against permissions and action strings defined in app’s manifest file</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><strong>Policy definitions</strong></td>
<td>-</td>
<td>I-T policy definitions</td>
<td>I-T and R-T permission selection, constraints definitions</td>
</tr>
<tr>
<td><strong>Policy enforcement</strong></td>
<td>I-T enforcement</td>
<td>I-T and R-T enforcement</td>
<td>R-T enforcement</td>
</tr>
<tr>
<td><strong>Policies defined by</strong></td>
<td>End-user</td>
<td>App developer</td>
<td>End-user</td>
</tr>
<tr>
<td><strong>(Additional) file reqd.</strong></td>
<td>AndroidManifest.xml</td>
<td>Policy configuration file in apk</td>
<td>-</td>
</tr>
<tr>
<td><strong>Decisions</strong></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><strong>What can be blocked?</strong></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><strong>Parameters considered for policy</strong></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><strong>Modifications</strong></td>
<td>Package Installer</td>
<td>Framework + Package Installer</td>
<td>Framework + Package Installer</td>
</tr>
<tr>
<td><strong>Policies defined for</strong></td>
<td>Caller</td>
<td>Caller, Callee</td>
<td>Caller</td>
</tr>
<tr>
<td><strong>Target</strong></td>
<td>General</td>
<td>General, Specific</td>
<td>Specific</td>
</tr>
</tbody>
</table>

Table 3.2: Comparison
<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 3.3: Comparison
<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 3.4: Comparison
Chapter 4

Problem Formulation

In this project, our aim is to design and implement a context-aware, multi-user policy enforcement framework for Android. 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, etc. The framework should be implemented in such a way that only trusted third parties (schools, teachers, parents) should be able to define policies for the device. Trusted third parties are to be considered at different priority levels to decide which policies to enforce in case policy conflicts. In existing frameworks, policies can be defined and enforced remotely using mediums like SMS, Bluetooth and WiFi. SMS is a paid service, Bluetooth has very limited range and use of WiFi requires continuous polling to the server, which drains the battery quickly. Our enforcement framework should make use of push-based solutions like GCM [14], to allow users to enforce policies remotely.

4.1 Proposed Solution

Based on the comparison done in section 3.10, it is clear that none of the existing security frameworks would be able to provide complete solution to the problems defined above. Some of the frameworks can enforce context-related policies, but they cannot provide multi-user mode. Some frameworks allow multiple users to define security policies, but those security policies cannot satisfy use cases mentioned in section 1.2. Hence, a new policy enforcement framework is required. This new framework would consist of some of the approaches of existing policy enforcement frameworks.

Saint [11] framework is the best suited framework for these problems, but it does not allow trusted third parties to define the policies. Also, a policy enforcement framework should be able to define policies for all the use-cases described in section 1.2. Policies should be context-aware as described in Saint [11], CRePE [5], and ConUCON [1]. This framework should allow schools/colleges, teachers, and parents to define the policies, similar to the models described in CRePE [5], and TrustDroid [4].

The framework will also consider relative battery consumption as a context attribute. Battery-based policies require details of subsystem-level (app-level) battery consumption. How much of total battery was consumed by particular app is required to calculate relative battery consumption. According to a question on StackOverflow [9] and source code of PowerUsageSummary.java file [16] in Android, Android has private API which can be used to get app-level battery consumption details.
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 stage, we studied various security frameworks that extend the default Android security mechanisms. These frameworks allow 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. We then saw how these existing frameworks cannot provide complete solution for the goals described in this report. Last part of the report suggests the solution for the same.

5.2 Looking Ahead

In the next stage we plan to do following tasks:

- **Implementation of policy enforcement model.** In the next stage we plan to implement the proposed solution. It requires deciding efficient and easy-to-understand policy syntax, implementing and deploying GCM [14] service and policy framework on actual mobile device.

- **Battery information for context-aware policies.** Android has inbuilt classes that provide subsystem level application battery consumption, but these APIs are not available as public APIs. Our framework requires access to these private APIs to implement context-aware policies. An effective solution would be to create a service wrapper around these APIs.
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 also like to thank Nagesh Karmali for his continuous inputs for my work. I would also like to thank everyone who supported me in this work.
Bibliography


